CSC 4304 - Systems Programming
Fall 2010

Lecture - XVI
Network Programming - II

Tevfik Koşar

Louisiana State University
November 11th, 2010

Simple Web Server

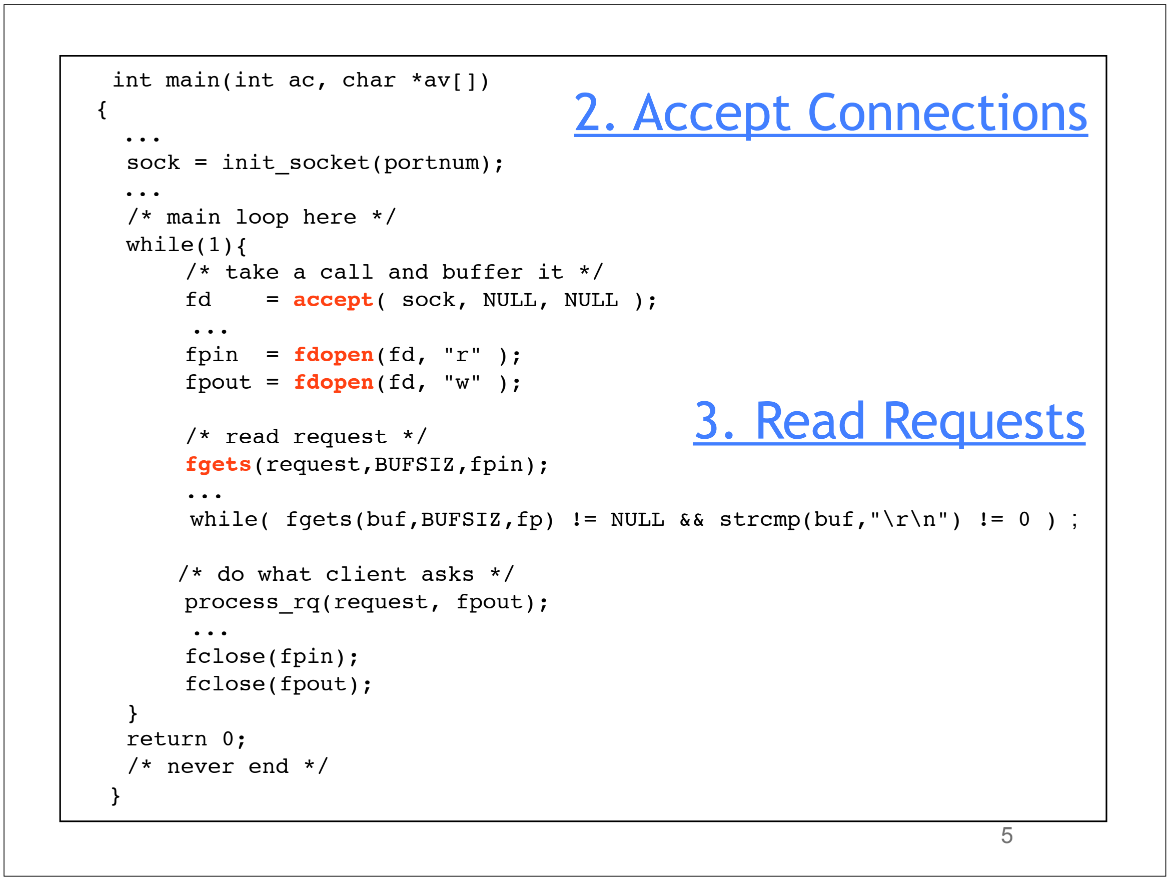
Logic of a Web Server

- 1. Setup the server
  - *socket, bind, listen*
- 2. Accept a connection
  - *accept, fdopen*
- 3. Read a request
  - *fread*
- 4. Handle the request
  - a. directory --> **list it**: *opendir, readdir*
  - b. regular file --> **cat the file**: *open, read*
  - c. .cgi file --> **run it**: *exec*
  - d. not exist --> **error message**
- 5. Send a reply
  - *fwrite*

1. Setup the Server

int init_socket(int portnum)

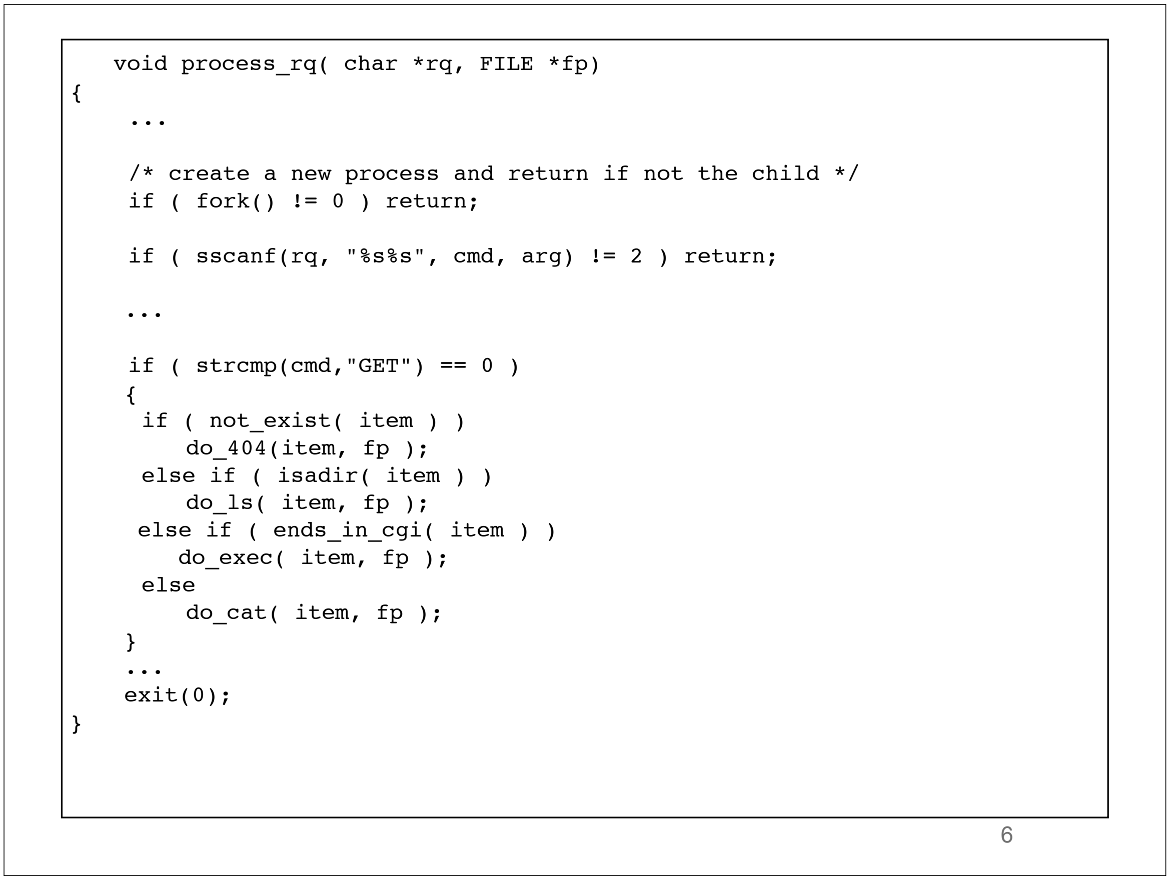
```
{   ...
    gethostname( hostname , 256 );          /* where am I ?         */
    hp = gethostbyname( hostname );         /* get info about host  */
    ...
    bzero( (void *)&saddr, sizeof(saddr) ); /* zero struct & fill host addr*/
    bcopy( (void *)hp->h_addr, (void *)&saddr.sin_addr, hp->h_length);
    saddr.sin_family = AF_INET ;            /* fill in socket type  */
    saddr.sin_port = htons(portnum);        /* fill in socket port  */

    sock_id = socket( AF_INET, SOCK_STREAM, 0 );    /* get a socket */
    ...
    rv = setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
    ...
    bind(sock_id, (struct sockaddr *) &saddr, sizeof(saddr));
    ...

    listen(sock_id, 1) != 0 );
    ...
    return sock_id;
 }
```



2. Accept Connections

3. Read Requests

```
 int main(int ac, char *av[])
{
  ...
  sock = init_socket(portnum);
  ...
  /* main loop here */
  while(1){
      /* take a call and buffer it */
      fd    = accept( sock, NULL, NULL );
       ...
      fpin  = fdopen(fd, "r" );
      fpout = fdopen(fd, "w" );

      /* read request */
      fgets(request,BUFSIZ,fpin);
      ...
       while( fgets(buf,BUFSIZ,fp) != NULL && strcmp(buf,"\r\n") != 0 ) ;

     /* do what client asks */
      process_rq(request, fpout);
       ...
      fclose(fpin);
      fclose(fpout);
  }
  return 0;
  /* never end */
 }
```



```
   void process_rq( char *rq, FILE *fp)
{
    ...

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

    if ( sscanf(rq, "%s%s", cmd, arg) != 2 ) return;

    ...

    if ( strcmp(cmd,"GET") == 0 )
    {
     if ( not_exist( item ) )
        do_404(item, fp );
     else if ( isadir( item ) )
        do_ls( item, fp );
    else if ( ends_in_cgi( item ) )
       do_exec( item, fp );
     else
        do_cat( item, fp );
    }
    ...
    exit(0);
}
```

4.a List Dir

```
void do_ls(char *dir, FILE *fp)
{
        int     fd;     /* file descriptor of stream */

        header(fp, "text/plain");
        fprintf(fp,"\r\n");
        fflush(fp);

        fd = fileno(fp);
        dup2(fd,1);
        dup2(fd,2);
        fclose(fp);
        execlp("ls","ls","-l",dir,NULL);
        perror(dir);
}
```

4.b Cat File

```
   void do_cat(char *f, FILE *fpsock)
{
    char *extension = file_type(f);
    char *content = "text/plain";
    FILE *fpfile;
    int c;

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

    fpfile = fopen( f , "r");
    if ( fpfile != NULL )
    {

        fprintf(fpsock, "HTTP/1.0 200 OK\r\n");
        fprintf(fpsock, "Content-type: %s\r\n", content );
        fprintf(fpsock, "\r\n");
        while( (c = getc(fpfile) ) != EOF )
            putc(c, fpsock);
        fclose(fpfile);
    }
}
```

4.c Exec Prog

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

        header(fp, NULL);
        fflush(fp);

        dup2(fd, 1);
        dup2(fd, 2);
        fclose(fp);
        execl(prog,prog,NULL);
        perror(prog);
}
```

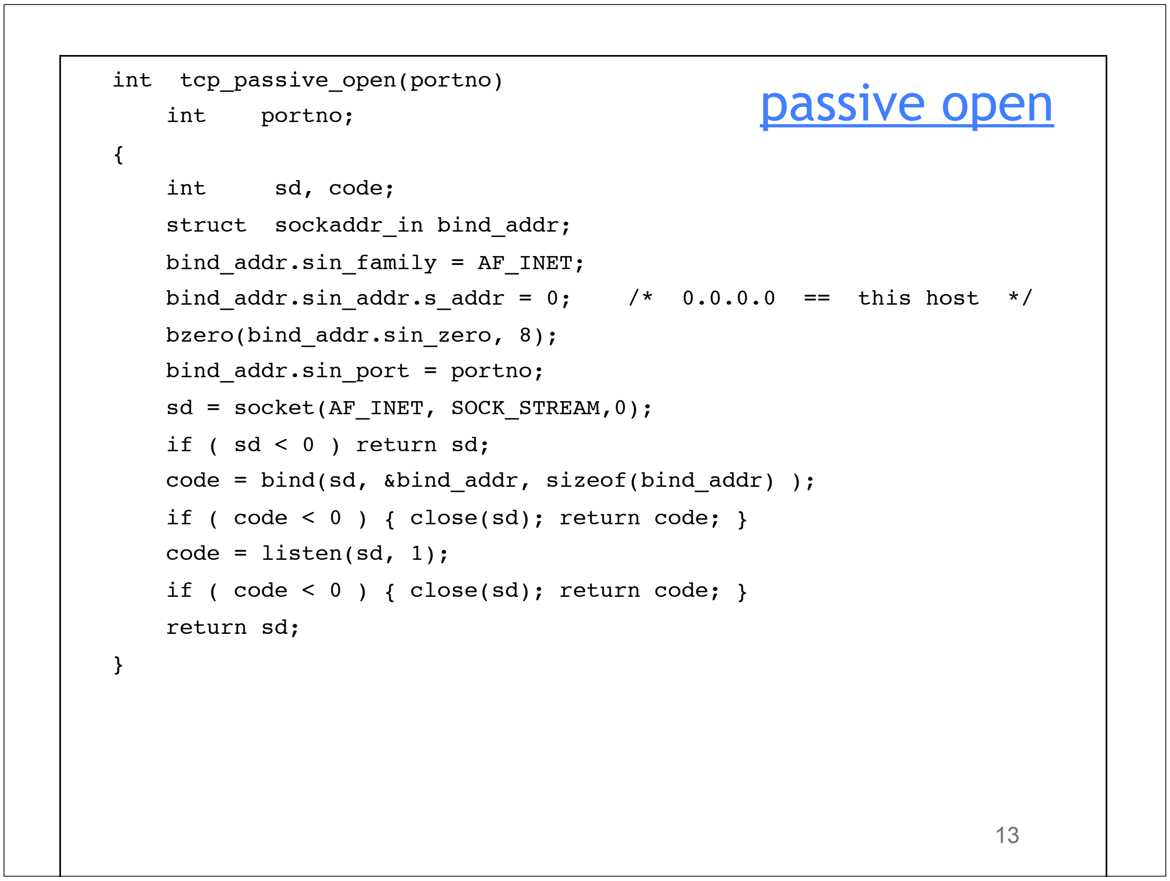
4.d Error!

```
void do_404(char *item, FILE *fp)
{
        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);
        fflush(fp);
}
```

Client Server Programming

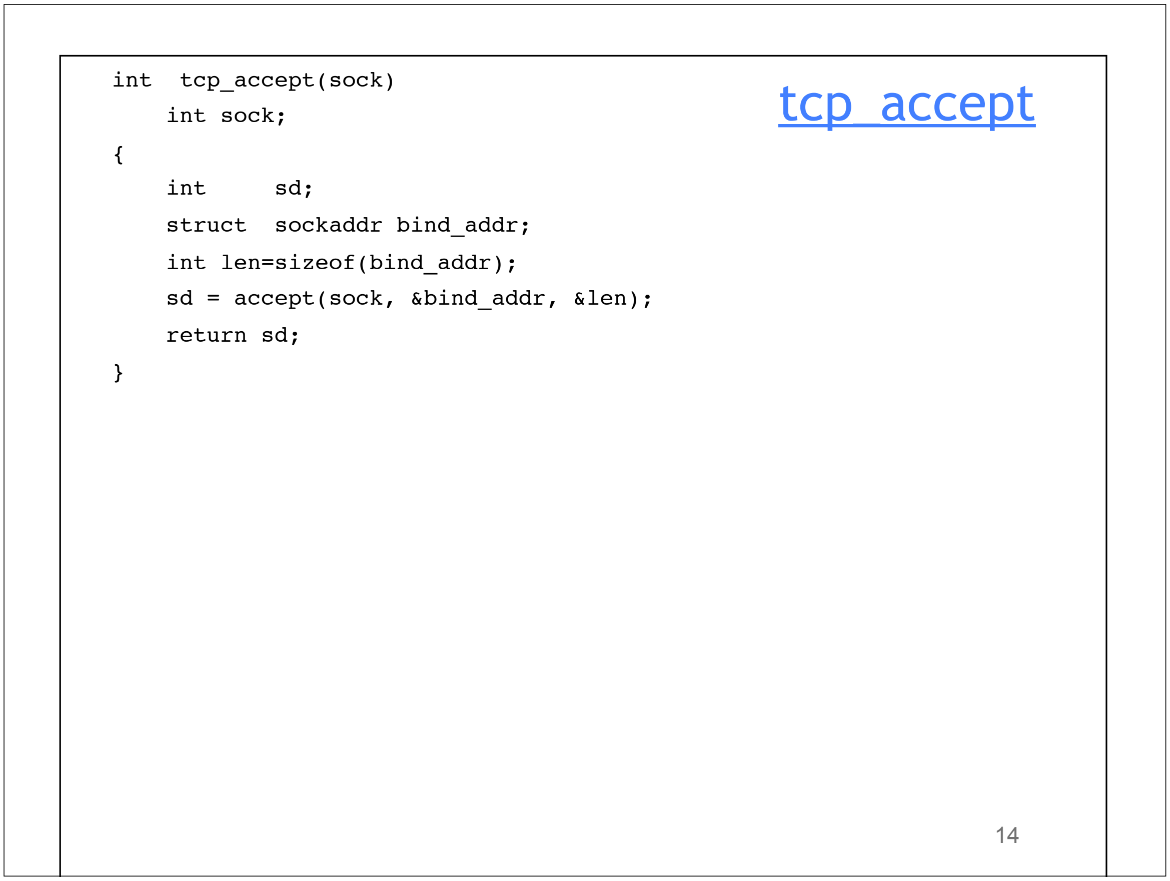
1.server code

```
main(int argc, char **argv){
    int    len, port_sk,  client_sk;
    char *errmess;

    port_sk = tcp_passive_open(port);  /*  establish port  */
    if ( port_sk < 0 ) { perror("socket"); exit(1); }
    printf("start up complete\n");

    client_sk = tcp_accept(port_sk);  /*  wait for client to connect  */

    close(port_sk);  /*  only want one client, so close port_sk  */

    for(;;) {  /*  talk to client  */
        len = read(client_sk,buff,buf_len);  //listen
        printf("client says: %s\n",buff);
        ....
        if ( gets(buff) == NULL ) {    /* user typed end of file  */
            close(client_sk); break;
        }
        write(client_sk,buff,strlen(buff));    //server’s turn
    } exit(0);
}
```



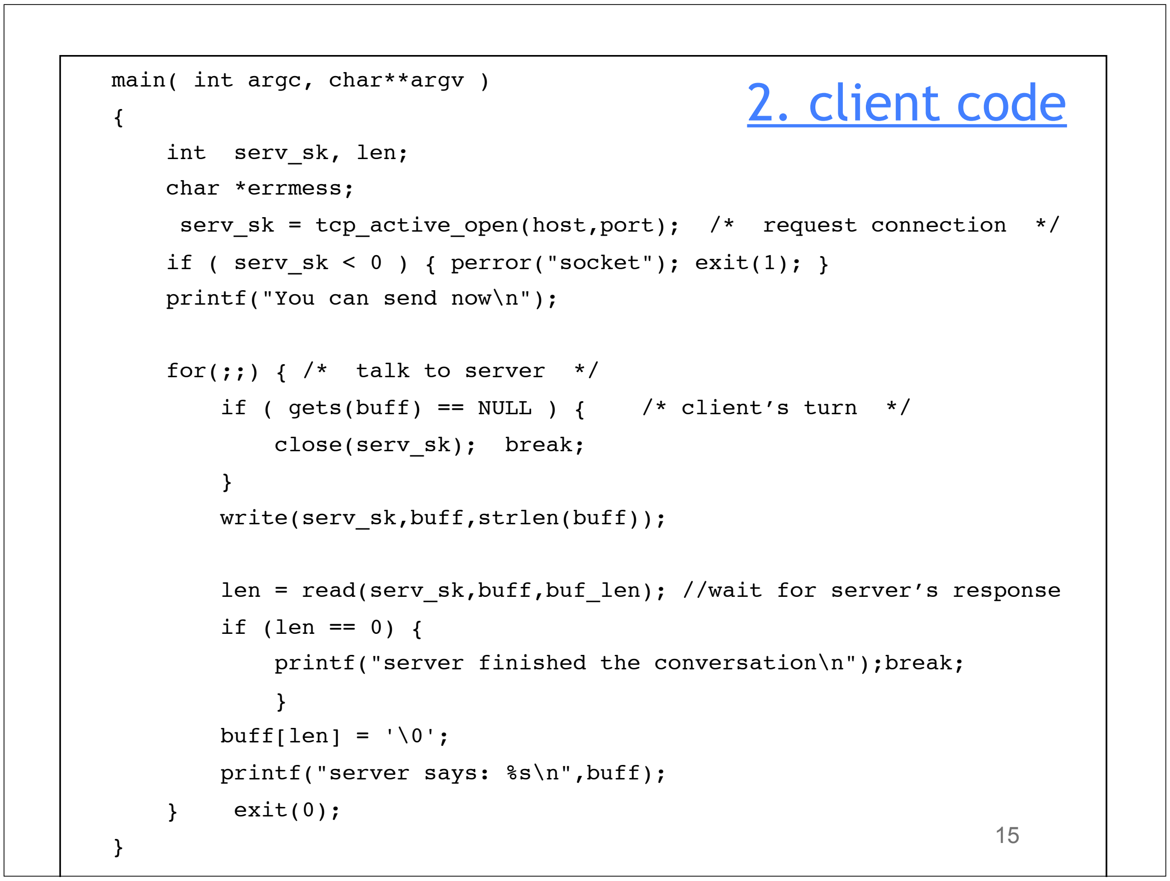
passive open

```
int  tcp_passive_open(portno)
    int    portno;
{
    int     sd, code;
    struct  sockaddr_in bind_addr;
    bind_addr.sin_family = AF_INET;
    bind_addr.sin_addr.s_addr = 0;    /*  0.0.0.0  ==  this host  */
    bzero(bind_addr.sin_zero, 8);
    bind_addr.sin_port = portno;
    sd = socket(AF_INET, SOCK_STREAM,0);
    if ( sd < 0 ) return sd;
    code = bind(sd, &bind_addr, sizeof(bind_addr) );
    if ( code < 0 ) { close(sd); return code; }
    code = listen(sd, 1);
    if ( code < 0 ) { close(sd); return code; }
    return sd;
}
```



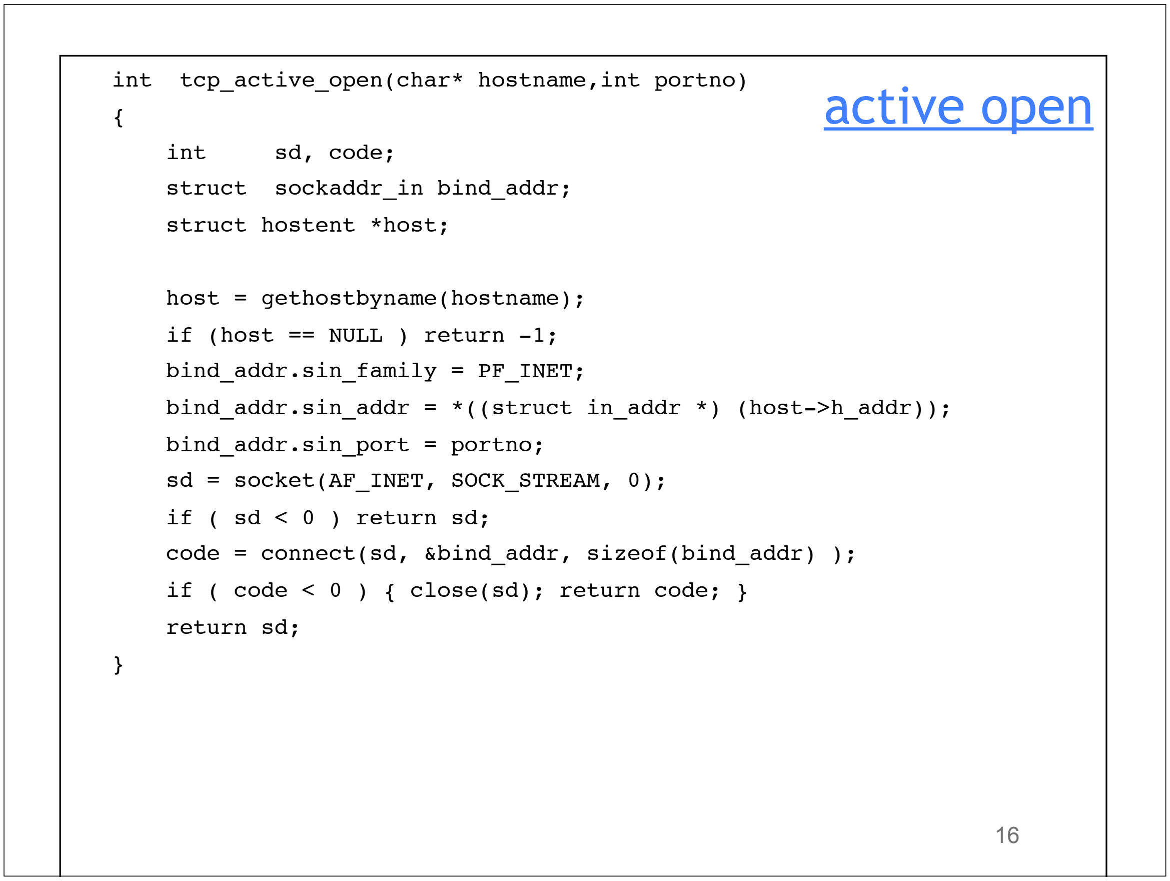
tcp_accept

```
int  tcp_accept(sock)
    int sock;
{
    int     sd;
    struct  sockaddr bind_addr;
    int len=sizeof(bind_addr);
    sd = accept(sock, &bind_addr, &len);
    return sd;
}
```



2. client code

```
main( int argc, char**argv )
{
    int  serv_sk, len;
    char *errmess;
     serv_sk = tcp_active_open(host,port);  /*  request connection  */
    if ( serv_sk < 0 ) { perror("socket"); exit(1); }
    printf("You can send now\n");

    for(;;) { /*  talk to server  */
        if ( gets(buff) == NULL ) {    /* client’s turn  */
            close(serv_sk);  break;
        }
        write(serv_sk,buff,strlen(buff));

        len = read(serv_sk,buff,buf_len); //wait for server’s response
        if (len == 0) {
            printf("server finished the conversation\n");break;
            }
        buff[len] = '\0';
        printf("server says: %s\n",buff);
    }    exit(0);
}
```



active open

```
int  tcp_active_open(char* hostname,int portno)
{
    int     sd, code;
    struct  sockaddr_in bind_addr;
    struct hostent *host;

    host = gethostbyname(hostname);
    if (host == NULL ) return -1;
    bind_addr.sin_family = PF_INET;
    bind_addr.sin_addr = *((struct in_addr *) (host->h_addr));
    bind_addr.sin_port = portno;
    sd = socket(AF_INET, SOCK_STREAM, 0);
    if ( sd < 0 ) return sd;
    code = connect(sd, &bind_addr, sizeof(bind_addr) );
    if ( code < 0 ) { close(sd); return code; }
    return sd;
}
```

Daemon Processes

Daemon Characteristics

Commonly, dæmon processes are created to offer a specific service.

Dæmon processes usually

- live for a long time
- are started at boot time
- terminate only during shutdown
- have no controlling terminal

The previously listed characteristics have certain implications:

- do one thing, and one thing only
- no (or only limited) user-interaction possible
- consider current working directory
- how to create (debugging) output

Writing a Daemon

- fork off the parent process
- change file mode mask (umask)
- create a unique Session ID (SID)
- change the current working directory to a safe place
- close (or redirect) standard file descriptors
- open any logs for writing
- enter actual dæmon code

Example Daemon Creation

```
int daemon_init(void)
{
   pid_t pid;
   if ((pid=fork())<0)  return (-1);
   else if (pid!=0) exit (0); //parent goes away
   setsid(); //becomes session leader
   chdir(“/”); //cwd
   umask(0);   //clear file creation mask
return (0)
}
```

Daemon Logging

A daemon cannot simply print error messages to the terminal or standard error. Also, we would not want each daemon writing their error messages into separate files in different formats. A central logging facility is needed.

There are three ways to generate log messages:

- via the kernel routine `log(9)`
- via the userland routine `syslog(3)`
- via UDP messages to port 514

Syslog()

`openlog(3)` allows us to set specific options when logging:

- prepend *ident* to each message
- specify logging options (`LOG_CONS | LOG_NDELAY | LOG_PERRO | LOG_PID`)
- specify a *facility* (such as `LOG_DAEMON`, `LOG_MAIL` etc.)

`syslog(3)` writes a message to the system message logger, tagged with *priority*. A *priority* is a combination of a *facility* (as above) and a *level* (such as `LOG_DEBUG`, `LOG_WARNING` or `LOG_EMERG`).

Acknowledgments

- Advanced Programming in the Unix Environment by R. Stevens
- The C Programming Language by B. Kernighan and D. Ritchie
- Understanding Unix/Linux Programming by B. Molay
- Lecture notes from B. Molay (Harvard), T. Kuo (UT-Austin), G. Pierre (Vrije), M. Matthews (SC), B. Knicki (WPI), M. Shacklette (UChicago), J. Kim (KAIST), A. Dix (Hiraeth), and J. Schaumann (SIT).