Client-Server Programming with TCP/IP Sockets

Keith Gaughan

March 22, 2003

Contents

1 Background

Sockets are one of the most important IPC mechanisms on UNIX. Originally introduced in 4.2BSD as a generalisation of pipes, they later became the basis for the UNIX networking subsystem. Sockets are the only IPC mechanism that allows communication between processes running on different machines. Essentially, it is an end-point of communication which may be bound to a name.

But enough with the bland introduction. A socket is just a way to allow processes to talk to one another. It doesn't matter if they're running on the same machine, just as long as one knows the other's contact details. Sockets are a lot like using a phone—to contact somebody else, you have to have a phone and the other person's phone number. They also have to have one installed and must be listening for any incoming calls[1].

There are a variety of different socket address families, the two most common being *UNIX*[2], denoted by the constant `AF_UNIX`, and *Internet*. Internet addressing, denoted by the constant `AF_INET`, is the address family used to allow communication between two different machines and is the one covered in this article.

1.1 Files & streams in UNIX

One of the most important abstractions in UNIX is that *nearly everything is a file*, or more precisely a continuous stream of bytes. It doesn't matter whether you're dealing with reading from the keyboard or mouse, talking to the network card, writing to the screen, communicating with another process with pipes, or anything—they all work the same way. This is rather a powerful abstraction, and it works pretty well.

1.2 TCP and UDP

As you should remember from networking, on top of IP, there are two major transport protocols on top of which all other protocols are built: TCP[3] and UDP[4]. These act as transportation mechanisms for other, higher-level, protocols.

TCP is a reliable, connection-oriented protocol that transmits data as a stream of bytes. UDP, on the other hand, is an unreliable, connectionless protocol that sends data in chunks called *datagrams*. You can't depend on anything you send using UDP to get through, and sometimes the datagrams you send can end up being recieved more than once on the other side, or in a different order from the one they were sent in.

Both have different objectives. For instance, TCP is used as the basis for FTP[5] because sending information like your username, password, current directory, and so on, between the client and the server would be a waste of time[6], and when you're transferring a file you don't want part of it to go missing on you all of a sudden! UDP is used for other protocols that don't require state, or where the overhead of a TCP connection isn't worth it because it's ok for some of the data to get lost[7] and the communications lag associated with the connection would be a liability.

UDP sockets are denoted by the constant `SOCK_DGRAM`, and aren't covered in these notes.

1.3 Connecting it together with sockets

Files in UNIX and TCP, both being based around streams of bytes, are an almost perfect fit. Sockets allow you to treat TCP connections as if they were files.

Now all you need is some way for the client and server to understand one another, and for that you need a *Protocol*.

[1] Although, unlike phones, sockets don't play *The Macarena*.
[2] Which maps sockets onto the filing system. This is used for IPC between processes on the same machine.
[3] Transmission Control Protocol
[4] User Datagram Protocol
[5] File Transfer Protocol
[6] These kinds of things are known as *state*, and TCP is well suited to protocols that require state.
[7] Multiplayer network games such as *Half Life*, *Age of Empires*, and *Unreal Tournament* use UDP-based communication protocols because of this.

2 Protocols

A protocol is a common language that the server and the client both understand.

2.1 What's the need?

TCP is fine for transporting data across a network, after all that's what it's for. However, TCP has no idea what data it's transporting about.

To best explain why you need a protocol, imagine if you called somebody in South Africa who didn't understand a word of Finnish and started talking down the line at them. Most likely, they'd think there's a lunatic on the other side and would just hang up. The same thing happens in network communications—without a protocol, the machines involved wouldn't know what each other was blathering on about and would probably get confused.

2.2 A simple protocol: String Length

Later on, we'll be building a client and a server to demonstrate how to use sockets. For that, we need a simple protocol that both will understand. Our ridiculously simple example will be a protocol for finding the length of a null-terminated string. Let's formalise the steps required.

1. The client connects to the server.
2. The server accepts the connection.
3. The client sends a null-terminated string to the server and waits for the server to reply.
4. The server reads the string sent, gets its length, sends the result as a null-terminated string back to the client, and closes the connection on its side.
5. The client reads the length sent back and closes the connection on its side.

3 Implementing clients and servers

There are a number of discrete steps you must follow to build a client or a server.

3.1 Libraries required

To use sockets, you need to include the following header files:

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <syslog.h>
#include <sys/types.h>
#include <sys/sockets.h>
#include <netinet/in.h>
#include <netdb.h>
```

3.2 Creating a socket

Both the client and server need to create a socket before they can do anything. The client uses its socket to connect to a server whilst the server uses its socket for listening for new connections. Socket creation is done using the `socket()` call.

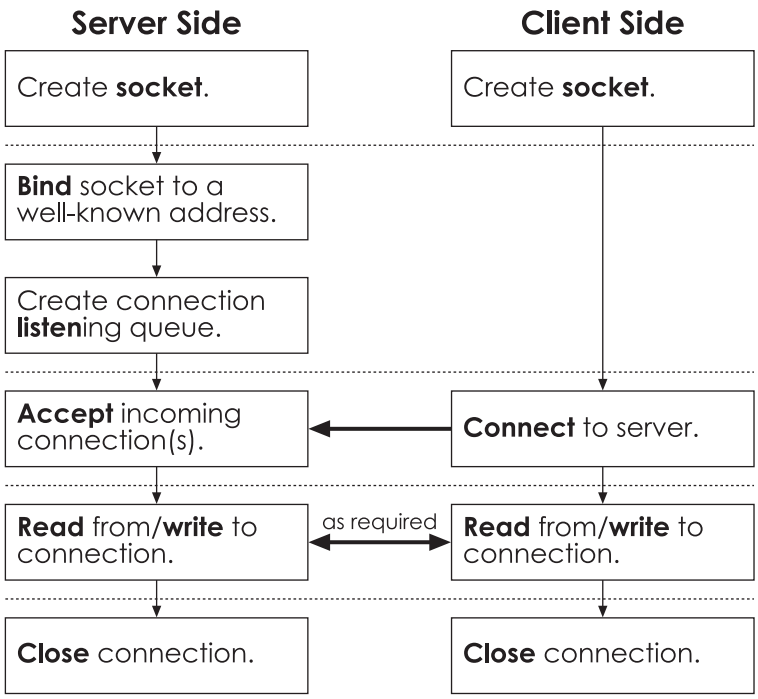

Figure 1: Steps in setting up a client and server

3.2.1 socket()

Creates a new socket. Returns a file descriptor representing the socket end-point, or `-1`, if an error occurs.

```
int socket(int af, int type, int protocol);
```

af is the address family to use with the socket. This could be either `AF_UNIX` for the UNIX address family (for local IPC), or `AF_INET` for the Internet address family (for network communication). TCP requires that you use `AF_INET`.

type is the socket type. This can be `SOCK_STREAM` for connection-oriented, stream-based protocols like TCP, `SOCK_DGRAM` for datagram-based, connectionless protocols like UDP, or `SOCK_RAW` where you want to use your own transportation protocol[8].

protocol is the transport protocol to use. It's best to pass `0`, which lets the system decide.

3.3 Binding the socket to a well-known address

For clients to connect to a server, they need to know its address, but sockets are created without an address of their own and so must be assigned to one that the client will know about. This is done by *binding* a socket to a *well-known address* on the system. For instance, web-servers are usually bound to port 80 as this is the well-known port for the HTTP protocol.

3.3.1 struct sockaddr

This is a generic socket address structure provided to allow flexibility in passing socket addresses to functions.

[8] . . . and there be dragons!

```
struct sockaddr
{
    unsigned short sa_family;   // Address family tag.
    char           sa_data[14]; // Padding.
};
```

3.3.2 struct sockaddr_in

`struct sockaddr_in` is a specialised version of `struct sockaddr` especially for the `AF_INET` address family.

```
struct sockaddr_in
{
    unsigned short sin_family;  // Set to AF_INET.
    unsigned short sin_port;    // Port number to bind to.
    struct in_addr sin_addr;    // IP address.
    char           sin_zero[8]; // Padding.
};
```

3.3.3 struct in_addr

`struct in_addr` represents an IP address. Why this structure exists and wasn't just incorporated directly into `struct sockaddr_in` is anybody's guess.

Setting its one field, `s_addr`, to `INADDR_ANY` will leave it up to the server to choose an appropriate host IP address, and this is usually the best thing to do.

```
struct in_addr
{
    unsigned long s_addr;   // IP address.
};
```

3.3.4 bind()

Binds a socket to a well-known address. This will return `0` if successful and `-1` if not.

```
int bind(int fd, struct sockaddr* addr, int len);
```

fd is the file descriptor of the socket to bind.

addr points to the address to bind the socket to.

len is the length of the address in bytes.

3.3.5 Example: BindSocket()

Here's a wrapper function that demonstrates how to use `bind()` to bind a socket to an address.

```
/**
 * Binds a socket to a given port.
 *
 * @param  fd    File descriptor of socket to bind.
 * @param  port  Port number to bind to.
 *
 * @return 0 if successful, -1 for failure.
 */
int BindSocket(int fd, unsigned short port)
```

```
{
    struct sockaddr_in addr;

    addr.sin_family      = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;  // Let the host decide.
    addr.sin_port        = htons(port); // Port to bind to.

    return bind(fd, (struct sockaddr*) &addr, sizeof(addr));
}
```

3.4 Creating a connection listening queue

After a socket is bound to an address, it's still not listening for any incoming connections. In addition, to ensure no incoming connections are lost, a *connection queue* needs to be set up. This allows incoming connections to queue up while the server is dealing with an earlier one.

3.4.1 listen()

Makes the socket listen for incoming connections[9] and sets up a connection queue for the socket.

```
int listen(int fd, int len);
```

fd is the file descriptor of the socket to put in passive mode.

len is the length of the connection queue. `5` is the usual value used.

3.5 Connecting to a server

After creating a socket, the client needs establish a connection with a server. Once this is done, the socket is placed in *active mode*.

3.5.1 struct hostent

Holds a host's details.

```
struct hostent
{
    char*  h_name;      // Host's real name.
    char** h_aliases;   // Alias list.
    int    h_addrtype;  // Address type, i.e. AF_INET.
    int    h_length;    // Address length.
    char** h_addr_list; // List of host's addresses.
};

// Used for getting first address from address list.
#define h_addr h_addr_list[0]
```

3.5.2 gethostbyname()

Allows you to discover a host's details (including its address) by specifying its name. This returns a structure specifying the host's details, or `NULL` if it fails.

```
struct hostent* gethostbyname(char* name);
```

To create an address suitable for passing into `connect()`, the following must be done to set it up:

[9] This is called *passive mode*.

```
struct hostent*    info;
struct sockaddr_in addr;

// Get host's details.
info = gethostbyname(hostname);

// Copy the address family.
addr.sin_family = info->h_addrtype;

// Set up the correct port.
addr.sin_port = htons(SERVER_PORT);

// Copy over the host's address.
memcpy((void*) &addr.sin_addr, info->h_addr, info->h_length);
```

3.5.3 connect()

Connects a socket to a given server, putting the socket in *active mode*. Returns `0` if successful, else `-1`.

```
int connect(int fd, struct sockaddr* addr, int len);
```

fd is the file descriptor of the socket to connect.

addr points to the address of the server to connect to.

len is the length in bytes of the address.

3.5.4 Example: ConnectToServer()

This wrapper function demonstrates all the steps a client must take to connect to a server.

```
/**
 * Creates a socket and connects to the specified server.
 *
 * @param  hostname  Host server is running on.
 * @param  port      Port server is bound to.
 *
 * @return File descriptor for connected socket, or -1 if failed.
 */
int ConnectToServer(char* hostname, unsigned short port)
{
    int fd;

    struct hostent*    info;
    struct sockaddr_in addr;

    // Get the server's details.
    info = gethostbyname(hostname);
    if (info == NULL)
    {
        perror("Host not found.");
        return -1;
    }

    // Get the server's address.
    addr.sin_family = info->h_addrtype;
```

```
    addr.sin_port   = htons(port);
    memcpy((void*) &addr.sin_addr, info->h_addr, info->h_length);

    // Create a socket.
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd == -1)
    {
        perror("Could not open socket.");
        return -1;
    }

    // Try to establish connection to server.
    if (connect(fd, (struct sockaddr*) &addr, sizeof(addr)) == -1)
    {
        perror("Could not connect to server.");
        close(fd);
        return -1;
    }

    return fd;
}
```

3.6 Accepting incoming connections

After the connection queue has been created, the server can accept incoming connections from clients wishing to talk to it. To accept one, it must call `accept()`.

3.6.1 accept()

Accepts a single incoming connection. Returns a file descriptor corresponding to the new connection, or `-1` if an error occurs.

```
int accept(int fd, struct sockaddr* addr, int* len);
```

fd is the file descriptor of the socket listening for incoming connections.

addr is an address structure to hold the address of the client making the connection. Pass `NULL` here if you don't care about getting this information.

len is pointer to a variable holding the length in bytes of the address structure passed in `addr`. On returning, this will hold the actual length of the client address. If you passed in `NULL` to `addr`, pass `NULL` in here too.

3.6.2 Notes on accepting connections

Calling `accept()` is only good for fetching one pending connection off the queue and so must be called in a loop to continually loop, each iteration fetching a new connection from the queue. For instance, this function accepts any incoming connections and shoots them off to a handler called `HandleConnection()`.

```
/**
 * Waits on and dispatches any incoming client connections to a
 * handler.
 *
 * @param  fdQueue  File descriptor of socket to wait on.
 */
void DispatchIncoming(int fdQueue)
```

```
{
    int fd;

    for (;;)
    {
        // Fetch a new connection off the queue.
        fd = accept(fdQueue, NULL, NULL);

        if (fd == -1)
        {
            // Oh dear! Something's gone wrong! Whimper and die.
            perror("Could not accept incoming connection.");
            exit(EXIT_FAILURE);
        }

        // Got a connection! Handle it.
        HandleConnection(fd);
    }
}
```

3.7 Reading and Writing data

After the client has connected to the server and the server has accepted the connection, they can start sending data back and forth. This is done with good old `read()` and `write()`. There are, however, some caveats attached to reading data, however…

3.7.1 write()

Writes data to a file descriptor.

```
int write(int fd, void* buf, unsigned int n);
```

fd is the file descriptor to write to.

buf is a buffer containing data to write.

n is the number of bytes from the buffer to write.

3.7.2 read()

Reads data from a file descriptor. It returns the number of bytes actually read, or `-1` if an error occurs.

```
int read(int fd, void* buf, unsigned int n);
```

fd is the file descriptor to read from.

buf is a buffer to write the data read to.

n is the size in bytes of the buffer.

3.7.3 Caveats when using read() with sockets

Your program can munch through incoming data faster than the network can supply it, and this network lag can lead to problems. When you call `read()`, you might not get as much data as you expect.

To cope with this, any calls to `read()` should be done in a loop until you get enough data back to process. For instance, the following short routine keeps calling `read()` until the buffer is filled or an error occurs:

```
int ReadData(int fd, char* buf, int n)
{
    int nTtlRead;
    int nRead;

    nTtlRead = 0;
    nRead = 0;

    while (nTtlRead < n)
    {
        nRead = read(fd, buf, n - nTtlRead);

        // An error occurred -- complain!
        if (nRead < 0)
        {
            perror("Error reading!");
            return -1;
        }

        // No more data sent -- all done!
        if (nRead == 0)
            break;

        // Record amount read.
        nTtlRead += nRead;

        // Move buffer on.
        buf += nRead;
    }

    return nTtlRead;
}
```

3.8 Closing the socket

Once the client or server is finished with a socket, it should call `close()` to deallocate it.

3.8.1 close()

Closes a file descriptor.

```
int close(int fd);
```

fd is the file descriptor to close.

4 Implementing the ‘String Length’ protocol

Here are two programs, a client and a server, that implement the example protocol using the functions and techniques listed earlier.

4.1 Sample client

```
/*                              vim:set ts=4 sw=4 noai sr sta et cin:
 * client.c
```

```
 * by Keith Gaughan <kmgaughan@eircom.net>
 *
 * Simple client implementing the ‘String Length’ protocol.
 *
 * Copyright (c) Keith Gaughan, 2003.
 * This software is free; you can redistribute it and/or modify it
 * under the terms of the Design Science License (DSL). If you didn’t
 * recieve a copy of the DSL with this software, one can be obtained
 * at <http://www.dsl.org/copyleft/dsl.txt>.
 */

#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <syslog.h>
#include <sys/types.h>
#include <sys/sockets.h>
#include <netinet/in.h>
#include <netdb.h>

// Sample wrapper from the notes -- connects to a given server.
int ConnectToServer(char* hostname, unsigned short port);

int ReadString(int fd, char* buf, int n);

void main(int argc, char* argv[])
{
    int fd;
    unsigned short port;
    char buf[1024];
    int nRead;

    // Check if enough arguments were passed in.
    if (argc < 3)
    {
        // Not enough! Complain.
        fprintf(stderr, "%s: Not enough arguments!\n", argv[0]);
        fprintf(stderr, "Syntax: %s hostname port\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // argv[2] needs to be parsed to get the port.
    port = (unsigned short) atoi(argv[2]);

    // Connect to server -- argv[1] contains the hostname.
    fd = ConnectToServer(argv[1], port);
    if (fd == -1)
    {
        perror("Could not connect to server!");
        exit(EXIT_FAILURE);
    }

    // Get some input from the user.
    printf("> ");
```

```
    fgets(stdin, buf, sizeof(buf) - 1);

    // Send everything in the buffer to the server.
    write(fd, buf, sizeof(buf));

    // Read the response from the server.
    nRead = ReadString(fd, buf, sizeof buf);
    if (nRead > 0)
        printf("Length of string is: %s\n", buf);
    else
        perror("Something went wrong talking to server!");

    // Clean up and exit.
    close(fd);
}

/**
 * Reads data from the file descriptor until it reaches a NUL.
 *
 * @param  fd   File descriptor to read from.
 * @param  buf  Buffer to write data to.
 * @param  n    Size of buffer.
 *
 * @return Length of string read, or -1 if error.
 */
int ReadString(int fd, char* buf, int n)
{
    int nTtlRead;
    int nRead;

    nTtlRead = 0;
    nRead = 0;

    while (nTtlRead < n)
    {
        // Read a character at a time, and check for error...
        if (read(fd, buf, 1) < 0)
        {
            perror("Error reading!");
            return -1;
        }

        // Have we reached the end?
        if (*buf == '\0')
            break;

        // Record amount read and move buffer on.
        nTtlRead += 1;
        buf += 1;
    }

    return nTtlRead;
}
```

4.2 Sample singleprocessing server

```
/*                              vim:set ts=4 sw=4 noai sr sta et cin:
 * simpleserver.c
 * by Keith Gaughan <kmgaughan@eircom.net>
 *
 * Simple server implementing the ‘String Length’ protocol.
 *
 * Copyright (c) Keith Gaughan, 2003.
 * This software is free; you can redistribute it and/or modify it
 * under the terms of the Design Science License (DSL). If you didn’t
 * recieve a copy of the DSL with this software, one can be obtained
 * at <http://www.dsl.org/copyleft/dsl.txt>.
 */

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

#define QUEUE_SIZE 5

int CreateTCPServerSocket(unsigned short port, int nQueued);
void DispatchIncoming(int fdQueue);     // From earlier in the notes.
void HandleConnection(int fd);
int ReadString(int fd, char* buf, int n);   // From client.c

void main(int argc, char* argv[])
{
    int port;
    int fdQueue;

    // Check if enough arguments were passed in.
    if (argc < 2)
    {
        // Not enough! Complain.
        fprintf(stderr, "%s: Not enough arguments!\n", argv[0]);
        fprintf(stderr, "Syntax: %s port\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    // argv[1] needs to be parsed to get the port.
    port = (unsigned short) atoi(argv[1]);

    // Initialise server.
    fdQueue = CreateTCPServerSocket(port, QUEUE_SIZE);
    if (fdQueue == -1)
        exit(EXIT_FAILURE);
```

```
    // Enter connection dispatch loop.
    DispatchIncoming(fdQueue);
}

/**
 * Opens a socket for the server to listen for connections on.
 *
 * @param  port     Port to bind socket to.
 * @param  nQueued  Maximum number of queued requests to allow.
 *
 * @return Socket to wait for new connections on.
 */
int CreateTCPServerSocket(unsigned short port, int nQueued)
{
    int fd;
    struct sockaddr_in addr;

    // Create a socket.
    fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd == -1)
    {
        perror("Could not create socket.");
        return -1;
    }

    // Build address.
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(port);

    // Bind to any of the server's addresses.
    if (bind(fd, (struct sockaddr*) &addr, sizeof addr) == -1)
    {
        perror("Could not bind socket.");
        close(fd);
        return -1;
    }

    // Listen for request and allow up to nQueued outstanding.
    if (listen(fd, nQueued) == -1)
    {
        perror("listen() failed.");
        close(fd);
        return -1;
    }

    return fd;
}

/**
 * Handle's communications with a client.
 */
void HandleConnection(int fd)
```

```
{
    int nRead;
    char buf[1024];

    // Get any incoming data for processing.
    nRead = ReadString(fd, buf, sizeof buf);

    // With nRead, we've effectively got the length of the string,
    // so write that out.
    sprintf(buf, "%d", nRead);

    // Send the data back to the client.
    write(fd, buf, strlen(buf));

    // Clean up!
    close(fd);
}
```

5 Concurrency

`simpleserver.c`, while fine for a basic demonstration, doesn't scale all that well.

If you imagine a repair shop where one person does everything—they serve customers, do the repairs, clean the toilets, and so on. Each time a customer comes, they have to be dealt with and the shop owner can't go on to the next customer until the current customer's business is finished. In this kind of situation, if there's a large influx of customers, our shop owner might have to turn some of them away because he just can't keep up with the work. Essentially, this is what's happening with `simpleserver.c`.

However, there is a way around this. If our shop owner had some people employed back in the workshop to handle jobs, he could watch the counter and hand off any customer to an available worker. This makes things much more efficient, and is the essence of concurrency.

That just covers server concurrency, but quite often it's useful to have concurrency in your clients too. Imagine how annoying it'd be if your email client or browser froze each time it was fetching or sending mail, or fetching a page?[10]

There are two methods for achieving concurrency: multiprocessing with `fork()`, and multithreading with, um, threads[11].

5.1 Concurrency with fork()

Multiprocessing is the simplest way of achieving server concurrency. This is the way that the Apache Webserver achieves concurrency[12] and some would say this is a superior and safer way to implement concurrency[13]. Introducing multiprocessing into `simpleserver.c` is just a matter of writing a new version of the `DispatchIncoming()` function. As it stands, it can't dispatch another incoming connection until the current one's been dealt with. Here's the new, multiprocessing version:

```
/**
 * Waits on and dispatches any incoming client connections to a
 * handler.
 *
 * @param  fdQueue  File descriptor of socket to wait on.
```

[10] But I think I'll leave this as an exercise to the reader!

[11] Well, duh!

[12] Well, at least it *did* until version two came out, and anyway what it actually does is maintain a *process pool* that it creates beforehand so it doesn't have to fork with each new connection. With version two, it started to use threads instead, the reason being that it made the Win32 version more stable—Windows isn't very good when it comes to processes.

[13] See my *Introduction to POSIX threads* for some reasons why.

```
 */
void DispatchIncoming(int fdQueue)
{
    int pid;
    int fd;

    for (;;)
    {
        // Fetch a new connection off the queue.
        fd = accept(fdQueue, NULL, NULL);

        if (fd == -1)
        {
            // Oh dear! Something's gone wrong! Whimper and die.
            perror("Could not accept incoming connection.");
            exit(EXIT_FAILURE);
        }

        pid = fork();
        if (pid == 0)
        {
            // We're in the child -- dispatch the request!
            close(fdQueue);
            HandleConnection(fd);
            exit(EXIT_SUCCESS);
        }
        else if (pid == -1)
        {
            // Aaarrgh! Moan and die!
            perror("Horribleness upon fork()!");
            close(fdQueue);
            close(fd);
            exit(EXIT_FAILURE);
        }

        // We're the parent.
        close(fd);
    }
}
```

5.1.1 Hmmmm... braaaiiin...

There's one problem with the multiprocessing server as it stands—zombies.

If your server isn't waiting for its children to die, they'll become *zombie processes* and roam around your computer's core looking for heads to crack open so as they can feast on the pink goodness inside[14]. To stop its children from becoming undead, the parent has to wait until it dies so the child can be laid to rest.

In a concurrent server, calling `wait()` after each `fork()` isn't really practical: the parent would be suspended until the child died and we'd be back to the bad old days of `simpleserver.c`. One way around this is to use *signals*.

Signals are events fired when something significant happens in the system. One of these events, called `SIGCHLD`, is triggered when one of a processes children dies. To stop them becoming zombies, we need

[14]... or maybe not.

to catch this event. Using signals requires that you include the library `signal.h`. Here's the extra code required to keep those dead bodies in the ground where they should be:

```
/**
 * Handler for the SIGCHLD event.
 */
void BuryChild(int sig)
{
    // Wait until child is completely dead.
    wait(-1);
}

// Modifications to start of DispatchIncoming():
void DispatchIncoming(int fdQueue)
{
    int pid;
    int fd;

    // Catch dying children.
    signal(SIGCHLD, BuryChild);

    for (;;)
    {
        // ...rest of DispatchIncoming()
```

5.2 Concurrency with threads

Concurrency can also be achieved with threads[15], although using them opens up a whole other desert of sandworms[16].

To adapt `simpleserver.c` to threads, `DispatchIncoming()` and `HandleConnection()` need a bit of fiddling with.

Oh, and don't forget that you need to include `pthread.h` and to include the flag `-lpthread` when linking.

5.2.1 Preparing HandleConnection() for threads

For `HandleConnection()` to be usable with threads, we need to change the way the file descriptor is passed into it. Here's the bits that need to be changed:

```
void* HandleConnection(void* arg)
{
    int fd;
    int nRead;
    char buf[1024];

    // Cast the argument to an int to get the file descriptor.
    fd = (int) arg;

    // ...rest of HandleConnection()

    return NULL;
}
```

[15] At this point, it'd be best if you went off and read my *Introduction to POSIX Threads* if you haven't already, 'cause otherwise you'll get lost.

[16] What? You haven't read *Dune*? Oh well, it seemed like a good joke at the time…

Piece of cake!

5.2.2 Preparing DispatchIncoming() for threads

This is a bit more complicated. Whenever a new connection is accepted, we need to spawn a new thread. Here's the altered version of `DispatchIncoming()` that handles this.

```
void DispatchIncoming(int fdQueue)
{
    pthread_t idThread;
    int fd;

    for (;;)
    {
        // Fetch a new connection off the queue.
        fd = accept(fdQueue, NULL, NULL);

        if (fd == -1)
        {
            // Oh dear! Something's gone wrong! Whimper and die.
            perror("Could not accept incoming connection.");
            exit(EXIT_FAILURE);
        }

        // Got a connection -- spawn the new worker thread.
        pthread_create(&idThread, NULL, HandleConnection, (void*) fd);
    }
}
```

If you feel it's necessary, you can ensure that the server doesn't exit until all the threads have too by including the following call to `pthread_join()` right after the call to `pthread_create()`:

```
pthread_join(idThread, NULL);
```

There are other issues[17] with threads that you should understand before you go fiddling with them. I'd advise you to read up on them[18] before you try and do something that could end up befuddling you no-end.

6 Non-blocking sockets

To understand non-blocking sockets, you have to first understand what a blocking function call is. A function call is *blocking* when the process calling it may be put to sleep, for example if a process calls `read()` when no data is available, `read()` won't return until data becomes available. However, a non-blocking call will return rather than cause the calling process to block.

6.1 A short digression: errno, accept(), and read()

Something I've glossed over up until now it how errors are returned. When a function returns an error, usually denoted with `-1`, there could be any number of things that could have gone wrong. So as to provide some way to find out what happened, there's a global variable called `errno`. To access `errno`, you need to include the header file `errno.h`.

[17] To put it mildly.
[18] Plug! "Introduction to POSIX Threads" Plug!

This is set to various error codes specific to the various functions depending on what error occurred. With non-blocking sockets, `errno`'s value becomes especially important as `accept()` can quite often throw an error.

If your listener socket is non-blocking and `accept()` returns `-1`, it's important to check `ERRNO`'s value. If it's set to the constant `EWOULDBLOCK`, this means that there are no connections present on the connection queue to accept.

The same thing applies to `read()`. If it returns `-1` and `errno` is set to `EWOULDBLOCK`, this means that while there's nothing wrong with the connection and it hasn't been closed, there's no data available as yet to read so you should either wait until there is or do something else.

6.2 Problems with blocking (synchronous) I/O

It's not uncommon for programmers to need to handle multiple file descriptors at once. With blocking I/O, whenever your process attempts to read or write a file descriptor, there's a chance it may be put to sleep until such time as the call can return. Meanwhile you might have another file descriptor with piles of data ready to be processed but your process is stuck and can't do anything about it.

6.3 Enabling non-blocking I/O

A file descriptor can be made non-blocking with the following call:

```
/**
 * Ensures any writes or reads on this file descriptor are
 * non-blocking.
 *
 * @param  fd  File descriptor to put in non-blocking mode.
 */
void MakeNonBlocking(int fd)
{
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) | O_NDELAY);
}
```

6.4 Difficulties with non-blocking I/O

Suppose you have a bunch of non-blocking file descriptors, but none of them have any data pending on them. You could repeatedly attempt `read()`s on them until one does not return `EWOULDBLOCK` in `errno`. This isn't exactly a feasible solution because it would burn up processor time like mad needlessly, time that would be better given to other processes to use. This can be solved using *asynchronous I/O*.

6.5 What's asynchronous I/O

A synchronous I/O is where you get signalled whenever there's data available on a file descriptor you own. There are a few different ways of doing this, but by far the simplest one is to use `select()`. To use `select()`, you must include the following header file:

```
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
```

6.5.1 fd_set

This is a data structure representing a set of file descriptors. You need to use the macros `FD_ZERO`, `FD_SET`, `FD_CLR`, and `FD_ISSET` to manipulate its contents.

6.5.2 FD_CLR()

Clears a file descriptor from an `fd_set`.

```
FD_CLR(int fd, fd_set* set);
```

fd is the file descriptor in question.

set points to the `fd_set` to be manipulated.

6.5.3 FD_SET()

Sets a file descriptor from an `fd_set`. Arguments are the same as for `FD_CLR`.

```
FD_SET(int fd, fd_set* set);
```

6.5.4 FD_ISSET()

Checks is a file descriptor in an `fd_set` is set on. Arguments are the same as for `FD_CLR`. Returns zero if off, non-zero if on.

```
FD_ISSET(int fd, fd_set* set);
```

6.5.5 FD_ZERO()

Totally clears an `fd_set`.

```
FD_ZERO(fd_set* set);
```

set points to the `fd_set` to be cleared.

6.5.6 select()

6.6 Making the multithreaded server asynchronous

7 So long, Pilgrim...

If you've any questions or comments, you'll always be able to get me at <kmgaughan@eircom.net>, though <kgaughan@mcom.cit.ie> should work too. The latest version of this document should be downloadable from either my website or my weblog.