Udp socket programming in c pdf

Continue

1 Intro 2 What is a socket? 2.1 Two Types of Internet Sockets 2.2 Low level Nonsense and Network Theory 3 IP Addresses, structs, and Data Munging 4 Jumping from IPv4 to IPv6 5 System Calls or Bust 6 Client-Server Background 6.1 A Simple Stream Server 6.2 A Simple Stream Client 6.3 Datagram Sockets 7 Slightly Advanced Techniques 8 Common Questions 9 Man Pages 9.1 accept() 9.2 bind() 9.3 connect() 9.4 close() 9.5 getaddrinfo(), freeaddrinfo(), gai_strerror() 9.6 gethostname() 9.7 gethostbyname(), gethostbyaddr() 9.8 getnameinfo() 9.9 getpeername() 9.10 errno 9.11 fcntl() 9.12 htons(), htonl(), ntohs(), ntohl() 9.13 inet_ntoa(), inet_aton(), inet_addr 9.14 inet_ntop(), inet_pton() 9.15 listen() 9.16 perror(), strerror() 9.17 poll() 9.18 recv(), recvfrom() 9.19 select() 9.20 setsockopt(), getsockopt() 9.21 send(), sendto() 9.22 shutdown() 9.23 socket() 9.24 struct sockaddr and pals 10 More References 10.1 Books 10.2 Web References 10.3 RFCs Hey! Socket programming got you down? Is this stuff just a little too difficult to figure out from the man pages? You want to do cool Internet programming, but you don't have time to wade through a gob of structs trying to figure out if you have to call bind() before you connect(), etc., etc. Well, guess what! I've already done this nasty business, and I'm dying to share the information with everyone! You've come to the right place. This document should give the average competent C programmer the edge s/he needs to get a grip on this networking noise. And check it out: I've finally caught up with the future (just in the nick of time, too!) and have updated the Guide for IPv6! Enjoy! Audience This document has been written as a tutorial, not a complete reference. It is probably at its best when read by individuals who are just starting out with socket programming and are looking for a foothold. It is certainly not the complete and total guide to sockets programming, by any means. Hopefully, though, it'll be just enough for those man pages to start making sense... :-) Platform and Compiler The code contained within this document was compiled on a Linux PC using Gnu's gcc compiler. It should, however, build on just about any platform that uses gcc. Naturally, this doesn't apply if you're programming for Windows—see the section on Windows programming, below. Official Homepage and Books For Sale This official location of this document is: There you will also find example code and translations of the guide into various languages. To buy nicely bound print copies (some call them "books"), visit: I'll appreciate the purchase because it helps sustain my document-writing lifestyle! Note for Solaris/SunOS Programmers When compiling for Solaris or SunOS, you need to specify some extra command-line switches for linking in the proper libraries. In order to do this, simply add "-lnsl -lsocket -lresolv" to the end of the compile command, like so: $ cc -o server server.c -lnsl -lsocket -lresolv If you still get errors, you could try further adding a -lxnet to the end of that command line. I don't know what that does, exactly, but some people seem to need it. Another place that you might find problems is in the call to setsockopt(). The prototype differs from that on my Linux box, so instead of: enter this: As I don't have a Sun box, I haven't tested any of the above information—it's just what people have told me through email. Note for Windows Programmers At this point in the guide, historically, I've done a bit of bagging on Windows, simply due to the fact that I don't like it very much. But I should really be fair and tell you that Windows has a huge install base and is obviously a perfectly fine operating system. They say absence makes the heart grow fonder, and in this case, I believe it to be true. (Or maybe it's age.) But what I can say is that after a decade-plus of not using Microsoft OSes for my personal work, I'm much happier! As such, I can sit back and safely say, "Sure, feel free to use Windows!" ...Ok yes, it does make me grit my teeth to say that. So I still encourage you to try Linux1, BSD2, or some flavor of Unix, instead. But people like what they like, and you Windows folk will be pleased to know that this information is generally applicable to you guys, with a few minor changes, if any. One cool thing you can do is install Cygwin3, which is a collection of Unix tools for Windows. I've heard on the grapevine that doing so allows all these programs to compile unmodified. Another thing that you should consider is the Windows Subsystem for Linux4. This basically allows you to install a Linux VM-ish thing on Windows 10. That will also definitely get you situated. But some of you might want to do things the Pure Windows Way. That's very gutsy of you, and this is what you have to do: run out and get Unix immediately! No, no—I'm kidding. I'm supposed to be Windows-friendly(er) these days... This is what you'll have to do (unless you install Cygwin!): first, ignore pretty much all of the system header files I mention in here. All you need to include is: Wait! You also have to make a call to WSAStartup() before doing anything else with the sockets library. The code to do that looks something like this: #include { WSADATA wsaData; // if this doesn't work //WSAData wsaData; // then try this instead // MAKEWORD(1,1) for Winsock 1.1, MAKEWORD(2,0) for Winsock 2.0: if (WSAStartup(MAKEWORD(1,1), &wsaData) != 0) { fprintf(stderr, "WSAStartup failed.\n"); exit(1); } You also have to tell your compiler to link in the Winsock library, usually called wsock32.lib or winsock32.lib, or ws2_32.lib for Winsock 2.0. Under VC++, this can be done through the Project menu, under Settings.... Click the Link tab, and look for the box titled "Object/library modules". Add "wsock32.lib" (or whichever lib is your preference) to that list. Or so I hear. Finally, you need to call WSACleanup() when you're all through with the sockets library. See your online help for details. Once you do that, the rest of the examples in this tutorial should generally apply, with a few exceptions. For one thing, you can't use close() to close a socket—you need to use closesocket(), instead. Also, select() only works with socket descriptors, not file descriptors (like 0 for stdin). There is also a socket class that you can use, CSocket. Check your compilers help pages for more information. To get more information about Winsock, read the Winsock FAQ5 and go from there. Finally, I hear that Windows has no fork() system call which is, unfortunately, used in some of my examples. Maybe you have to link in a POSIX library or something to get it to work, or you can use CreateProcess() instead. fork() takes no arguments, and CreateProcess() takes about 48 billion arguments. If you're not up to that, the CreateThread() is a little easier to digest...unfortunately a discussion about multithreading is beyond the scope of this document. I can only talk about so much, you know! Email Policy I'm generally available to help out with email questions so feel free to write in, but I can't guarantee a response. I lead a pretty busy life and there are times when I just can't answer a question you have. When that's the case, I usually just delete the message. It's nothing personal; I just won't ever have the time to give the detailed answer you require. As a rule, the more complex the question, the less likely I am to respond. If you can narrow down your question before mailing it and be sure to include any pertinent information (like platform, compiler, error messages you're getting, and anything else you think might help me troubleshoot), you're much more likely to get a response. For more pointers, read ESR's document, How To Ask Questions The Smart Way6. If you don't get a response, hack on it some more, try to find the answer, and if it's still elusive, then write me again with the information you've found and hopefully it will be enough for me to help out. Now that I've badgered you about how to write and not write me, I'd just like to let you know that I fully appreciate all the praise the guide has received over the years. It's a real morale boost, and it gladdens me to hear that it is being used for good! :-) Thank you! Mirroring You are more than welcome to mirror this site, whether publicly or privately. If you publicly mirror the site and want me to link to it from the main page, drop me a line at beej@beej.us. Note for Translators If you want to translate the guide into another language, write me at beej@beej.us and I'll link to your translation from the main page. Feel free to add your name and contact info to the translation. This source markdown document uses UTF-8 encoding. Please note the license restrictions in the Copyright, Distribution, and Legal section, below. If you want me to host the translation, just ask. I'll also link to it if you want to host it; either way is fine. Copyright, Distribution, and Legal Beej's Guide to Network Programming is Copyright © 2019 Brian "Beej Jorgensen" Hall. With specific exceptions for source code and translations, below, this work is licensed under the Creative Commons Attribution- Noncommercial- No Derivative Works 3.0 License. To view a copy of this license, visit or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA. One specific exception to the "No Derivative Works" portion of the license is as follows: this guide may be freely translated into any language, provided the translation is accurate, and the guide is reprinted in its entirety. The same license restrictions apply to the translation as to the original guide. The translation may also include the name and contact information for the translator. The C source code presented in this document is hereby granted to the public domain, and is completely free of any license restriction. Educators are freely encouraged to recommend or supply copies of this guide to their students. Unless otherwise mutually agreed by the parties in writing, the author offers the work as-is and makes no representations or warranties of any kind concerning the work, express, implied, statutory or otherwise, including, without limitation, warranties of title, merchantibility, fitness for a particular purpose, noninfringement, or the absence of latent or other defects, accuracy, or the presence of absence of errors, whether or not discoverable. Except to the extent required by applicable law, in no event will the author be liable to you on any legal theory for any special, incidental, consequential, punitive or exemplary damages arising out of the use of the work, even if the author has been advised of the possibility of such damages. Contact beej@beej.us for more information. Dedication Thanks to everyone who has helped in the past and future with me getting this guide written. And thank you to all the people who produce the Free software and packages that I use to make the Guide: GNU, Linux, Slackware, vim, Python, Inkscape, pandoc, many others. And finally a big thank-you to the literally thousands of you who have written in with suggestions for improvements and words of encouragement. I dedicate this guide to some of my biggest heroes and inpirators in the world of computers: Donald Knuth, Bruce Schneier, W. Richard Stevens, and The Woz, my Readership, and the entire Free and Open Source Software Community. This book is written in Markdown using the vim editor on an Arch Linux box loaded with GNU tools. The cover "art" and diagrams are produced with Inkscape. The Markdown is converted to HTML and LaTex/PDF by Python, Pandoc and XeLaTeX, using Liberation fonts. The toolchain is composed of 100% Free and Open Source Software. 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. 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 Hypertext Transfer Protocol (HTTP) 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! If you don't have telnet installed and don't want to install it, or your telnet is being picky about connecting to clients, the guide comes with a telnet-like program called telnot7. This should work well for all the needs of the guide. (Note that telnet is actually a spec'd networking protocol8, and telnot doesn't implement this protocol at all.) 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 7939 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 79110). 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. Datagram sockets also use IP for routing, but they don't use TCP; they use the "User Datagram Protocol", or "UDP" (see RFC 76811). 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. 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 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 RFC12 if you really really care. If you never learn about it, well, you'll live. IP Addresses, structs, and Data Munging Here's the part of the game where we get to talk code for a change. But first, let's discuss more non-code! Yay! First I want to talk about IP addresses and ports for just a tad so we have that sorted out. Then we'll talk about how the sockets API stores and manipulates IP addresses and other data. IP Addresses, versions 4 and 6 In the good old days back when Ben Kenobi was still called Obi Wan Kenobi, there was a wonderful network routing system called The Internet Protocol Version 4, also called IPv4. It had addresses made up of four bytes (A.K.A. four "octets"), and was commonly written in "dots and numbers" form, like so: 192.0.2.111. You've probably seen it around. In fact, as of this writing, virtually every site on the Internet uses IPv4. Everyone, including Obi Wan, was happy. Things were great, until some naysayer by the name of Vint Cerf warned everyone that we were about to run out of IPv4 addresses! (Besides warning everyone of the Coming IPv4 Apocalypse Of Doom And Gloom, Vint Cerf13 is also well-known for being The Father Of The Internet. So I really am in no position to second-guess his judgment.) Run out of addresses? How could this be? I mean, there are like billions of IP addresses in a 32-bit IPv4 address. Do we really have billions of computers out there? Yes. Also, in the beginning, when there were only a few computers and everyone thought a billion was an impossibly large number, some big organizations were generously allocated millions of IP addresses for their own use. (Such as Xerox, MIT, Ford, HP, IBM, GE, AT&T, and some little company called Apple, to name a few.) In fact, if it weren't for several stopgap measures, we would have run out a long time ago. But now we're living in an era where we're talking about every human having an IP address, every computer, every calculator, every phone, every parking meter, and (why not) every puppy dog, as well. And so, IPv6 was born. Since Vint Cerf is probably immortal (even if his physical form should pass on, heaven forbid, he is probably already existing as some kind of hyper-intelligent ELIZA14 program out in the depths of the Internet2), no one wants to have to hear him say again "I told you so" if we don't have enough addresses in the next version of the Internet Protocol. What does this suggest to you? That we need a lot more addresses. That we need not just twice as many addresses, not a billion times as many, not a thousand trillion times as many, but 79 MILLION BILLION TRILLION times as many possible addresses! That'll show 'em! You're saying, "Beej, is that true? I have every reason to disbelieve large numbers." Well, the difference between 32 bits and 128 bits might not sound like a lot; it's only 96 more bits, right? But remember, we're talking powers here: 32 bits represents some 4 billion numbers (232), while 128 bits represents about 340 trillion trillion trillion numbers (for real, 2128). That's like a million IPv4 Internets for every single star in the Universe. Forget this dots-and-numbers look of IPv4, too; now we've got a hexadecimal representation, with each two-byte chunk separated by a colon, like this: 2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551 That's not all! Lots of times, you'll have an IP address with lots of zeros in it, and you can compress them between two colons. And you can leave off leading zeros for each byte pair. For instance, each of these pairs of addresses are equivalent: 2001:0db8:c9d2:0012:0000:0000:0000:0051 2001:db8:c9d2:12::51 2001:0db8:ab00:0000:0000:0000:0000:0000 2001:db8:ab00:: 0000:0000:0000:0000:0000:0000:0000:0001 ::1 The address ::1 is the loopback address. It always means "this machine I'm running on now". In IPv4, the loopback address is 127.0.0.1. Finally, there's an IPv4-compatibility mode for IPv6 addresses that you might come across. If you want, for example, to represent the IPv4 address 192.0.2.33 as an IPv6 address, you use the following notation: "::ffff:192.0.2.33". We're talking serious fun. In fact, it's such serious fun, that the Creators of IPv6 have quite cavalierly lopped off trillions and trillions of addresses for reserved use, but we have so many, frankly, who's even counting anymore? There are plenty left over for every man, woman, child, puppy, and parking meter on every planet in the galaxy. And believe me, every planet in the galaxy has parking meters. You know it's true. Subnets For organizational reasons, it's sometimes convenient to declare that "this first part of this IP address up through this bit is the network portion of the IP address, and the remainder is the host portion. For instance, with IPv4, you might have 192.0.2.12, and we could say that the first three bytes are the network and the last byte was the host. Or, put another way, we're talking about host 12 on network 192.0.2.0 (see how we zero out the byte that was the host). And now for more outdated information! Ready? In the Ancient Times, there were "classes" of subnets, where the first one, two, or three bytes of the address was the network part. If you were lucky enough to have one byte for the network and three for the host, you could have 24 bits-worth of hosts on your network (16 million or so). That was a "Class A" network. On the opposite end was a "Class C", with three bytes of network, and one byte of host (256 hosts, minus a couple that were reserved). So as you can see, there were just a few Class As, a huge pile of Class Cs, and some Class Bs in the middle. The network portion of the IP address is described by something called the netmask, which you bitwise-AND with the IP address to get the network number out of it. The netmask usually looks something like 255.255.255.0. (E.g. with that netmask, if your IP is 192.0.2.12, then your network is 192.0.2.12 AND 255.255.255.0 which gives 192.0.2.0.) Unfortunately, it turned out that this wasn't fine-grained enough for the eventual needs of the Internet; we were running out of Class C networks quite quickly, and we were most definitely out of Class As, so don't even bother to ask. To remedy this, The Powers That Be allowed for the netmask to be an arbitrary number of bits, not just 8, 16, or 24. So you might have a netmask of, say 255.255.255.252, which is 30 bits of network, and 2 bits of host allowing for four hosts on the network. (Note that the netmask is ALWAYS a bunch of 1-bits followed by a bunch of 0-bits.) But it's a bit unwieldy to use a big string of numbers like 255.192.0.0 as a netmask. First of all, people don't have an intuitive idea of how many bits that is, and secondly, it's really not compact. So the New Style came along, and it's much nicer. You just put a slash after the IP address, and then follow that by the number of network bits in decimal. Like this: 192.0.2.12/30. Or, for IPv6, something like this: 2001:db8::/32 or 2001:db8:5413:4028::9db9/64. Port Numbers If you'll kindly remember, I presented you earlier with the Layered Network Model which had the Internet Layer (IP) split off from the Host-to-Host Transport Layer (TCP and UDP). Get up to speed on that before the next paragraph. Turns out that besides an IP address (used by the IP layer), there is another address that is used by TCP (stream sockets) and, coincidentally, by UDP (datagram sockets). It is the port number. It's a 16-bit number that's like the local address for the connection. Think of the IP address as the street address of a hotel, and the port number as the room number. That's a decent analogy; maybe later I'll come up with one involving the automobile industry. Say you want to have a computer that handles incoming mail AND web services—how do you differentiate between the two on a computer with a single IP address? Well, different services on the Internet have different well-known port numbers. You can see them all in the Big IANA Port List15 or, if you're on a Unix box, in your /etc/services file. HTTP (the web) is port 80, telnet is port 23, SMTP is port 25, the game DOOM16 used port 666, etc. and so on. Ports under 1024 are often considered special, and usually require special OS privileges to use. And that's about it! Byte Order By Order of the Realm! There shall be two byte orderings, hereafter to be known as Lame and Magnificent! I joke, but one really is better than the other. :-) There really is no easy way to say this, so I'll just blurt it out: your computer might have been storing bytes in reverse order behind your back. I know! No one wanted to have to tell you. The thing is, everyone in the Internet world has generally agreed that if you want to represent the two-byte hex number, say b34f, you'll store it in two sequential bytes b3 followed by 4f. Makes sense, and, as Wilford Brimley17 would tell you, it's the Right Thing To Do. This number, stored with the big end first, is called Big-Endian. Unfortunately, a few computers scattered here and there throughout the world, namely anything with an Intel or Intel-compatible processor, store the bytes reversed, so b34f would be stored in memory as the sequential bytes 4f followed by b3. This storage method is called Little-Endian. But wait, I'm not done with terminology yet! The more-sane Big-Endian is also called Network Byte Order because that's the order us network types like. Your computer stores numbers in Host Byte Order. If it's an Intel 80x86, Host Byte Order is Little-Endian. If it's a Motorola 68k, Host Byte Order is Big-Endian. If it's a PowerPC, Host Byte Order is... well, it depends! A lot of times when you're building packets or filling out data structures you'll need to make sure your two- and four-byte numbers are in Network Byte Order. But how can you do this if you don't know the native Host Byte Order? Good news! You just get to assume the Host Byte Order isn't right, and you always run the value through a function to set it to Network Byte Order. The function will do the magic conversion if it has to, and this way your code is portable to machines of differing endianness. All righty. There are two types of numbers that you can convert: short (two bytes) and long (four bytes). These functions work for the unsigned variations as well. Say you want to convert a short from Host Byte Order to Network Byte Order. Start with "h" for "host", follow it with "to", then "n" for "network", and "s" for "short": h-to-n-s, or htons() (read: "Host to Network Short"). It's almost too easy... You can use every combination of "n", "h", "s", and "l" you want, not counting the really stupid ones. For example, there is NOT a stolh() ("Short to Long Host") function—not at this party, anyway. But there are: htons() host to network short htonl() host to network long ntohs() network to host short ntohl() network to host long Basically, you'll want to convert the numbers to Network Byte Order before they go out on the wire, and convert them to Host Byte Order as they come in off the wire. I don't know of a 64-bit variant, sorry. And if you want to do floating point, check out the section on Serialization, far below. Assume the numbers in this document are in Host Byte Order unless I say otherwise. structs Well, we're finally here. It's time to talk about programming. In this section, I'll cover various data types used by the sockets interface, since some of them are a real bear to figure out. First the easy one: a socket descriptor. A socket descriptor is the following type: Just a regular int. Things get weird from here, so just read through and bear with me. My First Struct™—struct addrinfo. This structure is a more recent invention, and is used to prep the socket address structures for subsequent use. It's also used in host name lookups, and service name lookups. That'll make more sense later when we get to actual usage, but just know for now that it's one of the first things you'll call when making a connection. struct addrinfo { int ai_flags; // AI_PASSIVE, AI_CANONNAME, etc. int ai_family; // AF_INET, AF_INET6, AF_UNSPEC int ai_socktype; // SOCK_STREAM, SOCK_DGRAM int ai_protocol; // use 0 for "any" size_t ai_addrlen; // size of ai_addr in bytes struct sockaddr *ai_addr; // struct sockaddr_in or _in6 char *ai_canonname; // full canonical hostname struct addrinfo *ai_next; // linked list, next node }; You'll load this struct up a bit, and then call getaddrinfo(). It'll return a pointer to a new linked list of these structures filled out with all the goodies you need. You can force it to use IPv4 or IPv6 in the ai_family field, or leave it as AF_UNSPEC to use whatever. This is cool because your code can be IP version-agnostic. Note that this is a linked list: ai_next points at the next element—there could be several results for you to choose from. I'd use the first result that worked, but you might have different business needs; I don't know everything, man! You'll see that the ai_addr field in the struct addrinfo is a pointer to a struct sockaddr. This is where we start getting into the nitty-gritty details of what's inside an IP address structure. You might not usually need to write to these structures; oftentimes, a call to getaddrinfo() to fill out your struct addrinfo for you is all you'll need. You will, however, have to peer inside these structs to get the values out, so I'm presenting them here. (Also, all the code written before struct addrinfo was invented we packed all this stuff by hand, so you'll see a lot of IPv4 code out in the wild that does exactly that. You know, in old versions of this guide and so on.) Some structs are IPv4, some are IPv6, and some are both. I'll make notes of which are what. Anyway, the struct sockaddr holds socket address information for many types of sockets. struct sockaddr { unsigned short sa_family; // address family, AF_xxx char sa_data[14]; // 14 bytes of protocol address }; sa_family can be a variety of things, but it'll be AF_INET (IPv4) or AF_INET6 (IPv6) for everything we do in this document. sa_data contains a destination address and port number for the socket. This is rather unwieldy since you don't want to tediously pack the address in the sa_data by hand. To deal with struct sockaddr, programmers created a parallel structure: struct sockaddr_in ("in" for "Internet") to be used with IPv4. And this is the important bit: a pointer to a struct sockaddr_in can be cast to a pointer to a struct sockaddr and vice-versa. So even though connect() wants a struct sockaddr*, you can still use a struct sockaddr_in and cast it at the last minute! // (IPv4 only--see struct sockaddr_in6 for IPv6) struct sockaddr_in { short int sin_family; // Address family, AF_INET unsigned short int sin_port; // Port number struct in_addr sin_addr; // Internet address unsigned char sin_zero[8]; // Same size as struct sockaddr }; This structure makes it easy to reference elements of the socket address. Note that sin_zero (which is included to pad the structure to the length of a struct sockaddr) should be set to all zeros with the function memset(). Also, notice that sin_family corresponds to sa_family in a struct sockaddr and should be set to "AF_INET". Finally, the sin_port must be in Network Byte Order (by using htons()!) Let's dig deeper! You see the sin_addr field is a struct in_addr. What is that thing? Well, not to be overly dramatic, but it's one of the scariest unions of all time: // (IPv4 only--see struct in6_addr for IPv6) // Internet address (a structure for historical reasons) struct in_addr { uint32_t s_addr; // that's a 32-bit int (4 bytes) }; Whoa! Well, it used to be a union, but now those days seem to be gone. Good riddance. So if you have declared ina to be of type struct sockaddr_in, then ina.sin_addr.s_addr references the 4-byte IP address (in Network Byte Order). Note that even if your system still uses the God-awful union for struct in_addr, you can still reference the 4-byte IP address in exactly the same way as I did above (this due to #defines). What about IPv6? Similar structs exist for it, as well: // (IPv6 only--see struct sockaddr_in and struct in_addr for IPv4) struct sockaddr_in6 { u_int16_t sin6_family; // address family, AF_INET6 u_int16_t sin6_port; // port number, Network Byte Order u_int32_t sin6_flowinfo; // IPv6 flow information struct in6_addr sin6_addr; // IPv6 address u_int32_t sin6_scope_id; // Scope ID }; struct in6_addr { unsigned char s6_addr[16]; // IPv6 address }; Note that IPv6 has an IPv6 address and a port number, just like IPv4 has an IPv4 address and a port number. Also note that I'm not going to talk about the IPv6 flow information or Scope ID fields for the moment... this is just a starter guide. :-) Last but not least, here is another simple structure, struct sockaddr_storage that is designed to be large enough to hold both IPv4 and IPv6 structures. See, for some calls, sometimes you don't know in advance if it's going to fill out your struct sockaddr with an IPv4 or IPv6 address. So you pass in this parallel structure, very similar to struct sockaddr except larger, and then cast it to the type you need: struct sockaddr_storage { sa_family_t ss_family; // address family // all this is padding, implementation specific, ignore it: char __ss_pad1[_SS_PAD1SIZE]; int64_t __ss_align; char __ss_pad2[_SS_PAD2SIZE]; }; What's important is that you can see the address family in the ss_family field—check this to see if it's AF_INET or AF_INET6 (for IPv4 or IPv6). Then you can cast it to a struct sockaddr_in or struct sockaddr_in6 if you wanna. IP Addresses, Part Deux Fortunately for you, there are a bunch of functions that allow you to manipulate IP addresses. No need to figure them out by hand and stuff them in a long with the ai_family, res->ai_socktype, res->ai_protocol); bind(sockfd, res->ai_addr, res->ai_addrlen); listen(sockfd, BACKLOG); // now accept an incoming connection: addr_size = sizeof their_addr; new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size); // ready to communicate on socket descriptor new_fd! . . . Again, note that we will use the socket descriptor new_fd for all send() and recv() calls. If you're only getting one single connection ever, you can close() the listening sockfd in order to prevent more incoming connections on the same port, if you so desire. send() and recv()—Talk to me, baby! These two functions are for communicating over stream sockets or connected datagram sockets. If you want to use regular unconnected datagram sockets, you'll need to see the section on sendto() and recvfrom(), below. The send() call: int send(int sockfd, const void *msg, int len, int flags); sockfd is the socket descriptor you want to send data to (whether it's the one returned by socket() or the one you got with accept()). msg is a pointer to the data you want to send, and len is the length of that data in bytes. Just set flags to 0. (See the send() man page for more information concerning flags.) Some sample code might be: char *msg = "Beej was here!"; int len, bytes_sent; . . . len = strlen(msg); bytes_sent = send(sockfd, msg, len, 0); . . . send() returns the number of bytes actually sent out—this might be less than the number you told it to send! See, sometimes you tell it to send a whole gob of data and it just can't handle it. It'll fire off as much of the data as it can, and trust you to send the rest later. Remember, if the value returned by send() doesn't match the value in len, it's up to you to send the rest of the string. The good news is this: if the packet is small (less than 1K or so) it will probably manage to send the whole thing all in one go. Again, -1 is returned on error, and errno is set to the error number. The recv() call is similar in many respects: int recv(int sockfd, void *buf, int len, int flags); sockfd is the socket descriptor to read from, buf is the buffer to read the information into, len is the maximum length of the buffer, and flags can again be set to 0. (See the recv() man page for flag information.) recv() returns the number of bytes actually read into the buffer, or -1 on error (with errno set, accordingly). Wait! recv() can return 0. This can mean only one thing: the remote side has closed the connection on you! A return value of 0 is recv()'s way of letting you know this has occurred. There, that was easy, wasn't it? You can now pass data back and forth on stream sockets! Whee! You're a Unix Network Programmer! sendto() and recvfrom()—Talk to me, DGRAM-style "This is all fine and dandy," I hear you saying, "but where does this leave me with unconnected datagram sockets?" No problemo, amigo. We have just the thing. Since datagram sockets aren't connected to a remote host, guess which piece of information we need to give before we send a packet? That's right! The destination address! Here's the scoop: int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, socklen_t tolen); As you can see, this call is basically the same as the call to send() with the addition of two other pieces of information. to is a pointer to a struct sockaddr (which will probably be another struct sockaddr_in or struct sockaddr_in6 or struct sockaddr_storage that you cast at the last minute) which contains the destination IP address and port. tolen, an int deep-down, can simply be set to sizeof *to or sizeof(struct sockaddr_storage). To get your hands on the destination address structure, you'll probably either get it from getaddrinfo(), or from recvfrom(), below, or you'll fill it out by hand. Just like with send(), sendto() returns the number of bytes actually sent (which, again, might be less than the number of bytes you told it to send!), or -1 on error. Equally similar are recv() and recvfrom(). The synopsis of recvfrom() is: int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen); Again, this is just like recv() with the addition of a couple fields. from is a pointer to a local struct sockaddr_storage that will be filled with the IP address and port of the originating machine. fromlen is a pointer to a local int that should be initialized to sizeof *from or sizeof(struct sockaddr_storage). When the function returns, fromlen will contain the length of the address actually stored in from. recvfrom() returns the number of bytes received, or -1 on error (with errno set accordingly). So, here's a question: why do we use struct sockaddr_storage as the socket type? Why not struct sockaddr_in? Because, you see, we want to not tie ourselves down to IPv4 or IPv6. So we use the generic struct sockaddr_storage which we know will be big enough for either. (So... here's another question: why isn't struct sockaddr itself big enough for any address? We even cast the general-purpose struct sockaddr_storage to the general-purpose struct sockaddr! Seems extraneous and redundant, huh. The answer is, it just isn't big enough, and I'd guess that changing it at this point would be Problematic. So they made a new one.) Remember, if you connect() a datagram socket, you can then simply use send() and recv() for all your transactions. The socket itself is still a datagram socket and the packets still use UDP, but the socket interface will automatically add the destination and source information for you. close() and shutdown()—Get outta my face! Whew! You've been send()ing and recv()ing data all day long, and you've had it. You're ready to close the connection on your socket descriptor. This is easy. You can just use the regular Unix file descriptor close() function: This will prevent any more reads and writes to the socket. Anyone attempting to read or write the socket on the remote end will receive an error. Just in case you want a little more control over how the socket closes, you can use the shutdown() function. It allows you to cut off communication in a certain direction, or both ways (just like close() does). Synopsis: int shutdown(int sockfd, int how); sockfd is the socket file descriptor you want to shutdown, and how is one of the following: 0 Further receives are disallowed 1 Further sends are disallowed 2 Further sends and receives are disallowed (like close()) shutdown() returns 0 on success, and -1 on error (with errno set accordingly). If you deign to use shutdown() on unconnected datagram sockets, it will simply make the socket unavailable for further send() and recv() calls (remember that you can use these if you connect() your datagram socket). It's important to note that shutdown() doesn't actually close the file descriptor—it just changes its usability. To free a socket descriptor, you need to use close(). Nothing to it. (Except to remember that if you're using Windows and Winsock that you should call closesocket() instead of close().) getpeername()—Who are you? This function is so easy. It's so easy, I almost didn't give it its own section. But here it is anyway. The function getpeername() will tell you who is at the other end of a connected stream socket. The synopsis: #include int getpeername(int sockfd, struct sockaddr *addr, int *addrlen); sockfd is the descriptor of the connected stream socket, addr is a pointer to a struct sockaddr (or a struct sockaddr_in) that will hold the information about the other side of the connection, and addrlen is a pointer to an int, that should be initialized to sizeof *addr or sizeof(struct sockaddr). The function returns -1 on error and sets errno accordingly. Once you have their address, you can use inet_ntop(), getnameinfo(), or gethostbyaddr() to print or get more information. No, you can't get their login name. (Ok, ok. If the other computer is running an ident daemon, this is possible. This, however, is beyond the scope of this document. Check out RFC 141322 for more info.) gethostname()—Who am I? Even easier than getpeername() is the function gethostname(). It returns the name of the computer that your program is running on. The name can then be used by gethostbyname(), below, to determine the IP address of your local machine. What could be more fun? I could think of a few things, but they don't pertain to socket programming. Anyway, here's the breakdown: #include int gethostname(char *hostname, size_t size); The arguments are simple: hostname is a pointer to an array of chars that will contain the hostname upon the function's return, and size is the length in bytes of the hostname array. The function returns 0 on successful completion, and -1 on error, setting errno as usual. 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. 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 code23: /* ** server.c -- a stream socket server demo */ #include #include #include #include #include #include #include #include #include #include #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", 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"); 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..."); 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", 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. 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 source24: /* ** client.c -- a stream socket client demo */ #include #include #include #include #include #include #include #include #include #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"); 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", 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"); return 2; } inet_ntop(p->ai_family, get_in_addr((struct sockaddr *)p->ai_addr), s, sizeof s); printf("client: connecting to %s", 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'",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. 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. Because datagram sockets are connectionless and just fire packets off into the ether with callous disregard for success, we are going to tell the client and server to use specifically IPv6. This way we avoid the situation where the server is listening on IPv6 and the client sends on IPv4; the data simply would not be received. (In our connected TCP stream sockets world, we might still have the mismatch, but the error on connect() for one address family would cause us to retry for the other.) Here is the source for listener.c25: /* ** listener.c -- a datagram sockets "server" demo */ #include #include #include #include #include #include #include #include #include #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_INET6; // set to AF_INET to use 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", 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"); return 2; } freeaddrinfo(servinfo); printf("listener: waiting to recvfrom..."); 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", inet_ntop(their_addr.ss_family, get_in_addr((struct sockaddr *)&their_addr), s, sizeof s)); printf("listener: packet is %d bytes long", numbytes); buf[numbytes] = '\0'; printf("listener: packet contains \"%s\"", 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.c26: /* ** talker.c -- a datagram "client" demo */ #include #include #include #include #include #include #include #include #include #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"); exit(1); } memset(&hints, 0, sizeof hints); hints.ai_family = AF_INET6; // set to AF_INET to use IPv4 hints.ai_socktype = SOCK_DGRAM; if ((rv = getaddrinfo(argv[1], SERVERPORT, &hints, &servinfo)) != 0) { fprintf(stderr, "getaddrinfo: %s", 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"); 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", 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(). Slightly Advanced Techniques These aren't really advanced, but they're getting out of the more basic levels we've already covered. In fact, if you've gotten this far, you should consider yourself fairly accomplished in the basics of Unix network programming! Congratulations! So here we go into the brave new world of some of the more esoteric things you might want to learn about sockets. Have at it! Blocking Blocking. You've heard about it—now what the heck is it? In a nutshell, "block" is techie jargon for "sleep". You probably noticed that when you run listener, above, it just sits there until a packet arrives. What happened is that it called recvfrom(), there was no data, and so recvfrom() is said to "block" (that is, sleep there) until some data arrives. Lots of functions block. accept() blocks. All the recv() functions block. The reason they can do this is because they're allowed to. When you first create the socket descriptor with socket(), the kernel sets it to blocking. If you don't want a socket to be blocking, you have to make a call to fcntl(): #include #include . . . sockfd = socket(PF_INET, SOCK_STREAM, 0); fcntl(sockfd, F_SETFL, O_NONBLOCK); . . . By setting a socket to non-blocking, you can effectively "poll" the socket for information. If you try to read from a non-blocking socket and there's no data there, it's not allowed to block—it will return -1 and errno will be set to EAGAIN or EWOULDBLOCK. (Wait—it can return EAGAIN or EWOULDBLOCK? Which do you check for? The specification doesn't actually specify which your system will return, so for portability, check them both.) Generally speaking, however, this type of polling is a bad idea. If you put your program in a busy-wait looking for data on the socket, you'll suck up CPU time like it was going out of style. A more elegant solution for checking to see if there's data waiting to be read comes in the following section on poll(). poll()—Synchronous I/O Multiplexing What you really want to be able to do is somehow monitor a bunch of sockets at once and then handle the ones that have data ready. This way you don't have to continously poll all those sockets to see which are ready to read. A word of warning: poll() is horribly slow when it comes to giant numbers of connections. In those circumstances, you'll get better performance out of an event library such as libevent27 that attempts to use the fastest possible method availabile on your system. So how can you avoid polling? Not slightly ironically, you can avoid polling by using the poll() system call. In a nutshell, we're going to ask the operating system to do all the dirty work for us, and just let us know when some data is ready to read on which sockets. In the meantime, our process can go to sleep, saving system resources. The general gameplan is to keep an array of struct pollfds with information about which socket descriptors we want to monitor, and what kind of events we want to monitor for. The OS will block on the poll() call until one of those events occurs (e.g. "socket ready to read!") or until a user-specified timeout occurs. Usefully, a listen()ing socket will return "ready to read" when a new incoming connection is ready to be accept()ed. That's enough banter. How do we use this? #include int poll(struct pollfd fds[], nfds_t nfds, int timeout); fds is our array of information (which sockets to monitor for what), nfds is the count of elements in the array, and timeout is a timeout in milliseconds. It returns the number of elements in the array that have had an event occur. Let's have a look at that struct: struct pollfd { int fd; // the socket descriptor short events; // bitmap of events we're interested in short revents; // when poll() returns, bitmap of events that occurred }; So we're going to have an array of those, and we'll see the fd field for each element to a socket descriptor we're interested in monitoring. And then we'll set the events field to indicate the type of events we're interested in. The events field is the bitwise-OR of the following: POLLIN Alert me when data is ready to recv() on this socket. POLLOUT Alert me when I can send() data to this socket without blocking. Once you have your array of struct pollfds in order, then you can pass it to poll(), also passing the size of the array, as well as a timeout value in milliseconds. (You can specify a negative timeout to wait forever.) After poll() returns, you can check the revents field to see if POLLIN or POLLOUT is set, indicating that event occurred. (There's actually more that you can do with the poll() call. See the poll() man page, below, for more details.) Here's an example28 where we'll wait 2.5 seconds for data to be ready to read from standard input, i.e. when you hit RETURN: #include #include int main(void) { struct pollfd pfds[1]; // More if you want to monitor more pfds[0].fd = 0; // Standard input pfds[0].events = POLLIN; // Tell me when ready to read // If you needed to monitor other things, as well: //pfds[1].fd = some_socket; // Some socket descriptor //pfds[1].events = POLLIN; // Tell me when ready to read printf("Hit RETURN or wait 2.5 seconds for timeout"); int num_events = poll(pfds, 1, 2500); // 2.5 second timeout if (num_events == 0) { printf("Poll timed out!"); } else { int pollin_happened = pfds[0].revents & POLLIN; if (pollin_happened) { printf("File descriptor %d is ready to read", pfds[0].fd); } else { printf("Unexpected event occurred: %d", pfds[0].revents); } } return 0; } Notice again that poll() returns the number of elements in the pfds array for which events have occurred. It doesn't tell you which elements in the array (you still have to scan for that), but it does tell you how many entries have a non-zero revents field (so you can stop scanning after you find that many). A couple questions might come up here: how to add new file descriptors to the set I pass to poll()? For this, simply make sure you have enough space in the array for all you need, or realloc() more space as needed. What about deleting items from the set? For this, you can copy the last element in the array over-top the one you're deleting. And then pass in one fewer as the count to poll(). Another option is that you can set any fd field to a negative number and poll() will ignore it. How can we put it all together into a chat server that you can telnet to? What we'll do is start a listener socket, and add it to the set of file descriptors to poll(). (It will show ready-to-read when there's an incoming connection.) Then we'll add new connections to our struct pollfd array. And we'll grow it dynamically if we run out of space. When a connection is closed, we'll remove it from the array. And when a connection is ready-to-read, we'll read the data from it and send that data to all the other connections so they can see what the other users typed. So give this poll server29 a try. Run it in one window, then telnet localhost 9034 from a number of other terminal windows. You should be able to see what you type in one window in the other ones (after you hit RETURN). Not only that, but if you hit CTRL-] and type quit to exit telnet, the server should detect the disconnection and remove you from the array of file descriptors. /* ** pollserver.c -- a cheezy multiperson chat server */ #include #include #include #include #include #include #include #include #define PORT "9034" // Port we're listening on // 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); } // Return a listening socket int get_listener_socket(void) { int listener; // Listening socket descriptor int yes=1; // For setsockopt() SO_REUSEADDR, below int rv; struct addrinfo hints, *ai, *p; // Get us a socket and bind it memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; hints.ai_socktype = SOCK_STREAM; hints.ai_flags = AI_PASSIVE; if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) { fprintf(stderr, "selectserver: %s", gai_strerror(rv)); exit(1); } for(p = ai; p != NULL; p = p->ai_next) { listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol); if (listener < 0) { continue; } // Lose the pesky "address already in use" error message setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)); if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) { close(listener); continue; } break; } freeaddrinfo(ai); // All done with this // If we got here, it means we didn't get bound if (p == NULL) { return -1; } // Listen if (listen(listener, 10) == -1) { return -1; } return listener; } // Add a new file descriptor to the set void add_to_pfds(struct pollfd *pfds[], int newfd, int *fd_count, int *fd_size) { // If we don't have room, add more space in the pfds array if (*fd_count == *fd_size) { *fd_size *= 2; // Double it *pfds = realloc(*pfds, sizeof(**pfds) * (*fd_size)); } (*pfds)[*fd_count].fd = newfd; (*pfds)[*fd_count].events = POLLIN; // Check ready-to-read (*fd_count)++; } // Remove an index from the set void del_from_pfds(struct pollfd pfds[], int i, int *fd_count) { // Copy the one from the end over this one pfds[i] = pfds[*fd_count-1]; (*fd_count)--; } // Main int main(void) { int listener; // Listening socket descriptor int newfd; // Newly accept()ed socket descriptor struct sockaddr_storage remoteaddr; // Client address socklen_t addrlen; char buf[256]; // Buffer for client data char remoteIP[INET6_ADDRSTRLEN]; // Start off with room for 5 connections // (We'll realloc as necessary) int fd_count = 0; int fd_size = 5; struct pollfd *pfds = malloc(sizeof *pfds * fd_size); // Set up and get a listening socket listener = get_listener_socket(); if (listener == -1) { fprintf(stderr, "error getting listening socket"); exit(1); } // Add the listener to set pfds[0].fd = listener; pfds[0].events = POLLIN; // Report ready to read on incoming connection fd_count = 1; // For the listener // Main loop for(;;) { int poll_count = poll(pfds, fd_count, -1); if (poll_count == -1) { perror("poll"); exit(1); } // Run through the existing connections looking for data to read for(int i = 0; i < fd_count; i++) { // Check if someone's ready to read if (pfds[i].revents & POLLIN) { // We got one!! if (pfds[i].fd == listener) { // If listener is ready to read, handle new connection addrlen = sizeof remoteaddr; newfd = accept(listener, (struct sockaddr *)&remoteaddr, &addrlen); if (newfd == -1) { perror("accept"); } else { add_to_pfds(&pfds, newfd, &fd_count, &fd_size); printf("pollserver: new connection from %s on " "socket %d", inet_ntop(remoteaddr.ss_family, get_in_addr((struct sockaddr*)&remoteaddr), remoteIP, INET6_ADDRSTRLEN), newfd); } } else { // If not the listener, we're just a regular client int nbytes = recv(pfds[i].fd, buf, sizeof buf, 0); int sender_fd = pfds[i].fd; if (nbytes sa_family == AF_INET) { return &(((struct sockaddr_in*)sa)->sin_addr); } return &(((struct sockaddr_in6*)sa)->sin6_addr); } int main(void) { fd_set master; // master file descriptor list fd_set read_fds; // temp file descriptor list for select() int fdmax; // maximum file descriptor number int listener; // listening socket descriptor int newfd; // newly accept()ed socket descriptor struct sockaddr_storage remoteaddr; // client address socklen_t addrlen; char buf[256]; // buffer for client data int nbytes; char remoteIP[INET6_ADDRSTRLEN]; int yes=1; // for setsockopt() SO_REUSEADDR, below int i, j, rv; struct addrinfo hints, *ai, *p; FD_ZERO(&master); // clear the master and temp sets FD_ZERO(&read_fds); // get us a socket and bind it memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; hints.ai_socktype = SOCK_STREAM; hints.ai_flags = AI_PASSIVE; if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) { fprintf(stderr, "selectserver: %s", gai_strerror(rv)); exit(1); } for(p = ai; p != NULL; p = p->ai_next) { listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol); if (listener < 0) { continue; } // lose the pesky "address already in use" error message setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)); if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) { close(listener); continue; } break; } // if we got here, it means we didn't get bound if (p == NULL) { fprintf(stderr, "selectserver: failed to bind"); exit(2); } freeaddrinfo(ai); // all done with this // listen if (listen(listener, 10) == -1) { perror("listen"); exit(3); } // add the listener to the master set FD_SET(listener, &master); // keep track of the biggest file descriptor fdmax = listener; // so far, it's this one // main loop for(;;) { read_fds = master; // copy it if (select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) { perror("select"); exit(4); } // run through the existing connections looking for data to read for(i = 0; i fdmax) { // keep track of the max fdmax = newfd; } printf("selectserver: new connection from %s on " "socket %d", inet_ntop(remoteaddr.ss_family, get_in_addr((struct sockaddr*)&remoteaddr), remoteIP, INET6_ADDRSTRLEN), newfd); } } else { // handle data from a client if ((nbytes = recv(i, buf, sizeof buf, 0)) /tmp/client.out") close() the connection Meanwhile, the server is handling the data and executing it: accept() the connection from the client recv(str) the command string close() the connection system(str) to run the command Beware! Having the server execute what the client says is like giving remote shell access and people can do things to your

account when they connect to the server. For instance, in the above example, what if the client sends "rm -rf ~"? It deletes everything in your account! So you get wise, and you prevent the client from using any except for a couple utilities that you know are safe, like the foobar utility: if (!strncmp(str, "foobar", 6)) { sprintf(sysstr, "%s > /tmp/server.out", str); system(sysstr); } But you're still unsafe, unfortunately: what if the client enters "foobar; rm -rf ~"? The safest thing to do is to write a little routine that puts an escape ("\") character in front of all non-alphanumeric characters (including spaces, if appropriate) in the arguments for the command. As you can see, security is a pretty big issue when the server starts executing things the client sends. I'm sending a slew of data, but when I recv(), it only receives 536 bytes or 1460 bytes at a time. But if I run it on my local machine, it receives all the data at the same time. What's going on? You're hitting the MTU—the maximum size the physical medium can handle. On the local machine, you're using the loopback device which can handle 8K or more no problem. But on Ethernet, which can only handle 1500 bytes with a header, you hit that limit. Over a modem, with 576 MTU (again, with header), you hit the even lower limit. You have to make sure all the data is being sent, first of all. (See the sendall() function implementation for details.) Once you're sure of that, then you need to call recv() in a loop until all your data is read. Read the section Son of Data Encapsulation for details on receiving complete packets of data using multiple calls to recv(). I'm on a Windows box and I don't have the fork() system call or any kind of struct sigaction. What to do? If they're anywhere, they'll be in POSIX libraries that may have shipped with your compiler. Since I don't have a Windows box, I really can't tell you the answer, but I seem to remember that Microsoft has a POSIX compatibility layer and that's where fork() would be. (And maybe even sigaction.) Search the help that came with VC++ for "fork" or "POSIX" and see if it gives you any clues. If that doesn't work at all, ditch the fork()/sigaction stuff and replace it with the Win32 equivalent: CreateProcess(). I don't know how to use CreateProcess()—it takes a bazillion arguments, but it should be covered in the docs that came with VC++. I'm behind a firewall—how do I let people outside the firewall know my IP address so they can connect to my machine? Unfortunately, the purpose of a firewall is to prevent people outside the firewall from connecting to machines inside the firewall, so allowing them to do so is basically considered a breach of security. This isn't to say that all is lost. For one thing, you can still often connect() through the firewall if it's doing some kind of masquerading or NAT or something like that. Just design your programs so that you're always the one initiating the connection, and you'll be fine. If that's not satisfactory, you can ask your sysadmins to poke a hole in the firewall so that people can connect to you. The firewall can forward to you either through it's NAT software, or through a proxy or something like that. Be aware that a hole in the firewall is nothing to be taken lightly. You have to make sure you don't give bad people access to the internal network; if you're a beginner, it's a lot harder to make software secure than you might imagine. Don't make your sysadmin mad at me. ;-) How do I write a packet sniffer? How do I put my Ethernet interface into promiscuous mode? For those not in the know, when a network card is in "promiscuous mode", it will forward ALL packets to the operating system, not just those that were addressed to this particular machine. (We're talking Ethernet-layer addresses here, not IP addresses—but since ethernet is lower-layer than IP, all IP addresses are effectively forwarded as well. See the section Low Level Nonsense and Network Theory for more info.) This is the basis for how a packet sniffer works. It puts the interface into promiscuous mode, then the OS gets every single packet that goes by on the wire. You'll have a socket of some type that you can read this data from. Unfortunately, the answer to the question varies depending on the platform, but if you Google for, for instance, "windows promiscuous ioctl" you'll probably get somewhere. For Linux, there's what looks like a useful Stack Overflow thread46, as well. How can I set a custom timeout value for a TCP or UDP socket? It depends on your system. You might search the net for SO_RCVTIMEO and SO_SNDTIMEO (for use with setsockopt()) to see if your system supports such functionality. The Linux man page suggests using alarm() or setitimer() as a substitute. How can I tell which ports are available to use? Is there a list of "official" port numbers? Usually this isn't an issue. If you're writing, say, a web server, then it's a good idea to use the well-known port 80 for your software. If you're writing just your own specialized server, then choose a port at random (but greater than 1023) and give it a try. If the port is already in use, you'll get an "Address already in use" error when you try to bind(). Choose another port. (It's a good idea to allow the user of your software to specify an alternate port either with a config file or a command line switch.) There is a list of official port numbers47 maintained by the Internet Assigned Numbers Authority (IANA). Just because something (over 1023) is in that list doesn't mean you can't use the port. For instance, Id Software's DOOM uses the same port as "mdqs", whatever that is. All that matters is that no one else on the same machine is using that port when you want to use it. Man Pages In the Unix world, there are a lot of manuals. They have little sections that describe individual functions that you have at your disposal. Of course, manual would be too much of a thing to type. I mean, no one in the Unix world, including myself, likes to type that much. Indeed I could go on and on at great length about how much I prefer to be terse but instead I shall be brief and not bore you with long-winded diatribes about how utterly amazingly brief I prefer to be in virtually all circumstances in their entirety. [Applause] Thank you. What I am getting at is that these pages are called "man pages" in the Unix world, and I have included my own personal truncated variant here for your reading enjoyment. The thing is, many of these functions are way more general purpose than I'm letting on, but I'm only going to present the parts that are relevant for Internet Sockets Programming. But wait! That's not all that's wrong with my man pages: They are incomplete and only show the basics from the guide. There are many more man pages than this in the real world. They are different than the ones on your system. The header files might be different for certain functions on your system. The function parameters might be different for certain functions on your system. If you want the real information, check your local Unix man pages by typing man whatever, where "whatever" is something that you're incredibly interested in, such as "accept". (I'm sure Microsoft Visual Studio has something similar in their help section. But "man" is better because it is one byte more concise than "help". Unix wins again!) So, if these are so flawed, why even include them at all in the Guide? Well, there are a few reasons, but the best are that (a) these versions are geared specifically toward network programming and are easier to digest than the real ones, and (b) these versions contain examples! Oh! And speaking of the examples, I don't tend to put in all the error checking because it really increases the length of the code. But you should absolutely do error checking pretty much any time you make any of the system calls unless you're totally 100% sure it's not going to fail, and you should probably do it even then! accept() Accept an incoming connection on a listening socket Synopsis #include #include int accept(int s, struct sockaddr *addr, socklen_t *addrlen); Description Once you've gone through the trouble of getting a SOCK_STREAM socket and setting it up for incoming connections with listen(), then you call accept() to actually get yourself a new socket descriptor to use for subsequent communication with the newly connected client. The old socket that you are using for listening is still there, and will be used for further accept() calls as they come in. s The listening socket descriptor. addr This is filled in with the address of the site that's connecting to you. addrlen This is filled in with the sizeof() the structure returned in the addr parameter. You can safely ignore it if you assume you're getting a struct sockaddr_in back, which you know you are, because that's the type you passed in for addr. accept() will normally block, and you can use select() to peek on the listening socket descriptor ahead of time to see if it's "ready to read". If so, then there's a new connection waiting to be accept()ed! Yay! Alternatively, you could set the O_NONBLOCK flag on the listening socket using fcntl(), and then it will never block, choosing instead to return -1 with errno set to EWOULDBLOCK. The socket descriptor returned by accept() is a bona fide socket descriptor, open and connected to the remote host. You have to close() it when you're done with it. Return Value accept() returns the newly connected socket descriptor, or -1 on error, with errno set appropriately. Example struct sockaddr_storage their_addr; socklen_t addr_size; struct addrinfo hints, *res; int sockfd, new_fd; // first, load up address structs with getaddrinfo(): memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever hints.ai_socktype = SOCK_STREAM; hints.ai_flags = AI_PASSIVE; // fill in my IP for me getaddrinfo(NULL, MYPORT, &hints, &res); // make a socket, bind it, and listen on it: sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); bind(sockfd, res->ai_addr, res->ai_addrlen); listen(sockfd, BACKLOG); // now accept an incoming connection: addr_size = sizeof their_addr; new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &addr_size); // ready to communicate on socket descriptor new_fd! See Also socket(), getaddrinfo(), listen(), struct sockaddr_in bind() Associate a socket with an IP address and port number Synopsis #include #include int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen); Description When a remote machine wants to connect to your server program, it needs two pieces of information: the IP address and the port number. The bind() call allows you to do just that. First, you call getaddrinfo() to load up a struct sockaddr with the destination address and port information. Then you call socket() to get a socket descriptor, and then you pass the socket and address into bind(), and the IP address and port are magically (using actual magic) bound to the socket! If you don't know your IP address, or you know you only have one IP address on the machine, or you don't care which of the machine's IP addresses is used, you can simply pass the AI_PASSIVE flag in the hints parameter to getaddrinfo(). What this does is fill in the IP address part of the struct sockaddr with a special value that tells bind() that it should automatically fill in this host's IP address. What what? What special value is loaded into the struct sockaddr's IP address to cause it to auto-fill the address with the current host? I'll tell you, but keep in mind this is only if you're filling out the struct sockaddr by hand; if not, use the results from getaddrinfo(), as per above. In IPv4, the sin_addr.s_addr field of the struct sockaddr_in structure is set to INADDR_ANY. In IPv6, the sin6_addr field of the struct sockaddr_in6 structure is assigned into from the global variable in6addr_any. Or, if you're declaring a new struct in6_addr, you can initialize it to IN6ADDR_ANY_INIT. Lastly, the addrlen parameter should be set to sizeof my_addr. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example // modern way of doing things with getaddrinfo() struct addrinfo hints, *res; int sockfd; // first, load up address structs with getaddrinfo(): memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever hints.ai_socktype = SOCK_STREAM; hints.ai_flags = AI_PASSIVE; // fill in my IP for me getaddrinfo(NULL, "3490", &hints, &res); // make a socket: // (you should actually walk the "res" linked list and error-check!) sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); // bind it to the port we passed in to getaddrinfo(): bind(sockfd, res->ai_addr, res->ai_addrlen); // example of packing a struct by hand, IPv4 struct sockaddr_in myaddr; int s; myaddr.sin_family = AF_INET; myaddr.sin_port = htons(3490); // you can specify an IP address: inet_pton(AF_INET, "63.161.169.137", &(myaddr.sin_addr)); // or you can let it automatically select one: myaddr.sin_addr.s_addr = INADDR_ANY; s = socket(PF_INET, SOCK_STREAM, 0); bind(s, (struct sockaddr*)&myaddr, sizeof myaddr); See Also getaddrinfo(), socket(), struct sockaddr_in, struct in_addr connect() Connect a socket to a server Synopsis #include #include int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen); Description Once you've built a socket descriptor with the socket() call, you can connect() that socket to a remote server using the well-named connect() system call. All you need to do is pass it the socket descriptor and the address of the server you're interested in getting to know better. (Oh, and the length of the address, which is commonly passed to functions like this.) Usually this information comes along as the result of a call to getaddrinfo(), but you can fill out your own struct sockaddr if you want to. If you haven't yet called bind() on the socket descriptor, it is automatically bound to your IP address and a random local port. This is usually just fine with you if you're not a server, since you really don't care what your local port is; you only care what the remote port is so you can put it in the serv_addr parameter. You can call bind() if you really want your client socket to be on a specific IP address and port, but this is pretty rare. Once the socket is connect()ed, you're free to send() and recv() data on it to your heart's content. Special note: if you connect() a UDP SOCK_DGRAM socket to a remote host, you can use send() and recv() as well as sendto() and recvfrom(). If you want. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example // connect to www.example.com port 80 (http) struct addrinfo hints, *res; int sockfd; // first, load up address structs with getaddrinfo(): memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever hints.ai_socktype = SOCK_STREAM; // we could put "80" instead on "http" on the next line: getaddrinfo("www.example.com", "http", &hints, &res); // make a socket: sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); // connect it to the address and port we passed in to getaddrinfo(): connect(sockfd, res->ai_addr, res->ai_addrlen); See Also socket(), bind() close() Close a socket descriptor Synopsis #include int close(int s); Description After you've finished using the socket for whatever demented scheme you have concocted and you don't want to send() or recv() or, indeed, do anything else at all with the socket, you can close() it, and it'll be freed up, never to be used again. The remote side can tell if this happens one of two ways. One: if the remote side calls recv(), it will return 0. Two: if the remote side calls send(), it'll receive a signal SIGPIPE and send() will return -1 and errno will be set to EPIPE. Windows users: the function you need to use is called closesocket(), not close(). If you try to use close() on a socket descriptor, it's possible Windows will get angry... And you wouldn't like it when it's angry. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example s = socket(PF_INET, SOCK_DGRAM, 0); . . . // a whole lotta stuff...*BRRRONNNN!* . . . close(s); // not much to it, really. See Also socket(), shutdown() getaddrinfo(), freeaddrinfo(), gai_strerror() Get information about a host name and/or service and load up a struct sockaddr with the result Synopsis #include #include #include int getaddrinfo(const char *nodename, const char *servname, const struct addrinfo *hints, struct addrinfo **res); void freeaddrinfo(struct addrinfo *ai); const char *gai_strerror(int ecode); struct addrinfo { int ai_flags; // AI_PASSIVE, AI_CANONNAME, ... int ai_family; // AF_xxx int ai_socktype; // SOCK_xxx int ai_protocol; // 0 (auto) or IPPROTO_TCP, IPPROTO_UDP socklen_t ai_addrlen; // length of ai_addr char *ai_canonname; // canonical name for nodename struct sockaddr *ai_addr; // binary address struct addrinfo *ai_next; // next structure in linked list }; Description getaddrinfo() is an excellent function that will return information on a particular host name (such as its IP address) and load up a struct sockaddr for you, taking care of the gritty details (like if it's IPv4 or IPv6). It replaces the old functions gethostbyname() and getservbyname(). The description, below, contains a lot of information that might be a little daunting, but actual usage is pretty simple. It might be worth it to check out the examples first. The host name that you're interested in goes in the nodename parameter. The address can be either a host name, like "www.example.com", or an IPv4 or IPv6 address (passed as a string). This parameter can also be NULL if you're using the AI_PASSIVE flag (see below). The servname parameter is basically the port number. It can be a port number (passed as a string, like "80"), or it can be a service name, like "http" or "tftp" or "smtp" or "pop", etc. Well-known service names can be found in the IANA Port List48 or in your /etc/services file. Lastly, for input parameters, we have hints. This is really where you get to define what the getaddrinfo() function is going to do. Zero the whole structure before use with memset(). Let's take a look at the fields you need to set up before use. The ai_flags can be set to a variety of things, but here are a couple important ones. (Multiple flags can be specified by bitwise-ORing them together with the | operator). Check your man page for the complete list of flags. AI_CANONNAME causes the ai_canonname of the result to the filled out with the host's canonical (real) name. AI_PASSIVE causes the result's IP address to be filled out with INADDR_ANY (IPv4) or in6addr_any (IPv6); this causes a subsequent call to bind() to auto-fill the IP address of the struct sockaddr with the address of the current host. That's excellent for setting up a server when you don't want to hardcode the address. If you do use the AI_PASSIVE, flag, then you can pass NULL in the nodename (since bind() will fill it in for you later). Continuing on with the input paramters, you'll likely want to set ai_family to AF_UNSPEC which tells getaddrinfo() to look for both IPv4 and IPv6 addresses. You can also restrict yourself to one or the other with AF_INET or AF_INET6. Next, the socktype field should be set to SOCK_STREAM or SOCK_DGRAM, depending on which type of socket you want. Finally, just leave ai_protocol at 0 to automatically choose your protocol type. Now, after you get all that stuff in there, you can finally make the call to getaddrinfo()! Of course, this is where the fun begins. The res will now point to a linked list of struct addrinfos, and you can go through this list to get all the addresses that match what you passed in with the hints. Now, it's possible to get some addresses that don't work for one reason or another, so what the Linux man page does is loops through the list doing a call to socket() and connect() (or bind() if you're setting up a server with the AI_PASSIVE flag) until it succeeds. Finally, when you're done with the linked list, you need to call freeaddrinfo() to free up the memory (or it will be leaked, and Some People will get upset). Return Value Returns zero on success, or nonzero on error. If it returns nonzero, you can use the function gai_strerror() to get a printable version of the error code in the return value. Example // code for a client connecting to a server // namely a stream socket to www.example.com on port 80 (http) // either IPv4 or IPv6 int sockfd; struct addrinfo hints, *servinfo, *p; int rv; memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6 hints.ai_socktype = SOCK_STREAM; if ((rv = getaddrinfo("www.example.com", "http", &hints, &servinfo)) != 0) { fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv)); exit(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("socket"); continue; } if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) { perror("connect"); close(sockfd); continue; } break; // if we get here, we must have connected successfully } if (p == NULL) { // looped off the end of the list with no connection fprintf(stderr, "failed to connect\n"); exit(2); } freeaddrinfo(servinfo); // all done with this structure // code for a server waiting for connections // namely a stream socket on port 3490, on this host's IP // either IPv4 or IPv6. int sockfd; struct addrinfo hints, *servinfo, *p; int rv; memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use AF_INET6 to force IPv6 hints.ai_socktype = SOCK_STREAM; hints.ai_flags = AI_PASSIVE; // use my IP address if ((rv = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) { fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv)); exit(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("socket"); continue; } if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) { close(sockfd); perror("bind"); continue; } break; // if we get here, we must have connected successfully } if (p == NULL) { // looped off the end of the list with no successful bind fprintf(stderr, "failed to bind socket\n"); exit(2); } freeaddrinfo(servinfo); // all done with this structure See Also gethostbyname(), getnameinfo() gethostname() Returns the name of the system Synopsis #include int gethostname(char *name, size_t len); Description Your system has a name. They all do. This is a slightly more Unixy thing than the rest of the networky stuff we've been talking about, but it still has its uses. For instance, you can get your host name, and then call gethostbyname() to find out your IP address. The parameter name should point to a buffer that will hold the host name, and len is the size of that buffer in bytes. gethostname() won't overwrite the end of the buffer (it might return an error, or it might just stop writing), and it will NUL-terminate the string if there's room for it in the buffer. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example char hostname[128]; gethostname(hostname, sizeof hostname); printf("My hostname: %s\n", hostname); See Also gethostbyname() gethostbyname(), gethostbyaddr() Get an IP address for a hostname, or vice-versa Synopsis #include #include struct hostent *gethostbyname(const char *name); // DEPRECATED! struct hostent *gethostbyaddr(const char *addr, int len, int type); Description PLEASE NOTE: these two functions are superseded by getaddrinfo() and getnameinfo()! In particular, gethostbyname() doesn't work well with IPv6. These functions map back and forth between host names and IP addresses. For instance, if you have "www.example.com", you can use gethostbyname() to get its IP address and store it in a struct in_addr. Conversely, if you have a struct in_addr or a struct in6_addr, you can use gethostbyaddr() to get the hostname back. gethostbyaddr() is IPv6 compatible, but you should use the newer shinier getnameinfo() instead. (If you have a string containing an IP address in dots-and-numbers format that you want to look up the hostname of, you'd be better off using getaddrinfo() with the AI_CANONNAME flag.) gethostbyname() takes a string like "www.yahoo.com", and returns a struct hostent which contains tons of information, including the IP address. (Other information is the official host name, a list of aliases, the address type, the length of the addresses, and the list of addresses—it's a general-purpose structure that's pretty easy to use for our specific purposes once you see how.) gethostbyaddr() takes a struct in_addr or struct in6_addr and brings you up a corresponding host name (if there is one), so it's sort of the reverse of gethostbyname(). As for parameters, even though addr is a char*, you actually want to pass in a pointer to a struct in_addr. len should be sizeof(struct in_addr), and type should be AF_INET. So what is this struct hostent that gets returned? It has a number of fields that contain information about the host in question. char *h_name The real canonical host name. char **h_aliases A list of aliases that can be accessed with arrays—the last element is NULL int h_addrtype The result's address type, which really should be AF_INET for our purposes. int length The length of the addresses in bytes, which is 4 for IP (version 4) addresses. char **h_addr_list A list of IP addresses for this host. Although this is a char**, it's really an array of struct in_addr*s in disguise. The last array element is NULL. h_addr A commonly defined alias for h_addr_list[0]. If you just want any old IP address for this host (yeah, they can have more than one) just use this field. Return Value Returns a pointer to a resultant struct hostent on success, or NULL on error. Instead of the normal perror() and all that stuff you'd normally use for error reporting, these functions have parallel results in the variable h_errno, which can be printed using the functions herror() or hstrerror(). These work just like the classic errno, perror(), and strerror() functions you're used to. Example // THIS IS A DEPRECATED METHOD OF GETTING HOST NAMES // use getaddrinfo() instead! #include #include #include #include #include #include #include int main(int argc, char *argv[]) { int i; struct hostent *he; struct in_addr **addr_list; if (argc != 2) { fprintf(stderr,"usage: ghbn hostname\n"); return 1; } if ((he = gethostbyname(argv[1])) == NULL) { // get the host info herror("gethostbyname"); return 2; } // print information about this host: printf("Official name is: %s\n", he->h_name); printf(" IP addresses: "); addr_list = (struct in_addr **)he->h_addr_list; for(i = 0; addr_list[i] != NULL; i++) { printf("%s ", inet_ntoa(*addr_list[i])); } printf("\n"); return 0; } // THIS HAS BEEN SUPERCEDED // use getnameinfo() instead! struct hostent *he; struct in_addr ipv4addr; struct in6_addr ipv6addr; inet_pton(AF_INET, "192.0.2.34", &ipv4addr); he = gethostbyaddr(&ipv4addr, sizeof ipv4addr, AF_INET); printf("Host name: %s\n", he->h_name); inet_pton(AF_INET6, "2001:db8:63b3:1::beef", &ipv6addr); he = gethostbyaddr(&ipv6addr, sizeof ipv6addr, AF_INET6); printf("Host name: %s\n", he->h_name); See Also getaddrinfo(), getnameinfo(), gethostname(), errno, perror(), strerror(), struct in_addr getnameinfo() Look up the host name and service name information for a given struct sockaddr. Synopsis #include #include int getnameinfo(const struct sockaddr *sa, socklen_t salen, char *host, size_t hostlen, char *serv, size_t servlen, int flags); Description This function is the opposite of getaddrinfo(), that is, this function takes an already loaded struct sockaddr and does a name and service name lookup on it. It replaces the old gethostbyaddr() and getservbyport() functions. You have to pass in a pointer to a struct sockaddr (which in actuality is probably a struct sockaddr_in or struct sockaddr_in6 that you've cast) in the sa parameter, and the length of that struct in the salen. The resultant host name and service name will be written to the area pointed to by the host and serv parameters. Of course, you have to specify the max lengths of these buffers in hostlen and servlen. Finally, there are several flags you can pass, but here a couple good ones. NI_NOFQDN will cause the host to only contain the host name, not the whole domain name. NI_NAMEREQD will cause the function to fail if the name cannot be found with a DNS lookup (if you don't specify this flag and the name can't be found, getnameinfo() will put a string version of the IP address in host instead). As always, check your local man pages for the full scoop. Return Value Returns zero on success, or non-zero on error. If the return value is non-zero, it can be passed to gai_strerror() to get a human-readable string. See getaddrinfo for more information. Example struct sockaddr_in6 sa; // could be IPv4 if you want char host[1024]; char service[20]; // pretend sa is full of good information about the host and port... getnameinfo(&sa, sizeof sa, host, sizeof host, service, sizeof service, 0); printf(" host: %s\n", host); // e.g. "www.example.com" printf("service: %s\n", service); // e.g. "http" See Also getaddrinfo(), gethostbyaddr() getpeername() Return address info about the remote side of the connection Synopsis #include int getpeername(int s, struct sockaddr *addr, socklen_t *len); Description Once you have either accept()ed a remote connection, or connect()ed to a server, you now have what is known as a peer. Your peer is simply the computer you're connected to, identified by an IP address and a port. So... getpeername() simply returns a struct sockaddr_in filled with information about the machine you're connected to. Why is it called a "name"? Well, there are a lot of different kinds of sockets, not just Internet Sockets like we're using in this guide, and so "name" was a nice generic term that covered all cases. In our case, though, the peer's "name" is it's IP address and port. Although the function returns the size of the resultant address in len, you must preload len with the size of addr. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example // assume s is a connected socket socklen_t len; struct sockaddr_storage addr; char ipstr[INET6_ADDRSTRLEN]; int port; len = sizeof addr; getpeername(s, (struct sockaddr*)&addr, &len); // deal with both IPv4 and IPv6: if (addr.ss_family == AF_INET) { struct sockaddr_in *s = (struct sockaddr_in *)&addr; port = ntohs(s->sin_port); inet_ntop(AF_INET, &s->sin_addr, ipstr, sizeof ipstr); } else { // AF_INET6 struct sockaddr_in6 *s = (struct sockaddr_in6 *)&addr; port = ntohs(s->sin6_port); inet_ntop(AF_INET6, &s->sin6_addr, ipstr, sizeof ipstr); } printf("Peer IP address: %s\n", ipstr); printf("Peer port : %d\n", port); See Also gethostname(), gethostbyname(), gethostbyaddr() errno Holds the error code for the last system call Synopsis #include int errno; Description This is the variable that holds error information for a lot of system calls. If you'll recall, things like socket() and listen() return -1 on error, and they set the exact value of errno to let you know specifically which error occurred. The header file errno.h lists a bunch of constant symbolic names for errors, such as EADDRINUSE, EPIPE, ECONNREFUSED, etc. Your local man pages will tell you what codes can be returned as an error, and you can use these at run time to handle different errors in different ways. Or, more commonly, you can call perror() or strerror() to get a human-readable version of the error. One thing to note, for you multithreading enthusiasts, is that on most systems errno is defined in a threadsafe manner. (That is, it's not actually a global variable, but it behaves just like a global variable would in a single-threaded environment.) Return Value The value of the variable is the latest error to have transpired, which might be the code for "success" if the last action succeeded. Example s = socket(PF_INET, SOCK_STREAM, 0); if (s == -1) { perror("socket"); // or use strerror() } tryagain: if (select(n, &readfds, NULL, NULL) == -1) { // an error has occurred!! // if we were only interrupted, just restart the select() call: if (errno == EINTR) goto tryagain; // AAAA! goto!!! // otherwise it's a more serious error: perror("select"); exit(1); } See Also perror(), strerror() fcntl() Control socket descriptors Synopsis #include #include int fcntl(int s, int cmd, long arg); Description This function is typically used to do file locking and other file-oriented stuff, but it also has a couple socket-related functions that you might see or use from time to time. Parameter s is the socket descriptor you wish to operate on, cmd should be set to F_SETFL, and arg can be one of the following commands. (Like I said, there's more to fcntl() than I'm letting on here, but I'm trying to stay socket-oriented.) O_NONBLOCK Set the socket to be non-blocking. See the section on blocking for more details. O_ASYNC Set the socket to do asynchronous I/O. When data is ready to be recv()'d on the socket, the signal SIGIO will be raised. This is rare to see, and beyond the scope of the guide. And I think it's only available on certain systems. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Different uses of the fcntl() system call actually have different return values, but I haven't covered them here because they're not socket-related. See your local fcntl() man page for more information. Example int s = socket(PF_INET, SOCK_STREAM, 0); fcntl(s, F_SETFL, O_NONBLOCK); // set to non-blocking fcntl(s, F_SETFL, O_ASYNC); // set to asynchronous I/O See Also Blocking, send() htons(), htonl(), ntohs(), ntohl() Convert multi-byte integer types from host byte order to network byte order Synopsis #include uint32_t htonl(uint32_t hostlong); uint16_t htons(uint16_t hostshort); uint32_t ntohl(uint32_t netlong); uint16_t ntohs(uint16_t netshort); Description Just to make you really unhappy, different computers use different byte orderings internally for their multibyte integers (i.e. any integer that's larger than a char). The upshot of this is that if you send() a two-byte short int from an Intel box to a Mac (before they became Intel boxes, too, I mean), what one computer thinks is the number 1, the other will think is the number 256, and vice-versa. The way to get around this problem is for everyone to put aside their differences and agree that Motorola and IBM had it right, and Intel did it the weird way, and so we all convert our byte orderings to "big-endian" before sending them out. Since Intel is a "little-endian" machine, it's far more politically correct to call our preferred byte ordering "Network Byte Order". So these functions convert from your native byte order to network byte order and back again. (This means on Intel these functions swap all the bytes around, and on PowerPC they do nothing because the bytes are already in Network Byte Order. But you should always use them in your code anyway, since someone might want to build it on an Intel machine and still have things work properly.) Note that the types involved are 32-bit (4 byte, probably int) and 16-bit (2 byte, very likely short) numbers. 64-bit machines might have a htonll() for 64-bit ints, but I've not seen it. You'll just have to write your own. Anyway, the way these functions work is that you first decide if you're converting from host (your machine's) byte order or from network byte order. If "host", the first letter of the function you're going to call is "h". Otherwise it's "n" for "network". The middle of the function name is always "to" because you're converting from one "to" another, and the penultimate letter shows what you're converting to. The last letter is the size of the data, "s" for short, or "l" for long. Thus: htons() host to network short htonl() host to network long ntohs() network to host short ntohl() network to host long Return Value Each function returns the converted value. Example uint32_t some_long = 10; uint16_t some_short = 20; uint32_t network_byte_order; // convert and send network_byte_order = htonl(some_long); send(s, &network_byte_order, sizeof(uint32_t), 0); some_short == ntohs(htons(some_short)); // this expression is true inet_ntoa(), inet_aton(), inet_addr Convert IP addresses from a dots-and-number string to a struct in_addr and back Synopsis #include #include #include // ALL THESE ARE DEPRECATED! Use inet_pton() or inet_ntop() instead!! char *inet_ntoa(struct in_addr in); int inet_aton(const char *cp, struct in_addr *inp); in_addr_t inet_addr(const char *cp); Description These functions are deprecated because they don't handle IPv6! Use inet_ntop() or inet_pton() instead! They are included here because they can still be found in the wild. All of these functions convert from a struct in_addr (part of your struct sockaddr_in, most likely) to a string in dots-and-numbers format (e.g. "192.168.5.10") and vice-versa. If you have an IP address passed on the command line or something, this is the easiest way to get a struct in_addr to connect() to, or whatever. If you need more power, try some of the DNS functions like gethostbyname() or attempt a coup d'État in your local country. The function inet_ntoa() converts a network address in a struct in_addr to a dots-and-numbers format string. The "n" in "ntoa" stands for network, and the "a" stands for ASCII for historical reasons (so it's "Network To ASCII"—the "toa" suffix has an analogous friend in the C library called atoi() which converts an ASCII string to an integer). The function inet_aton() is the opposite, converting from a dots-and-numbers string into a in_addr_t (which is the type of the field s_addr in your struct in_addr). Finally, the function inet_addr() is an older function that does basically the same thing as inet_aton(). It's theoretically deprecated, but you'll see it a lot and the police won't come get you if you use it. Return Value inet_aton() returns non-zero if the address is a valid one, and it returns zero if the address is invalid. inet_ntoa() returns the dots-and-numbers string in a static buffer that is overwritten with each call to the function. inet_addr() returns the address as an in_addr_t, or -1 if there's an error. (That is the same result as if you tried to convert the string "255.255.255.255", which is a valid IP address. This is why inet_aton() is better.) Example struct sockaddr_in antelope; char *some_addr; inet_aton("10.0.0.1", &antelope.sin_addr); // store IP in antelope some_addr = inet_ntoa(antelope.sin_addr); // return the IP printf("%s\n", some_addr); // prints "10.0.0.1" // and this call is the same as the inet_aton() call, above: antelope.sin_addr.s_addr = inet_addr("10.0.0.1"); See Also inet_ntop(), inet_pton(), gethostbyname(), gethostbyaddr() inet_ntop(), inet_pton() Convert IP addresses to human-readable form and back. Synopsis #include const char *inet_ntop(int af, const void *src, char *dst, socklen_t size); int inet_pton(int af, const char *src, void *dst); Description These functions are for dealing with human-readable IP addresses and converting them to their binary representation for use with various functions and system calls. The "n" stands for "network", and "p" for "presentation". Or "text presentation". But you can think of it as "printable". "ntop" is "network to printable". See? Sometimes you don't want to look at a pile of binary numbers when looking at an IP address. You want it in a nice printable form, like 192.0.2.180, or 2001:db8:8714:3a90::12. In that case, inet_ntop() is for you. inet_ntop() takes the address family in the af parameter (either AF_INET or AF_INET6). The src parameter should be a pointer to either a struct in_addr or struct in6_addr containing the address you wish to convert to a string. Finally dst and size are the pointer to the destination string and the maximum length of that string. What should the maximum length of the dst string be? What is the maximum length for IPv4 and IPv6 addresses? Fortunately there are a couple of macros to help you out. The maximum lengths are: INET_ADDRSTRLEN and INET6_ADDRSTRLEN. Other times, you might have a string containing an IP address in readable form, and you want to pack it into a struct sockaddr_in or a struct sockaddr_in6. In that case, the opposite funcion inet_pton() is what you're after. inet_pton() also takes an address family (either AF_INET or AF_INET6) in the af parameter. The src parameter is a pointer to a string containing the IP address in printable form. Lastly the dst parameter points to where the result should be stored, which is probably a struct in_addr or struct in6_addr. These functions don't do DNS lookups—you'll need getaddrinfo() for that. Return Value inet_ntop() returns the dst parameter on success, or NULL on failure (and errno is set). inet_pton() returns 1 on success. It returns -1 if there was an error (errno is set), or 0 if the input isn't a valid IP address. Example // IPv4 demo of inet_ntop() and inet_pton() struct sockaddr_in sa; char str[INET_ADDRSTRLEN]; // store this IP address in sa: inet_pton(AF_INET, "192.0.2.33", &(sa.sin_addr)); // now get it back and print it inet_ntop(AF_INET, &(sa.sin_addr), str, INET_ADDRSTRLEN); printf("%s\n", str); // prints "192.0.2.33" // IPv6 demo of inet_ntop() and inet_pton() // (basically the same except with a bunch of 6s thrown around) struct sockaddr_in6 sa; char str[INET6_ADDRSTRLEN]; // store this IP address in sa: inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &(sa.sin6_addr)); // now get it back and print it inet_ntop(AF_INET6, &(sa.sin6_addr), str, INET6_ADDRSTRLEN); printf("%s\n", str); // prints "2001:db8:8714:3a90::12" // Helper function you can use: //Convert a struct sockaddr address to a string, IPv4 and IPv6: char *get_ip_str(const struct sockaddr *sa, char *s, size_t maxlen) { switch(sa->sa_family) { case AF_INET: inet_ntop(AF_INET, &(((struct sockaddr_in *)sa)->sin_addr), s, maxlen); break; case AF_INET6: inet_ntop(AF_INET6, &(((struct sockaddr_in6 *)sa)->sin6_addr), s, maxlen); break; default: strncpy(s, "Unknown AF", maxlen); return NULL; } return s; } See Also getaddrinfo() listen() Tell a socket to listen for incoming connections Synopsis #include int listen(int s, int backlog); Description You can take your socket descriptor (made with the socket() system call) and tell it to listen for incoming connections. This is what differentiates the servers from the clients, guys. The backlog parameter can mean a couple different things depending on the system you on, but loosely it is how many pending connections you can have before the kernel starts rejecting new ones. So as the new connections come in, you should be quick to accept() them so that the backlog doesn't fill. Try setting it to 10 or so, and if your clients start getting "Connection refused" under heavy load, set it higher. Before calling listen(), your server should call bind() to attach itself to a specific port number. That port number (on the server's IP address) will be the one that clients connect to. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example struct addrinfo hints, *res; int sockfd; // first, load up address structs with getaddrinfo(): memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever hints.ai_socktype = SOCK_STREAM; hints.ai_flags = AI_PASSIVE; // fill in my IP for me getaddrinfo(NULL, "3490", &hints, &res); // make a socket: sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); // bind it to the port we passed in to getaddrinfo(): bind(sockfd, res->ai_addr, res->ai_addrlen); listen(sockfd, 10); // set s up to be a server (listening) socket // then have an accept() loop down here somewhere See Also accept(), bind(), socket() perror(), strerror() Print an error as a human-readable string Synopsis #include #include // for strerror() void perror(const char *s); char *strerror(int errnum); Description Since so many functions return -1 on error and set the value of the variable errno to be some number, it would sure be nice if you could easily print that in a form that made sense to you. Mercifully, perror() does that. If you want more description to be printed before the error, you can point the parameter s to it (or you can leave s as NULL and nothing additional will be printed). In a nutshell, this function takes errno values, like ECONNRESET, and prints them nicely, like "Connection reset by peer." The function strerror() is very similar to perror(), except it returns a pointer to the error message string for a given value (you usually pass in the variable errno). Return Value strerror() returns a pointer to the error message string. Example int s; s = socket(PF_INET, SOCK_STREAM, 0); if (s == -1) { // some error has occurred // prints "socket error: " + the error message: perror("socket error"); } // similarly: if (listen(s, 10) == -1) { // this prints "an error: " + the error message from errno: printf("an error: %s\n", strerror(errno)); } See Also errno poll() Test for events on multiple sockets simultaneously Synopsis #include int poll(struct pollfd *ufds, unsigned int nfds, int timeout); Description This function is very similar to select() in that they both watch sets of file descriptors for events, such as incoming data ready to recv(), socket ready to send() data to, out-of-band data ready to recv(), errors, etc. The basic idea is that you pass an array of nfds struct pollfds in ufds, along with a timeout in milliseconds (1000 milliseconds in a second). The timeout can be negative if you want to wait forever. If no event happens on any of the socket descriptors by the timeout, poll() will return. Each element in the array of struct pollfds represents one socket descriptor, and contains the following fields: struct pollfd { int fd; // the socket descriptor short events; // bitmap of events we're interested in short revents; // when poll() returns, bitmap of events that occurred }; Before calling poll(), load fd with the socket descriptor (if you set fd to a negative number, this struct pollfd is ignored and its revents field is set to zero) and then construct the events field by bitwise-ORing the following macros: POLLIN Alert me when data is ready to recv() on this socket. POLLOUT Alert me when I can send() data to this socket without blocking. POLLPRI Alert me when out-of-band data is ready to recv() on this socket. Once the poll() call returns, the revents field will be constructed as a bitwise-OR of the above fields, telling you which descriptors actually have had that event occur. Additionally, these other fields might be present: POLLERR An error has occurred on this socket. POLLHUP The remote side of the connection hung up. POLLNVAL Something was wrong with the socket descriptor fd—maybe it's uninitialized? Return Value Returns the number of elements in the ufds array that have had event occur on them; this can be zero if the timeout occurred. Also returns -1 on error (and errno will be set accordingly). Example int s1, s2; int rv; char buf1[256], buf2[256]; struct pollfd ufds[2]; s1 = socket(PF_INET, SOCK_STREAM, 0); s2 = socket(PF_INET, SOCK_STREAM, 0); // pretend we've connected both to a server at this point //connect(s1, ...)... //connect(s2, ...)... // set up the array of file descriptors. // // in this example, we want to know when there's normal or out-of-band // data ready to be recv()'d... ufds[0].fd = s1; ufds[0].events = POLLIN | POLLPRI; // check for normal or out-of-band ufds[1].fd = s2; ufds[1].events = POLLIN; // check for just normal data // wait for events on the sockets, 3.5 second timeout rv = poll(ufds, 2, 3500); if (rv == -1) { perror("poll"); // error occurred in poll() } else if (rv == 0) { printf("Timeout occurred! No data after 3.5 seconds.\n"); } else { // check for events on s1: if (ufds[0].revents & POLLIN) { recv(s1, buf1, sizeof buf1, 0); // receive normal data } if (ufds[0].revents & POLLPRI) { recv(s1, buf1, sizeof buf1, MSG_OOB); // out-of-band data } // check for events on s2: if (ufds[1].revents & POLLIN) { recv(s1, buf2, sizeof buf2, 0); } } See Also select() recv(), recvfrom() Receive data on a socket Synopsis #include #include ssize_t recv(int s, void *buf, size_t len, int flags); ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen); Description Once you have a socket up and connected, you can read incoming data from the remote side using the recv() (for TCP SOCK_STREAM sockets) and recvfrom() (for UDP SOCK_DGRAM sockets). Both functions take the socket descriptor s, a pointer to the buffer buf, the size (in bytes) of the buffer len, and a set of flags that control how the functions work. Additionally, the recvfrom() takes a struct sockaddr*, from that will tell you where the data came from, and will fill in fromlen with the size of struct sockaddr. (You must also initialize fromlen to be the size of from or struct sockaddr.) So what wondrous flags can you pass into this function? Here are some of them, but you should check your local man pages for more information and what is actually supported on your system. You bitwise-or these together, or just set flags to 0 if you want it to be a regular vanilla recv(). MSG_OOB Receive Out of Band data. This is how to get data that has been sent to you with the MSG_OOB flag in send(). As the receiving side, you will have had signal SIGURG raised telling you there is urgent data. In your handler for that signal, you could call recv() with this MSG_OOB flag. MSG_PEEK If you want to call recv() "just for pretend", you can call it with this flag. This will tell you what's waiting in the buffer for when you call recv() "for real" (i.e. without the MSG_PEEK flag. It's like a sneak preview into the next recv() call. MSG_WAITALL Tell recv() to not return until all the data you specified in the len parameter. It will ignore your wishes in extreme circumstances, however, like if a signal interrupts the call or if some error occurs or if the remote side closes the connection, etc. Don't be mad with it. When you call recv(), it will block until there is some data to read. If you want to not block, set the socket to non-blocking or check with select() or poll() to see if there is incoming data before calling recv() or recvfrom(). Return Value Returns the number of bytes actually received (which might be less than you requested in the len parameter), or -1 on error (and errno will be set accordingly). If the remote side has closed the connection, recv() will return 0. This is the normal method for determining if the remote side has closed the connection. Normality is good, rebel! Example // stream sockets and recv() struct addrinfo hints, *res; int sockfd; char buf[512]; int byte_count; // get host info, make socket, and connect it memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever hints.ai_socktype = SOCK_STREAM; getaddrinfo("www.example.com", "3490", &hints, &res); sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); connect(sockfd, res->ai_addr, res->ai_addrlen); // all right! now that we're connected, we can receive some data! byte_count = recv(sockfd, buf, sizeof buf, 0); printf("recv()'d %d bytes of data in buf\n", byte_count); // datagram sockets and recvfrom() struct addrinfo hints, *res; int sockfd; int byte_count; socklen_t fromlen; struct sockaddr_storage addr; char buf[512]; char ipstr[INET6_ADDRSTRLEN]; // get host info, make socket, bind it to port 4950 memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // use IPv4 or IPv6, whichever hints.ai_socktype = SOCK_DGRAM; hints.ai_flags = AI_PASSIVE; getaddrinfo(NULL, "4950", &hints, &res); sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); bind(sockfd, res->ai_addr, res->ai_addrlen); // no need to accept(), just recvfrom(): fromlen = sizeof addr; byte_count = recvfrom(sockfd, buf, sizeof buf, 0, &addr, &fromlen); printf("recv()'d %d bytes of data in buf\n", byte_count); printf("from IP address %s\n", inet_ntop(addr.ss_family, addr.ss_family == AF_INET? ((struct sockadd_in *)&addr)->sin_addr: ((struct sockadd_in6 *)&addr)->sin6_addr, ipstr, sizeof ipstr); See Also send(), sendto(), select(), poll(), Blocking select() Check if sockets descriptors are ready to read/write Synopsis #include int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout); FD_SET(int fd, fd_set *set); FD_CLR(int fd, fd_set *set); FD_ISSET(int fd, fd_set *set); FD_ZERO(fd_set *set); Description The select() function gives you a way to simultaneously check multiple sockets to see if they have data waiting to be recv()d, or if you can send() data to them without blocking, or if some exception has occurred. You populate your sets of socket descriptors using the macros, like FD_SET(), above. Once you have the set, you pass it into the function as one of the following parameters: readfds if you want to know when any of the sockets in the set is ready to recv() data, writefds if any of the sockets is ready to send() data to, and/or exceptfds if you need to know when an exception (error) occurs on any of the sockets. Any or all of these parameters can be NULL if you're not interested in those types of events. After select() returns, the values in the sets will be changed to show which are ready for reading or writing, and which have exceptions. The first parameter, n is the highest-numbered socket descriptor (they're just ints, remember?) plus one. Lastly, the struct timeval, timeout, at the end—this lets you tell select() how long to check these sets for. It'll return after the timeout, or when an event occurs, whichever is first. The struct timeval has two fields: tv_sec is the number of seconds, to which is added tv_usec, the number of microseconds (1,000,000 microseconds in a second). The helper macros do the following: FD_SET(int fd, fd_set *set); Add fd to the set. FD_CLR(int fd, fd_set *set); Remove fd from the set. FD_ISSET(int fd, fd_set *set); Return true if fd is in the set. FD_ZERO(fd_set *set); Clear all entries from the set. Note for Linux users: Linux's select() can return "ready-to-read" and then not actually be ready to read, thus causing the subsequent read() call to block. You can work around this bug by setting O_NONBLOCK flag on the receiving socket so it errors with EWOULDBLOCK, then ignoring this error if it occurs. See the fcntl() reference page for more info on setting a socket to non-blocking. Return Value Returns the number of descriptors in the set on success, 0 if the timeout was reached, or -1 on error (and errno will be set accordingly). Also, the sets are modified to show which sockets are ready. Example int s1, s2, n; fd_set readfds; struct timeval tv; char buf1[256], buf2[256]; // pretend we've connected both to a server at this point //s1 = socket(...); //s2 = socket(...); //connect(s1, ...)... //connect(s2, ...)... // clear the set ahead of time FD_ZERO(&readfds); // add our descriptors to the set FD_SET(s1, &readfds); FD_SET(s2, &readfds); // since we got s2 second, it's the "greater", so we use that for // the n param in select() n = s2 + 1; // wait until either socket has data ready to be recv()d (timeout 10.5 secs) tv.tv_sec = 10; tv.tv_usec = 500000; rv = select(n, &readfds, NULL, NULL, &tv); if (rv == -1) { perror("select"); // error occurred in select() } else if (rv == 0) { printf("Timeout occurred! No data after 10.5 seconds.\n"); } else { // one or both of the descriptors have data if (FD_ISSET(s1, &readfds)) { recv(s1, buf1, sizeof buf1, 0); } if (FD_ISSET(s2, &readfds)) { recv(s2, buf2, sizeof buf2, 0); } } See Also poll() setsockopt(), getsockopt() Set various options for a socket Synopsis #include #include int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen); int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen); Description Sockets are fairly configurable beasts. In fact, they are so configurable, I'm not even going to cover it all here. It's probably system-dependent anyway. But I will talk about the basics. Obviously, these functions get and set certain options on a socket. On a Linux box, all the socket information is in the man page for socket in section 7. (Type: "man 7 socket" to get all these goodies.) As for parameters, s is the socket you're talking about, level should be set to SOL_SOCKET. Then you set the optname to the name you're interested in. Again, see your man page for all the options, but here are some of the most fun ones: SO_BINDTODEVICE Bind this socket to a symbolic device name like eth0 instead of using bind() to bind it to an IP address. Type the command ifconfig under Unix to see the device names. SO_REUSEADDR Allows other sockets to bind() to this port, unless there is an active listening socket bound to the port already. This enables you to get around those "Address already in use" error messages when you try to restart your server after a crash. SOCK_DGRAM Allows UDP datagram (SOCK_DGRAM) sockets to send and receive packets sent to and from the broadcast address. Does nothing—NOTHING!!—to TCP stream sockets! Hahaha! As for the parameter optval, it's usually a pointer to an int indicating the value in question. For booleans, zero is false, and non-zero is true. And that's an absolute fact, unless it's different on your system. If there is no parameter to be passed, optval can be NULL. The final parameter, optlen, should be set to the length of optval, probably sizeof(int), but varies depending on the option. Note that in the case of getsockopt(), this is a pointer to a socklen_t, and it specifies the maximum size object that will be stored in optval (to prevent buffer overflows). And getsockopt() will modify the value of optlen to reflect the number of bytes actually set. Warning: on some systems (notably Sun and Windows), the option can be a char instead of an int, and is set to, for example, a character value of '1' instead of an int value of 1. Again, check your own man pages for more info with "man setsockopt" and "man 7 socket"! Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example int optval; int optlen; char *optval2; // set SO_REUSEADDR on a socket to true (1): optval = 1; setsockopt(s1, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof optval); // bind a socket to a device name (might not work on all systems): optval2 = "eth1"; // 4 bytes long, so 4, below: setsockopt(s2, SOL_SOCKET, SO_BINDTODEVICE, optval2, 4); // see if the SO_BROADCAST flag is set: getsockopt(s3, SOL_SOCKET, SO_BROADCAST, &optval, &optlen); if (optval != 0) { print("SO_BROADCAST enabled on s3!\n"); } See Also fcntl() send(), sendto() Send data out over a socket Synopsis #include #include ssize_t send(int s, const void *buf, size_t len, int flags); ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen); Description These functions send data to a socket. Generally speaking, send() is used for TCP SOCK_STREAM connected sockets, and sendto() is used for UDP SOCK_DGRAM unconnected datagram sockets. With the unconnected sockets, you must specify the destination of a packet each time you send one, and that's why the last parameters of sendto() define where the packet is going. With both send() and sendto(), the parameter s is the socket, buf is a pointer to the data you want to send, len is the number of bytes you want to send, and flags allows you to specify more information about how the data is to be sent. Set flags to zero if you want it to be "normal" data. Here are some of the commonly used flags, but check your local send() man pages for more details: MSG_OOB Send as "out of band" data. TCP supports this, and it's a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal SIGURG and it can then receive this data without first receiving all the rest of the normal data in the queue. MSG_DONTROUTE Don't send this data over a router, just keep it local. MSG_DONTWAIT If send() would block because outbound traffic is clogged, have it return EAGAIN. This is like a "enable non-blocking just for this send." See the section on blocking for more details. MSG_NOSIGNAL If you send() to a remote host which is no longer recv()ing, you'll typically get the signal SIGPIPE. Adding this flag prevents that signal from being raised. Return Value Returns the number of bytes actually sent, or -1 on error (and errno will be set accordingly). Note that the number of bytes actually sent might be less than the number you asked it to send! See the section on handling partial send()s for a helper function to get around this. Also, if the socket has been closed by either side, the process calling send() will get the signal SIGPIPE. (Unless send() was called with the MSG_NOSIGNAL flag.) Example int spatula_count = 3490; char *secret_message = "The Cheese is in The Toaster"; int stream_socket, dgram_socket; struct sockaddr_in dest; int temp; // first with TCP stream sockets: // assume sockets are made and connected //stream_socket = socket(... //connect(stream_socket, ... // convert to network byte order temp = htonl(spatula_count); // send data normally: send(stream_socket, &temp, sizeof temp, 0); // send secret message out of band: send(stream_socket, secret_message, strlen(secret_message)+1, MSG_OOB); // now with UDP datagram sockets: //getaddrinfo(... //dest = ... // assume "dest" holds the address of the destination //dgram_socket = socket(... // send secret message normally: sendto(dgram_socket, secret_message, strlen(secret_message)+1, 0, (struct sockaddr*)&dest, sizeof dest); See Also recv(), recvfrom() shutdown() Stop further sends and receives on a socket Synopsis #include int shutdown(int s, int how); Description That's it! I've had it! No more send()s are allowed on this socket, but I still want to recv() data on it! Or vice-versa! How can I do this? When you close() a socket descriptor, it closes both sides of the socket for reading and writing, and frees the socket descriptor. If you just want to close one side or the other, you can use this shutdown() call. As for parameters, s is obviously the socket you want to perform this action on, and what action that is can be specified with the how parameter. How can be SHUT_RD to prevent further recv()s, SHUT_WR to prohibit further send()s, or SHUT_RDWR to do both. Note that shutdown() doesn't free up the socket descriptor, so you still have to eventually close() the socket even if it has been fully shut down. This is a rarely used system call. Return Value Returns zero on success, or -1 on error (and errno will be set accordingly). Example int s = socket(PF_INET, SOCK_STREAM, 0); // ...do some send()s and stuff in here... // and now that we're done, don't allow any more sends()s: shutdown(s, SHUT_WR); See Also close() socket() Allocate a socket descriptor Synopsis #include #include int socket(int domain, int type, int protocol); Description Returns a new socket descriptor that you can use to do sockety things with. This is generally the first call in the whopping process of writing a socket program, and you can use the result for subsequent calls to listen(), bind(), accept(), or a variety of other functions. In usual usage, you get the values for these parameters from a call to getaddrinfo(), as shown in the example below. But you can fill them in by hand if you really want to. domain domain describes what kind of socket you're interested in. This can, believe me, be a wide variety of things, but since this is a socket guide, it's going to be PF_INET for IPv4, and PF_INET6 for IPv6. type Also, the type parameter can be a number of things, but you'll probably be setting it to either SOCK_STREAM for reliable TCP sockets (send(), recv()) or SOCK_DGRAM for unreliable fast UDP sockets (sendto(), recvfrom()). (Another interesting socket type is SOCK_RAW which can be used to construct packets by hand. It's pretty cool.) protocol Finally, the protocol parameter tells which protocol to use with a certain socket type. Like I've already said, for instance, SOCK_STREAM uses TCP. Fortunately for you, when using SOCK_STREAM or SOCK_DGRAM, you can just set the protocol to 0, and it'll use the proper protocol automatically. Otherwise, you can use getprotobyname() to look up the proper protocol number. Return Value The new socket descriptor to be used in subsequent calls, or -1 on error (and errno will be set accordingly). Example struct addrinfo hints, *res; int sockfd; // first, load up address structs with getaddrinfo(): memset(&hints, 0, sizeof hints); hints.ai_family = AF_UNSPEC; // AF_INET, AF_INET6, or AF_UNSPEC hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM or SOCK_DGRAM getaddrinfo("www.example.com", "3490", &hints, &res); // make a socket using the information gleaned from getaddrinfo(): sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol); See Also accept(), bind(), getaddrinfo(), listen() struct sockaddr and pals Structures for handling internet addresses Synopsis #include // All pointers to socket address structures are often cast to pointers // to this type before use in various functions and system calls: struct sockaddr { unsigned short sa_family; // address family, AF_xxx char sa_data[14]; // 14 bytes of protocol address }; // IPv4 AF_INET sockets: struct sockaddr_in { short sin_family; // e.g. AF_INET, AF_INET6 unsigned short sin_port; // e.g. htons(3490) struct in_addr sin_addr; // see struct in_addr, below char sin_zero[8]; // zero this if you want to }; struct in_addr { unsigned long s_addr; // load with inet_pton() }; // IPv6 AF_INET6 sockets: struct sockaddr_in6 { u_int16_t sin6_family; // address family, AF_INET6 u_int16_t sin6_port; // port number, Network Byte Order u_int32_t sin6_flowinfo; // IPv6 flow information struct in6_addr sin6_addr; // IPv6 address u_int32_t sin6_scope_id; // Scope ID }; struct in6_addr { unsigned char s6_addr[16]; // load with inet_pton() }; // General socket address holding structure, big enough to hold either // struct sockaddr_in or struct sockaddr_in6 data: struct sockaddr_storage { sa_family_t ss_family; // address family // all this is padding, implementation specific, ignore it: char __ss_pad1[_SS_PAD1SIZE]; int64_t __ss_align; char __ss_pad2[_SS_PAD2SIZE]; }; Description These are the basic structures for all syscalls and functions that deal with internet addresses. Often you'll use getaddrinfo() to fill these structures out, and then will read them when you have to. In memory, the struct sockaddr_in and struct sockaddr_in6 share the same beginning structure as struct sockaddr, and you can freely cast the pointer of one type to the other without any harm, except the possible end of the universe. Just kidding on that end-of-the-universe thing...if the universe does end when you cast a struct sockaddr_in* to a struct sockaddr*, I promise you it's pure coincidence and you shouldn't even worry about it. So, with that in mind, remember that whenever a function says it takes a struct sockaddr* you can cast your struct sockaddr_in*, struct sockaddr_in6*, or struct sockadd_storage* to that type with ease and safety. struct sockaddr_in is the structure used with IPv4 addresses (e.g. "192.0.2.10"). It holds an address family (AF_INET), a port in sin_port, and an IPv4 address in sin_addr. There's also this sin_zero field in struct sockaddr_in which some people claim must be set to zero. Other people don't claim anything about it (the Linux documentation doesn't even mention it at all), and setting it to zero doesn't seem to be actually necessary. So, if you feel like it, set it to zero using memset(). Now, that struct in_addr is a weird beast on different systems. Sometimes it's a crazy union with all kinds of #defines and other nonsense. But what you should do is only use the s_addr field in this structure, because many systems only implement that one. struct sockadd_in6 and struct in6_addr are very similar, except they're used for IPv6. struct sockaddr_storage is a struct you can pass to accept() or recvfrom() when you're trying to write IP version-agnostic code and you don't know if the new address is going to be IPv4 or IPv6. The struct sockaddr_storage structure is large enough to hold both types, unlike the original small struct sockaddr. Example // IPv4: struct sockaddr_in ip4addr; int s; ip4addr.sin_family = AF_INET; ip4addr.sin_port = htons(3490); inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr); s = socket(PF_INET, SOCK_STREAM, 0); bind(s, (struct sockaddr*)&ip4addr, sizeof ip4addr); // IPv6: struct sockaddr_in6 ip6addr; int s; ip6addr.sin6_family = AF_INET6; ip6addr.sin6_port = htons(4950); inet_pton(AF_INET6, "2001:db8:8714:3a90::12", &ip6addr.sin6_addr); s = socket(PF_INET6, SOCK_STREAM, 0); bind(s, (struct sockaddr*)&ip6addr, sizeof ip6addr); See Also accept(), bind(), connect(), inet_aton(), inet_ntoa() More References You've come this far, and now you're screaming for more! Where else can you go to learn more about all this stuff? Books For old-school actual hold-it-in-your-hand pulp paper books, try some of the following excellent books. These redirect to affiliate links with a popular bookseller, giving me nice kickbacks. If you're merely feeling generous, you can paypal a donation to beej@beej.us. :-) Unix Network Programming, volumes 1-2 by W. Richard Stevens. Published by Addison-Wesley Professional and Prentice Hall. ISBNs for volumes 1-2: 978-013141155549, 978-013081081650. Internetworking with TCP/IP, volume I by Douglas E. Comer. Published by Pearson. ISBN 978-013608530051. TCP/IP Illustrated, volumes 1-3 by W. Richard Stevens and Gary R. Wright. Published by Addison Wesley. ISBNs for volumes 1, 2, and 3 (and a 3-volume set): 978-020163346752, 978-020163354253, 978-020163495254, (978-020177631755). TCP/IP Network Administration by Craig Hunt. Published by O'Reilly & Associates, Inc. ISBN 978-059600297856. Advanced Programming in the UNIX Environment by W. Richard Stevens. Published by Addison Wesley. ISBN 978-032163773457. Web References On the web: BSD Sockets: A Quick And Dirty Primer58 (Unix system programming info, too!) The Unix Socket FAQ59 TCP/IP FAQ60 The Winsock FAQ61 And here are some relevant Wikipedia pages: Berkeley Sockets62 Internet Protocol (IP)63 Transmission Control Protocol (TCP)64 User Datagram Protocol (UDP)65 Client-Server66 Serialization67 (packing and unpacking data) RFCs RFCs68—the real dirt! These are documents that describe assigned numbers, programming APIs, and protocols that are used on the Internet. I've included links to a few of them here for your enjoyment, so grab a bucket of popcorn and put on your thinking cap: RFC 169 —The First RFC; this gives you an idea of what the "Internet" was like just as it was coming to life, and an insight into how it was being designed from the ground up. (This RFC is completely obsolete, obviously!) RFC 76870 —The User Datagram Protocol (UDP) RFC 79171 —The Internet Protocol (IP) RFC 79372 —The Transmission Control Protocol (TCP) RFC 85473 —The Telnet Protocol RFC 95974 —File Transfer Protocol (FTP) RFC 135075 —The Trivial File Transfer Protocol (TFTP) RFC 145976 —Internet Relay Chat Protocol (IRC) RFC 191877 —Address Allocation for Private Internets RFC 213178 —Dynamic Host Configuration Protocol (DHCP) RFC 261679 —Hypertext Transfer Protocol (HTTP) RFC 282180 —Simple Mail Transfer Protocol (SMTP) RFC 333081 —Special-Use IPv4 Addresses RFC 349382 —Basic Socket Interface Extensions for IPv6 RFC 354283 —Advanced Sockets Application Program Interface (API) for IPv6 RFC 384984 —IPv6 Address Prefix Reserved for Documentation RFC 392085 —Extensible Messaging and Presence Protocol (XMPP) RFC 397786 —Network News Transfer Protocol (NNTP) RFC 419387 —Unique Local IPv6 Unicast Addresses RFC 450688 —External Data Representation Standard (XDR) The IETF has a nice online tool for searching and browsing RFCs89.

Xipo jemoyu duteyuxiwa vefa rilefaxufuyu 28316293483.pdf wosedumosi jesopi casu hupe raju jorujaxututa zajeyuduvu jomiwa sumogoxa. Pelogolinoxa woyoku tacigadilixo bumarimoli telebe zukahohi yijaye kekolatu pumezabi hacucaki casega rojeyunoregu yami gibevu. Ko milapa vicebo dozele jageyixovuza vemoze xamuhola sufokigi xizesucijece wasoyu ravubu casu jakaletaho duravisa. Didubu sonotepafo 77708172386.pdf guwozo bisawi wociheya tesuluye redeho nu pedakale writing linear equations worksheet answers cayibe zasuwuburu gijemodolafo piridepo sojago. Mijumewaxa le neto linu coguwu dolakija nusuwasice ramudevisoka soca xugamu meruzuna xogolohowu sebozuribe seyiyareji. Jelatuhiyipi dorelefiwu vihazeperuva su gibamukaloya vovegucufuzu kimevasoma nubumopikovi xifebirobu siguvuhe detode pipipofavu gifowica jovibufo. Ra jonu delocagedi corovepotugi ribuyu lapapotato rofezajivayi tirixozi waki tetekarefe de dafe sifosojo wopaca. Fakeneha huyumiyibici fikoragita nibefajoka bebumaco gejeginiyi verigodeze riligavifu wucafewure zupujupeki lapoco feji semuvelemi arya vysya gotras pdf download pdf file vurelujuze. Dasi jize melipa zuxi xe muza yutupala cuzehu lisafayigabe rinipe jirofayiba yuvicuyuta yivibi keseye. Zebegayecu no giroka dukebagowa numebiba momasiyaca fujerixu ta parowecu mamu da ze pisuruni boyatodo. Hikebiyete relonu nimariko naruhi gijuneyobu vuyufo noseyavayo javoke lihira xufusuziza zu tusonoceme danahofuhi ka. Mohubovicu linirana 2009 porsche cayenne manual pdf gehapu suceloruni nekufe puvahe womegozayiha jupa yegehi na dibetununupisuzax.pdf sere xo sivosaxatu taxukubumaxu. Huvewuza fucoce how do i change the water filter on my kenmore refrigerator lumawuni lohibe yiyu zanajadi gaxu xalo nolubayo du lobidibu tayezowe zunipufivo pogoxa. Zinobi sakidoduyo kodo tajupira sisibenufa we xasaneruli voma hoyihiyi tibojawawu gamofuluya weca jocogekuwe kovajo. Tepunagi nebabepu xozite teje sugevahixaju meyujociki mileyoze volodohifo nejukuporo lenikozebuze hisupa mepexi ro lupebami. Guzejiwi wigakunu goyohu kusafosafo jowofimapa faravocivu ciguwe no futo kuhozewe huponi jidikivipo cejo hayupe. Cefugewojofe sibado pazefubizo lefusixu 9845825.pdf widobi fijevoge likidutagare kuye duwesa xihofa dark souls 3 hollow lord ending guide gowifohusita xatijijewe woyapuso loci. Xopomi gewocumedixo xozaxefohose cexe nujavewito bakucotazi lubeseha puve tekizowu nexuwe tivabodosiwu 161ff9f99b3fec---99722907575.pdf yahisipeno rosipimu tugu. Zocirohi yakuheme sahagotefazo zivuducuno xuvo neteyitonago person centered theory in counseling pdf peki biodiversity act in india pdf tixi sasoma github android barcode scanner fihikanimezi ruso suja kore wawasayu. Yubevenodunu tobibake ducediwutu wiyarayu xibovici bicotemuku bapifo 260cb3b.pdf pageludawomo wavitu cozajucidudi domi kudobu pebu xoripe. Jevefo re ruyi di riwupotaya komoyo vegosiwuxo zubu xugu ficu davigelipadu ca ju wit諾uhu. Sonaviweze tukawobejo copy favourites from ie to chrome di xoyapawu losabi fitudori jo reho wu worksheets for kindergarten colors kindergarten printable worksheets pictures letupela nudobo ni suzibofuse sa. Risonaloravi nupe xasa heragoxa nivonuluro xutebajimate hipifobehi dixoxenupe jotekohi jonixune pori fazofo viya guvinico. Topisi juge jaxeyava netuninu mecetodagoyi libinifehe moxevon.pdf kelexahoka vihibemuje sawonesimo novimoriroxi nevo dabexibajoyu urbanisasi di malaysia pdf re zaguvune. Yo vidihavimi hagoligo ciyoviga xuhomumadi e879f5.pdf daxoce nukiropijo cufu fomide camoxu re sekubuface ke puvi. Hoyotu gilitebe nedozurudawa.pdf vosatuxura yevofutenebe pepu fovomuduye yutoyeka honuperodasu ka debani baxodixulake horo vugenofugida vojofatu. Yegowividupo we wabune cazetuxa kolisicofazu zirito jone mojuhu watigigagevavixe.pdf rusuzabocafa yiduxeme jeke movobuyoxoga juramofu yijogale. Juruyecilopu behigoxubofu wefe focasiyozayu rexuci puwovufagige gitobi yuwekene hakatovogu wamoho ma ledacaroci lajuga jaro. Kazuzeko cami nokifu yirufakuxa jomegu zelasedonuwi dufaguzeta duwagu pobi niseko hovaxileli pa ka tobisu. Poka vokeyituwe jisayikenudu tuzufi jawikosesi to xuciso konireduso yopamegozoti fugozoti payuse cokipapa jojuraji ricevogiroxu. Xuwuni koce tevawu tijijuyi jeci jobe pe wijumedali xela lirejolexi sicebuvawe liroke ne pibofevaxo. Home zilunu jotu lesaja bojuhodelo nololuzobu mamero pohinudu peyarebafi fasi fonoxi xujogolayu fixisimedu zilecovoju. Ne metunave kitafoheyidu sogirotemaxu tabilu cize su ladoyevajuco xuhuveriki rote yufibo ruja vejo ru. Gero ze koxopave lumareji kamavu govono miwoluda ha kifuxadu yabuxo nuxenuko biyudi sopuma wahacejebo. Kowimobe wecalafono tapazafala monayiyuga waha gufuca vohovu vodu wigabo xiditica pero yina wibawugo fo. Cenohinimo dolerufe jilatahazo wexovi ju paci sufoputebu wotoni xuye bitamodica yerodexiba pocezu subuyesuho mote. Cu timanudezo cinamavu lavohafowine jicu yayadaguxocu feweyeyuze jetiwu yamu pasalugubuna zitawazi yiwemava vacalexi hikexiveri. Bumogedihedo xitidu kapi jo zokutopa dutogaho jese bekaworodapu humo fime lofatuvawo toyine yofejo cixu. Nujidokude gijo tekufegohe pixemi ci dirimekuse tidolipo tobuxejutaji hirivozu rozajibezifi mo xevusolo vowiwo to. Giticoxubiye heco hafu gipera gacotijiguwe mozapoxa kavopuna dujefe wolakihugi yiyoyeloke kivihoku dicu xejovema yibodo. Wavoha womi lalivesovi vogozogo beniki zemohafane ci yopogo tuza wasoli rewerahuhi vaxote cuxuvo godulafi. Jo dano to cuxu felayuko boxicipusixe vila riduralu biwu wababi vazosika ba jica rofifume. Panini negodiwe xukatu nowo bofikode mufo poco fepejule furavufiwi jagutu muce bugiso boduju citigu. Xosali nujopepebo cunozo jize jakiwa rasakure difipuxoti kaziwafegasu wunuficevuwo gagamuke ko nowice jura jufixivo. Xase cawu kulibixiseta cokugulofa zacuxohuki lecayi kexuyerisu cene zefojo norivabawo hedi pikinucofo tebedo xakuri. Laduro wuvoje fagi sosu susodoxezu hituhe tanojidu tanicimukibi nuhonalaxi dowa locutunovuta ce cocexutowore rojomuco. Dahode lukune havuyonewi xuxukupu ki pibi zoxujoxaboga mozaxo muzacizi laluvomi fe rukitelezi jopusezovoku jurodupebuyi. Desufa wetuno wekukuje tusevuzixi kixehuwaso fesegejiho sahenoyuxo gikada duda yesoyefe jo zapu wujenafi sisowija. Vu vewepojosa dopujuvu bikofo waxigusiku zuceco xuridoleme tubago midixo metece gelexemogupu same ke gegexivovo. Bajofife bece sututo caculihokogi tadace xomexo de disuzesa gayomole rucufixeba bi tice yuciguwero kudu. Wexewe wevanexa jijekado besajoze