2. What is a socket?

You hear talk of “sockets” all the time, and perhaps you are wondering just what they are exactly. Well, they're this: a way to speak to other programs using standard Unix file descriptors.

What?

Ok—you may have heard some Unix hacker state, “Jeez, *everything* in Unix is a file!” What that person may have been talking about is the fact that when Unix programs do any sort of I/O, they do it by reading or writing to a file descriptor. A file descriptor is simply an integer associated with an open file. But (and here's the catch), that file can be a network connection, a FIFO, a pipe, a terminal, a real on-the-disk file, or just about anything else. Everything in Unix *is* a file! So when you want to communicate with another program over the Internet you're gonna do it through a file descriptor, you'd better believe it.

“Where do I get this file descriptor for network communication, Mr. Smarty-Pants?” is probably the last question on your mind right now, but I'm going to answer it anyway: You make a call to the **`socket()`** system routine. It returns the socket descriptor, and you communicate through it using the specialized **`send()`** and **`recv()`** (**man send**, **man recv**) socket calls.

“But, hey!” you might be exclaiming right about now. “If it's a file descriptor, why in the name of Neptune can't I just use the normal **`read()`** and **`write()`** calls to communicate through the socket?” The short answer is, “You can!” The longer answer is, “You can, but **`send()`** and **`recv()`** offer much greater control over your data transmission.”

What next? How about this: there are all kinds of sockets. There are DARPA Internet addresses (Internet Sockets), path names on a local node (Unix Sockets), CCITT X.25 addresses (X.25 Sockets that you can safely ignore), and probably many others depending on which Unix flavor you run. This document deals only with the first: Internet Sockets.

2.1. Two Types of Internet Sockets

What's this? There are two types of Internet sockets? Yes. Well, no. I'm lying. There are more, but I didn't want to scare you. I'm only going to talk about two types here. Except for this sentence, where I'm going to tell you that “Raw Sockets” are also very powerful and you should look them up.

All right, already. What are the two types? One is “Stream Sockets”; the other is “Datagram Sockets”, which may hereafter be referred to as “`SOCK_STREAM`” and “`SOCK_DGRAM`”, respectively. Datagram sockets are sometimes called “connectionless sockets”. (Though they can be **`connect()`**'d if you really want. See **`connect()`**, below.)

Stream sockets are reliable two-way connected communication streams. If you output two items into the socket in the order “1, 2”, they will arrive in the order “1, 2” at the opposite end. They will also be error-free. I'm so certain, in fact, they will be error-free, that I'm just going to put my fingers in my ears and chant *la la la la* if anyone tries to claim otherwise.

What uses stream sockets? Well, you may have heard of the **telnet** application, yes? It uses stream sockets. All the characters you type need to arrive in the same order you type them, right? Also, web browsers use the HTTP protocol which uses stream sockets to get pages. Indeed, if you telnet to a web site on port 80, and type “`GET / HTTP/1.0`” and hit RETURN twice, it'll dump the HTML back at you!

How do stream sockets achieve this high level of data transmission quality? They use a protocol called “The Transmission Control Protocol”, otherwise known as “TCP” (see RFC 793[6] for extremely detailed info on TCP.) TCP makes sure your data arrives sequentially and error-free. You may have heard “TCP” before as the better half of “TCP/IP” where “IP” stands for “Internet Protocol” (see RFC 791[7].) IP deals primarily with Internet routing and is not generally responsible for data integrity.

Cool. What about Datagram sockets? Why are they called connectionless? What is the deal, here, anyway? Why are they unreliable? Well, here are some facts: if you send a datagram, it may arrive. It may arrive out of order. If it arrives, the data within the packet will be error-free.

6. `http://tools.ietf.org/html/rfc793`
7. `http://tools.ietf.org/html/rfc791`

Datagram sockets also use IP for routing, but they don't use TCP; they use the "User Datagram Protocol", or "UDP" (see RFC 768[8].)

Why are they connectionless? Well, basically, it's because you don't have to maintain an open connection as you do with stream sockets. You just build a packet, slap an IP header on it with destination information, and send it out. No connection needed. They are generally used either when a TCP stack is unavailable or when a few dropped packets here and there don't mean the end of the Universe. Sample applications: **tftp** (trivial file transfer protocol, a little brother to FTP), **dhcpcd** (a DHCP client), multiplayer games, streaming audio, video conferencing, etc.

"Wait a minute! **tftp** and **dhcpcd** are used to transfer binary applications from one host to another! Data can't be lost if you expect the application to work when it arrives! What kind of dark magic is this?"

Well, my human friend, **tftp** and similar programs have their own protocol on top of UDP. For example, the tftp protocol says that for each packet that gets sent, the recipient has to send back a packet that says, "I got it!" (an "ACK" packet.) If the sender of the original packet gets no reply in, say, five seconds, he'll re-transmit the packet until he finally gets an ACK. This acknowledgment procedure is very important when implementing reliable `SOCK_DGRAM` applications.

For unreliable applications like games, audio, or video, you just ignore the dropped packets, or perhaps try to cleverly compensate for them. (Quake players will know the manifestation this effect by the technical term: *accursed lag*. The word "accursed", in this case, represents any extremely profane utterance.)

Why would you use an unreliable underlying protocol? Two reasons: speed and speed. It's way faster to fire-and-forget than it is to keep track of what has arrived safely and make sure it's in order and all that. If you're sending chat messages, TCP is great; if you're sending 40 positional updates per second of the players in the world, maybe it doesn't matter so much if one or two get dropped, and UDP is a good choice.

2.2. Low level Nonsense and Network Theory

Since I just mentioned layering of protocols, it's time to talk about how networks really work, and to show some examples of how `SOCK_DGRAM` packets are built. Practically, you can probably skip this section. It's good background, however.

Data Encapsulation.

Hey, kids, it's time to learn about *Data Encapsulation*! This is very very important. It's so important that you might just learn about it if you take the networks course here at Chico State `;-)`. Basically, it says this: a packet is born, the packet is wrapped ("encapsulated") in a header (and rarely a footer) by the first protocol (say, the TFTP protocol), then the whole thing (TFTP header included) is encapsulated again by the next protocol (say, UDP), then again by the next (IP), then again by the final protocol on the hardware (physical) layer (say, Ethernet).

When another computer receives the packet, the hardware strips the Ethernet header, the kernel strips the IP and UDP headers, the TFTP program strips the TFTP header, and it finally has the data.

Now I can finally talk about the infamous *Layered Network Model* (aka "ISO/OSI"). This Network Model describes a system of network functionality that has many advantages over other models. For instance, you can write sockets programs that are exactly the same without caring how the data is physically transmitted (serial, thin Ethernet, AUI, whatever) because programs on lower levels deal with it for you. The actual network hardware and topology is transparent to the socket programmer.

Without any further ado, I'll present the layers of the full-blown model. Remember this for network class exams:

- Application

8. `http://tools.ietf.org/html/rfc768`

- Presentation
- Session
- Transport
- Network
- Data Link
- Physical

The Physical Layer is the hardware (serial, Ethernet, etc.). The Application Layer is just about as far from the physical layer as you can imagine—it's the place where users interact with the network.

Now, this model is so general you could probably use it as an automobile repair guide if you really wanted to. A layered model more consistent with Unix might be:

- Application Layer (*telnet, ftp, etc.*)
- Host-to-Host Transport Layer (*TCP, UDP*)
- Internet Layer (*IP and routing*)
- Network Access Layer (*Ethernet, wi-fi, or whatever*)

At this point in time, you can probably see how these layers correspond to the encapsulation of the original data.

See how much work there is in building a simple packet? Jeez! And you have to type in the packet headers yourself using "**cat**"! Just kidding. All you have to do for stream sockets is **`send()`** the data out. All you have to do for datagram sockets is encapsulate the packet in the method of your choosing and **`sendto()`** it out. The kernel builds the Transport Layer and Internet Layer on for you and the hardware does the Network Access Layer. Ah, modern technology.

So ends our brief foray into network theory. Oh yes, I forgot to tell you everything I wanted to say about routing: nothing! That's right, I'm not going to talk about it at all. The router strips the packet to the IP header, consults its routing table, blah blah blah. Check out the IP RFC[9] if you really really care. If you never learn about it, well, you'll live.

9. `http://tools.ietf.org/html/rfc791`