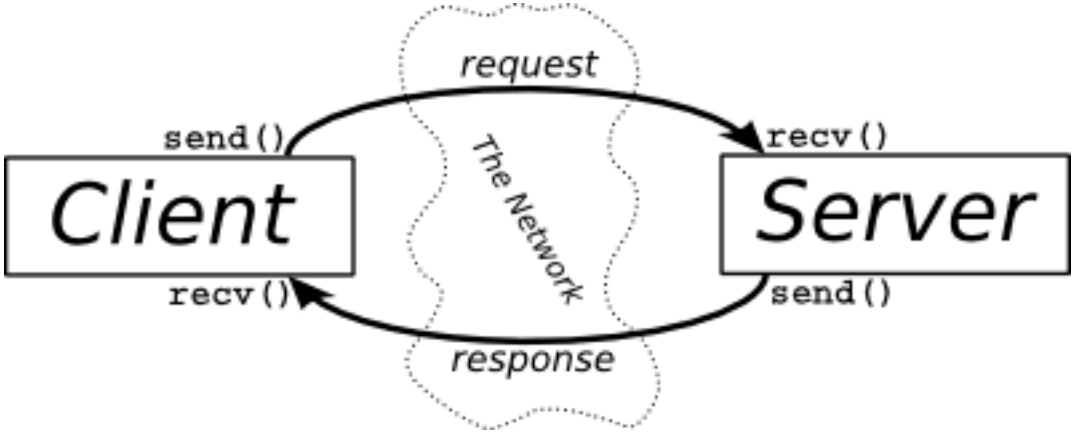
6. Client-Server Background

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 the above diagram.

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 **Firefox/Apache**. 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.

6.1. A Simple Stream Server

All this server does is send the string "`Hello, world!`" 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:

```
/*
** 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 <netdb.h>
#include <arpa/inet.h>
```

```
#include <sys/wait.h>
#include <signal.h>

#define PORT "3490"  // the port users will be connecting to

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

void sigchld_handler(int s)
{
    // waitpid() might overwrite errno, so we save and restore it:
    int saved_errno = errno;

    while(waitpid(-1, NULL, WNOHANG) > 0);

    errno = saved_errno;
}


// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(void)
{
    int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr; // connector's address information
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char s[INET6_ADDRSTRLEN];
    int rv;

    memset(&hints, 0, sizeof hints);
```

```
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("server: socket");
            continue;
        }

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

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("server: bind");
            continue;
        }

        break;
    }

    freeaddrinfo(servinfo); // all done with this structure

    if (p == NULL)  {
        fprintf(stderr, "server: failed to bind\n");
        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);
    }

    printf("server: waiting for connections...\n");

    while(1) {  // main accept() loop
        sin_size = sizeof their_addr;
        new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
        if (new_fd == -1) {
            perror("accept");
            continue;
        }

        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s);
        printf("server: got connection from %s\n", s);

        if (!fork()) { // this is the child process
            close(sockfd); // child doesn't need the listener
            if (send(new_fd, "Hello, world!", 13, 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.

6.2. 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>

#include <arpa/inet.h>

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

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

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

int main(int argc, char *argv[])
{
    int sockfd, numbytes;
    char buf[MAXDATASIZE];
    struct addrinfo hints, *servinfo, *p;
    int rv;
    char s[INET6_ADDRSTRLEN];

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

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and connect to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("client: socket");
            continue;
```

```
        }

        if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("client: connect");
            continue;
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "client: failed to connect\n");
        return 2;
    }

    inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr),
            s, sizeof s);
    printf("client: connecting to %s\n", s);

    freeaddrinfo(servinfo); // all done with this structure

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

    buf[numbytes] = '\0';

    printf("client: received '%s'\n",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.

6.3. Datagram Sockets

We've already covered the basics of UDP datagram sockets with our discussion of **`sendto()`** and **`recvfrom()`**, above, 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>
#include <netdb.h>

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

#define MAXBUFLEN 100

// get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
```

```
}

int main(void)
{
    int sockfd;
    struct addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;
    struct sockaddr_storage their_addr;
    char buf[MAXBUFLEN];
    socklen_t addr_len;
    char s[INET6_ADDRSTRLEN];

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC; // set to AF_INET to force IPv4
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, MYPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and bind to the first we can
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("listener: socket");
            continue;
        }

        if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
            close(sockfd);
            perror("listener: bind");
            continue;
        }

        break;
    }
```

```
    if (p == NULL) {
        fprintf(stderr, "listener: failed to bind socket\n");
        return 2;
    }

    freeaddrinfo(servinfo);

    printf("listener: waiting to recvfrom...\n");

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

    printf("listener: got packet from %s\n",
        inet_ntop(their_addr.ss_family,
            get_in_addr((struct sockaddr *)&their_addr),
            s, sizeof s));
    printf("listener: packet is %d bytes long\n", numbytes);
    buf[numbytes] = '\0';
    printf("listener: packet contains \"%s\"\n", buf);

    close(sockfd);

    return 0;
}
```

Notice that in our call to **`getaddrinfo()`** 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 addrinfo hints, *servinfo, *p;
    int rv;
    int numbytes;

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

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_DGRAM;

    if ((rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }

    // loop through all the results and make a socket
    for(p = servinfo; p != NULL; p = p->ai_next) {
        if ((sockfd = socket(p->ai_family, p->ai_socktype,
                p->ai_protocol)) == -1) {
            perror("talker: socket");
            continue;
```

```
        }

        break;
    }

    if (p == NULL) {
        fprintf(stderr, "talker: failed to create socket\n");
        return 2;
    }

    if ((numbytes = sendto(sockfd, argv[2], strlen(argv[2]), 0,
             p->ai_addr, p->ai_addrlen)) == -1) {
        perror("talker: sendto");
        exit(1);
    }

    freeaddrinfo(servinfo);

    printf("talker: sent %d bytes to %s\n", numbytes, argv[1]);
    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!

You don't even have to run the server this time! You can run **talker** by itself, and it just happily fires packets off into the ether where they disappear if no one is ready with a **`recvfrom()`** on the other side. Remember: data sent using UDP datagram sockets isn't guaranteed to arrive!

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()`**.