

**BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI (RAJASTHAN)**  
**IS F462 – Network Programming**  
**Lab#7**

---

## **TCP Webserver**

Consider the following code for a very **simple web server**. It takes port number as the input over which it listens for new connections.

### **EXAMPLE- 1**

```
/*webserver.c*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int
main (int argc, char **argv)
{
    int connfd, lfd, listenfd, i = 0;
    pid_t pid, ret;
    int p[2];
    FILE *fp;
    socklen_t clilen;
    char buff[20000], *buf, ch;
    struct sockaddr_in cliaddr, servaddr;
    if ((lfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror ("sockfd error");
    }

    buf =
        "HTTP/1.1 200 OK\nDate: Mon, 19 Oct 2009 01:26:17 GMT\nServer: Apache/1.2.6 Red
        Hat\nContent-Length: 18\nAccept-Ranges: bytes\nKeep-Alive: timeout=15,
        max=100\nConnection: Keep-Alive\nContent-Type: text/html\n\n<html>Hello</html>";

    bzero (&servaddr, sizeof (servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
    servaddr.sin_port = htons (atoi (argv[1]));
    if (bind (lfd, (struct sockaddr *) &servaddr, sizeof (servaddr)))
    {
        perror ("Bind error");
    }
    listen (lfd, 10);
    for (;;)
    {
        clilen = sizeof (cliaddr);
        if ((connfd = accept (lfd, (struct sockaddr *) &cliaddr, &clilen)) < 0)
        {
```

```

        perror ("connection error");
    }
    if ((pid = fork ()) < 0)
    {
        perror ("fork:");
    }
    if (pid == 0)
    {
        close (listenfd);
        recv (connfd, buff, sizeof (buff), 0);
        printf ("%s\n", buff);
        send (connfd, buf, strlen (buf), 0);
        close (connfd);
        exit (0);
    }

    close (connfd);
}
}

```

## Q?

1. Compile the above program and run it with a port number. Connect to this webserver using a browser. In the browser give the url as <http://localhost:portnumber>. Observe the output.
2. Connect to the web server multiple times using multiple instances of browser. Check if there are any zombie processes created. Modify the above code to handle all zombie processes.
3. Does the above server withstand the signal interruptions? Modify the code to handle signal interruptions?
4. Modify the above server to handle dynamic HTTP requests which require help of an external executable. Read more at this point: <http://web.archive.org/web/20100127161358/http://hoohoo.ncsa.illinois.edu/cgi/>

## UDP Echo server



The User Datagram Protocol (UDP) provides a simpler end-to-end service than TCP provides. In fact, UDP performs only two functions: (1) It adds another layer of addressing (ports) to that of IP; and (2) it detects data corruption that may occur in transit and discards any corrupted datagrams.

Because of this simplicity, UDP (datagram) sockets have some different characteristics from the TCP (stream) sockets. For example, UDP sockets do not have to be connected before being used. Where TCP is analogous to telephone communication, UDP is analogous to communicating by

mail: You do not have to “connect” before you send a package or letter, but you do have to specify the destination address for each one. In receiving, a UDP socket is like a mailbox into which letters or packages from many different sources can be placed. Another difference between UDP sockets and TCP sockets is the way they deal with message boundaries: UDP sockets preserve them. This makes receiving an application message simpler, in some ways, than with TCP sockets. A final difference is that the end-to-end transport service UDP provides is best effort: There is no guarantee that a message sent via a UDP socket will arrive at its destination. This means that a program using UDP sockets must be prepared to deal with loss and reordering of messages.

We will see a simple echo client and echo server using UDP.  
The echo client and server programs are present as UDPEchoClient.c and UDPEchoServer.c.

## Q?

1. \$make it will compile client and server.
2. Modify UDPEchoClient.c to use connect(). After the final recv(), show how to disconnect the UDP socket. Using getsockname() and getpeername(), print the local and foreign address before and after connect(), and after disconnect.
3. Modify UDPEchoServer.c to use connect(). 
4. Verify experimentally the size of the largest datagram you can send and receive using a UDP socket. This is done as follows:
  - a. First send msg of size x bytes. If it successful, double the size and resend the msg. keep doing it until the message is delivered successfully.
  - b. At some point t, if the msg of size  $y_t$  is lost, then experiment with  $(y_t + y_{t-1})/2$ . This way we can arrive at a size which is the maximum UDP segment size. 
5. Modify UDPEchoClient.c to print the local and foreign address for the socket immediately before and after sendto().
6. You can use the same UDP socket to send datagrams to many different destinations. Modify UDPEchoClient.c to send and receive an echo datagram to/from two different UDPEcho servers. You can use the server running on multiple hosts or twice on the same host with different ports.

---

## Unix Domain Sockets

The following are the two programs which act as stream echo client and server.

```
//unixdomainserver.c  
  
#include <signal.h>  
#include <stdio.h>
```

```

#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <unistd.h>
#include <strings.h>
#include <sys/un.h>

void str_echo(int sockfd)
{
    ssize_t n;
    char buf[1000];
    while((n=read(sockfd,buf,1000))>0)
    {
        write(sockfd,buf,n);
        printf("%s\n",buf);
    }
}

int
main(int argc, char **argv)
{
    int                listenfd, connfd;
    pid_t              childpid;
    socklen_t          clilen;
    struct sockaddr_un cliaddr, servaddr;
    void                sig_chld(int h)
    {
        return;
    }

    listenfd = socket(AF_LOCAL, SOCK_STREAM, 0);

    unlink(argv[1]);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, argv[1]);

    bind(listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

    listen(listenfd, 10);

    signal(SIGCHLD, sig_chld);

    for ( ; ; ) {
        clilen = sizeof(cliaddr);
        if ( (connfd = accept(listenfd, (struct sockaddr *) &cliaddr, &clilen)) <
0) {
            printf("connection error\n");
            continue;          /* back to for() */
        }

        if ( (childpid = fork()) == 0) { /* child process */
            close(listenfd);      /* close listening socket */
            str_echo(connfd);     /* process the request */
            exit(0);
        }
        close(connfd);           /* parent closes connected socket */
    }
}

//Unixdomainclient.c

```

```

#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <unistd.h>
#include <strings.h>
#include <string.h>
#include <sys/un.h>
#define MAXLINE 1000

void
str_cli(FILE *fp, intsockfd)
{
    char    sendline[MAXLINE], recvline[MAXLINE];

    while (fgets(sendline, MAXLINE, fp) != NULL) {

        write(sockfd, sendline, strlen(sendline));

        if (read(sockfd, recvline, MAXLINE) == 0)
            printf("str_cli: server terminated prematurely\n");

        fputs(recvline, stdout);

    }
}

int
main(intargc, char **argv)
{
    int                sockfd;
    structsockaddr_un  servaddr;

    sockfd = socket(AF_LOCAL, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, argv[1]);

    connect(sockfd, (structsockaddr *) &servaddr, sizeof(servaddr));

    str_cli(stdin, sockfd);          /* do it all */

    exit(0);
}

```

## Q?

1. Observe the unix domain sockets created in your system using the netstat command. Can you find the addresses where servers are running? (use netstat -na command)
2. Run the above server and client and see the netstat. Do you find the server?
3. Modify the above server and client to be unix datagram client and server.

**The following is the program which is about passing a file descriptor from child to parent**

## using unix domain protocols.

```
/* passfd.c -- sample program which passes a file descriptor */

/* We create Unix domain sockets through
socketpair(), and then fork(). The child opens the file whose
name is passed on the command line, passes the file descriptor
and file name back to the parent, and then exits. The parent waits
for the file descriptor from the child, then copies data from that
file descriptor to stdout until no data is left. The parent then
exits. */

#include <alloca.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/uio.h>
#include <sys/un.h>
#include <sys/wait.h>
#include <unistd.h>

#include "sockutil.h"          /* simple utility functions */

/* The child process. This sends the file descriptor. */
intchildProcess(char * filename, int sock) {
    intfd;
    structiovec vector;        /* some data to pass w/ the fd */
    structmsgghdrmsg;          /* the complete message */
    structcmsghdr * cmsg;      /* the control message, which will */
                                /* include the fd */

    /* Open the file whose descriptor will be passed. */
    if ((fd = open(filename, O_RDONLY)) < 0) {
        perror("open");
        return 1;
    }

    /* Send the file name down the socket, including the trailing
    '\0' */
    vector.iov_base = filename;
    vector.iov_len = strlen(filename) + 1;

    /* Put together the first part of the message. Include the
    file name iovec */
    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = &vector;
    msg.msg_iovlen = 1;

    /* Now for the control message. We have to allocate room for
    the file descriptor. */
    cmsg = alloca(sizeof(structcmsghdr) + sizeof(fd));
    cmsg->cmsg_len = sizeof(structcmsghdr) + sizeof(fd);
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SCM_RIGHTS;

    /* copy the file descriptor onto the end of the control
    message */
    memcpy(CMSG_DATA(cmsg), &fd, sizeof(fd));

    msg.msg_control = cmsg;
```

```

msg.msg_controllen = cmsg->cmsg_len;

if (sendmsg(sock, &msg, 0) != vector.iov_len)
die("sendmsg");

return 0;
}

/* The parent process. This receives the file descriptor. */
int parentProcess(int sock) {
    char buf[80]; /* space to read file name into */
    struct iovec vector; /* file name from the child */
    struct msghdr msg; /* full message */
    struct cmsghdr * cmsg; /* control message with the fd */
    int fd;

    /* set up the iovec for the file name */
    vector.iov_base = buf;
    vector.iov_len = 80;

    /* the message we're expecting to receive */

    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = &vector;
    msg.msg_iovlen = 1;

    /* dynamically allocate so we can leave room for the file
    descriptor */
    cmsg = alloca(sizeof(struct cmsghdr) + sizeof(fd));
    cmsg->cmsg_len = sizeof(struct cmsghdr) + sizeof(fd);
    msg.msg_control = cmsg;
    msg.msg_controllen = cmsg->cmsg_len;

    if (!recvmsg(sock, &msg, 0))
    return 1;

    printf("got file descriptor for '%s'\n",
           (char *) vector.iov_base);

    /* grab the file descriptor from the control structure */
    memcpy(&fd, CMSG_DATA(cmsg), sizeof(fd));

    copyData(fd, 1);

    return 0;
}

int main(int argc, char ** argv) {
    int socks[2];
    int status;

    if (argc != 2) {
        fprintf(stderr, "only a single argument is supported\n");
        return 1;
    }

    /* Create the sockets. The first is for the parent and the
    second is for the child (though we could reverse that
    if we liked. */
    if (socketpair(PF_UNIX, SOCK_STREAM, 0, socks))
    die("socketpair");

```

```

if (!fork()) {
    /* child */
    close(socks[0]);
    return childProcess(argv[1], socks[1]);
}

/* parent */
close(socks[1]);
parentProcess(socks[0]);

/* reap the child */
wait(&status);

if (WEXITSTATUS(status))
    fprintf(stderr, "child failed\n");

return 0;
}

```

## Q?

1. compile the code: `gcc passfd.c sockutil.c -o passfd`
2. Run the file with argument as a file name. The child opens the file and passes on the descriptor to the parent.
3. Extend the above program to work with unrelated processes.

## DNS & Domain Name Lookups

The following program demonstrates the usage of `gethostbyname()` library function.

```

/*gethostbyname.c*/
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <string.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <strings.h>
#include <sys/socket.h>

int main()
{
    char *ptr, **pptr, url[50], address[100];
    char str[INET_ADDRSTRLEN];
    struct hostent *hptr;
    printf("Enter Hostname:");
    while((gets(url)), !feof(stdin))
    {
        ptr=url;
        if((hptr=gethostbyname(ptr))==NULL)

```




```

        continue;
    }
    strcpy(url, hptr->h_name);
    printf("official hostname:%s\n", hptr->h_name);
    for(pptr=hptr->h_aliases; *pptr!=NULL; pptr++)
        printf("alias:%s\n", *pptr);
    switch(hptr->h_addrtype)
    {
        case AF_INET:
            pptr=hptr->h_addr_list;
            for(; *pptr!=NULL; pptr++) {
                strcpy(address, inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
                printf("address:%s\n", inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
            }
            break;
        default:
            break;
    }
    printf("\n Enter url:");
}
exit(0);
}

```

# Q?

1. Run the above program and see the output for [www.google.com](http://www.google.com) or [www.yahoo.com](http://www.yahoo.com). What did you understand? Run few times for the same domain say [www.google.com](http://www.google.com) and note the order of the IPs listed. Can you guess why the order gets changed?
2. Instead of using the `gethostbyname()`, use `getaddrinfo()`. Sample code is given below 

```

struct addrinfo  hints, *res, *ressave;
bzero(&hints, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
if ( (n = getaddrinfo(host, serv, &hints, &res)) != 0)
{
    printf("tcp_connect error for %s, %s: %s",
        host, serv, gai_strerror(n));
    exit(0);
}
ressave = res;
do {
    sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
    if (sockfd < 0)
        continue;    /* ignore this one */

    if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
        break;    /* success */
    close(sockfd);    /* ignore this one */
} while ( (res = res->ai_next) != NULL);

```

```
if (res == NULL) /* errno set from final connect() */
    printf("tcp_connect error for %s, %s", host, serv);
freeaddrinfo(ressave);
```

### Working with DNS Protocol (Information):

Instead of using the library functions, one can use a direct UDP socket to connect to DNS server provided the message should be as per RFC 1035. Please refer to [DN protocol.pdf](#).

There are two files given in 'DNS client' directory. Look at the [msgformat.h](#) file. This file defines the structures corresponding to DNS message format. DNS msg contains header, question, answer, authority and additional section.

DNS client program contains filling in the structure and sending it to the server. It receives the reply and displays the contents.

## Q?

1. compile the program and run it. Run it for google.com, yahoo.com, amazon.com, gmail.com etc. observe the ip address orders.
2. make query for MX record (you need to change the QTYPE field)
3. make the query for CNAME record
4. make an iterative query
5. make a query for AAAA record

=====