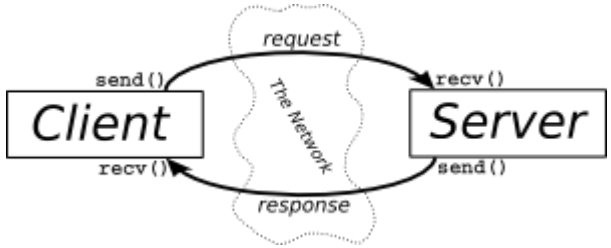
Prev Contents Next

5. Client-Server Background

It's a client-server world, baby. Just about everything on the network deals with client processes talking to server processes and vice-versa. Take **telnet**, for instance. When you connect to a remote host on port 23 with telnet (the client), a program on that host (called **telnetd**, the server) springs to life. It handles the incoming telnet connection, sets you up with a login prompt, etc.



Client-Server Interaction.

The exchange of information between client and server is summarized in Figure 2.

Note that the client-server pair can speak **SOCK_STREAM**, **SOCK_DGRAM**, or anything else (as long as they're speaking the same thing.) Some good examples of client-server pairs are **telnet**/**telnetd**, **ftp**/**ftpd**, or **bootp**/**bootpd**. Every time you use **ftp**, there's a remote program, **ftpd**, that serves you.

Often, there will only be one server on a machine, and that server will handle multiple clients using `fork()`. The basic routine is: server will wait for a connection, `accept()` it, and `fork()` a child process to handle it. This is what our sample server does in the next section.

5.1. A Simple Stream Server

All this server does is send the string "`Hello, World!\n`" out over a stream connection. All you need to do to test this server is run it in one window, and telnet to it from another with:

```
$ telnet remotehostname 3490
```

where `remotehostname` is the name of the machine you're running it on.

The server code: (Note: a trailing backslash on a line means that the line is continued on the next.)

```
/*
** server.c -- a stream socket server demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/wait.h>
#include <signal.h>

#define MYPORT 3490    // the port users will be connecting to

#define BACKLOG 10     // how many pending connections queue will hold

void sigchld_handler(int s)
{
    while(waitpid(-1, NULL, WNOHANG) > 0);
}

int main(void)
{
    int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
    struct sockaddr_in my_addr;    // my address information
    struct sockaddr_in their_addr; // connector's address information
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    if (setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1) {
        perror("setsockopt");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
```

```
    my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))
                                                                == -1) {
        perror("bind");
        exit(1);
    }

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    sa.sa_handler = sigchld_handler; // reap all dead processes
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        perror("sigaction");
        exit(1);
    }

    while(1) {  // main accept() loop
        sin_size = sizeof(struct sockaddr_in);
        if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                                                    &sin_size)) == -1) {
            perror("accept");
            continue;
        }
        printf("server: got connection from %s\n",
                                        inet_ntoa(their_addr.sin_addr));
        if (!fork()) { // this is the child process
            close(sockfd); // child doesn't need the listener
            if (send(new_fd, "Hello, world!\n", 14, 0) == -1)
                perror("send");
            close(new_fd);
            exit(0);
        }
        close(new_fd);  // parent doesn't need this
    }

    return 0;
}
```

In case you're curious, I have the code in one big `main()` function for (I feel) syntactic clarity. Feel free to split it into smaller functions if it makes you feel better.

(Also, this whole `sigaction()` thing might be new to you--that's ok. The code that's there is responsible for reaping zombie processes that appear as the `fork()`ed child processes exit. If you make lots of zombies and don't reap them, your system administrator will become agitated.)

You can get the data from this server by using the client listed in the next section.

5.2. A Simple Stream Client

This guy's even easier than the server. All this client does is connect to the host you specify on the command line, port 3490. It gets the string that the server sends.

The client source:

```
/*
** client.c -- a stream socket client demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define PORT 3490 // the port client will be connecting to

#define MAXDATASIZE 100 // max number of bytes we can get at once

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct hostent *he;
    struct sockaddr_in their_addr; // connector's address information

    if (argc != 2) {
        fprintf(stderr,"usage: client hostname\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
        herror("gethostbyname");
```

```
        exit(1);
    }

    if ((sockfd = socket(PF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;    // host byte order
    their_addr.sin_port = htons(PORT);  // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), '\0', 8);  // zero the rest of the struct

    if (connect(sockfd, (struct sockaddr *)&their_addr,
                                        sizeof(struct sockaddr)) == -1) {
        perror("connect");
        exit(1);
    }

    if ((numbytes=recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
        perror("recv");
        exit(1);
    }

    buf[numbytes] = '\0';

    printf("Received: %s",buf);

    close(sockfd);

    return 0;
}
```

Notice that if you don't run the server before you run the client, `connect()` returns "Connection refused". Very useful.

5.3. Datagram Sockets

I really don't have that much to talk about here, so I'll just present a couple of sample programs: *`talker.c`* and *`listener.c`*.

`listener` sits on a machine waiting for an incoming packet on port 4950. **`talker`** sends a packet to that port, on the specified machine, that contains whatever the user enters on the command line.

Here is the source for *`listener.c`*:

```
/*
** listener.c -- a datagram sockets "server" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define MYPORT 4950    // the port users will be connecting to

#define MAXBUFLEN 100

int main(void)
{
    int sockfd;
    struct sockaddr_in my_addr;    // my address information
    struct sockaddr_in their_addr; // connector's address information
    socklen_t addr_len;
    int numbytes;
    char buf[MAXBUFLEN];

    if ((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;         // host byte order
    my_addr.sin_port = htons(MYPORT);     // short, network byte order
    my_addr.sin_addr.s_addr = INADDR_ANY; // automatically fill with my IP
    memset(&(my_addr.sin_zero), '\0', 8); // zero the rest of the struct

    if (bind(sockfd, (struct sockaddr *)&my_addr,
        sizeof(struct sockaddr)) == -1) {
        perror("bind");
        exit(1);
    }

    addr_len = sizeof(struct sockaddr);
    if ((numbytes=recvfrom(sockfd, buf, MAXBUFLEN-1 , 0,
        (struct sockaddr *)&their_addr, &addr_len)) == -1) {
        perror("recvfrom");
        exit(1);
```

```
    }

    printf("got packet from %s\n",inet_ntoa(their_addr.sin_addr));
    printf("packet is %d bytes long\n",numbytes);
    buf[numbytes] = '\0';
    printf("packet contains \"%s\"\n",buf);

    close(sockfd);

    return 0;
}
```

Notice that in our call to `socket()` we're finally using **`SOCK_DGRAM`**. Also, note that there's no need to `listen()` or `accept()`. This is one of the perks of using unconnected datagram sockets!

Next comes the source for *`talker.c`*:

```
/*
** talker.c -- a datagram "client" demo
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950    // the port users will be connecting to

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // connector's address information
    struct hostent *he;
    int numbytes;

    if (argc != 3) {
        fprintf(stderr,"usage: talker hostname message\n");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
```

```
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    their_addr.sin_family = AF_INET;     // host byte order
    their_addr.sin_port = htons(SERVERPORT); // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(&(their_addr.sin_zero), '\0', 8);  // zero the rest of the struct

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
             (struct sockaddr *)&their_addr, sizeof(struct sockaddr))) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n", numbytes, inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

And that's all there is to it! Run **`listener`** on some machine, then run **`talker`** on another. Watch them communicate! Fun G-rated excitement for the entire nuclear family!

Except for one more tiny detail that I've mentioned many times in the past: connected datagram sockets. I need to talk about this here, since we're in the datagram section of the document. Let's say that **`talker`** calls `connect()` and specifies the **`listener`**'s address. From that point on, **`talker`** may only sent to and receive from the address specified by `connect()`. For this reason, you don't have to use `sendto()` and `recvfrom()`; you can simply use `send()` and `recv()`.
