Prev Contents Next

7. Common Questions

Where can I get those header files?

If you don't have them on your system already, you probably don't need them. Check the manual for your particular platform. If you're building for Windows, you only need to `#include <winsock.h>`.

What do I do when `bind()` reports "Address already in use"?

You have to use `setsockopt()` with the **`SO_REUSEADDR`** option on the listening socket. Check out the section on `bind()` and the section on `select()` for an example.

How do I get a list of open sockets on the system?

Use the **`netstat`**. Check the **`man`** page for full details, but you should get some good output just typing:

```
$ netstat
```

The only trick is determining which socket is associated with which program. `:-)`

How can I view the routing table?

Run the **`route`** command (in */sbin* on most Linuxes) or the command **`netstat -r`**.

How can I run the client and server programs if I only have one computer? Don't I need a network to write network program?

Fortunately for you, virtually all machines implement a loopback network "device" that sits in the kernel and pretends to be a network card. (This is the interface listed as "`lo`" in the routing table.)

Pretend you're logged into a machine named "`goat`". Run the client in one window and the server in another. Or start the server in the background ("**`server &`**") and run the client in the same window. The upshot of the loopback device is that you can either **`client goat`** or **`client localhost`** (since "`localhost`" is likely defined in your */etc/hosts* file) and you'll have the client talking to the server without a network!

In short, no changes are necessary to any of the code to make it run on a single non-networked machine! Huzzah!

How can I tell if the remote side has closed connection?

You can tell because `recv()` will return **`0`**.

How do I implement a "ping" utility? What is ICMP? Where can I find out more about raw sockets and `SOCK_RAW`?

All your raw sockets questions will be answered in W. Richard Stevens' UNIX Network Programming books. See the books section of this guide.

How do I build for Windows?

First, delete Windows and install Linux or BSD. `};-)`. No, actually, just see the section on building for Windows in the introduction.

How do I build for Solaris/SunOS? I keep getting linker errors when I try to compile!

The linker errors happen because Sun boxes don't automatically compile in the socket libraries. See the section on building for Solaris/SunOS in the introduction for an example of how to do this.

Why does `select()` keep falling out on a signal?

Signals tend to cause blocked system calls to return **`-1`** with *`errno`* set to **`EINTR`**. When you set up a signal handler with `sigaction()`, you can set the flag **`SA_RESTART`**, which is supposed to restart the system call after it was interrupted.

Naturally, this doesn't always work.

My favorite solution to this involves a `goto` statement. You know this irritates your professors to no end, so go for it!

```
select_restart:
if ((err = select(fdmax+1, &readfds, NULL, NULL, NULL)) == -1) {
    if (errno == EINTR) {
        // some signal just interrupted us, so restart
        goto select_restart;
    }
    // handle the real error here:
    perror("select");
}
```

Sure, you don't *need* to use `goto` in this case; you can use other structures to control it. But I think the `goto` statement is actually cleaner.

How can I implement a timeout on a call to `recv()`?

Use `select()`! It allows you to specify a timeout parameter for socket descriptors that you're looking to read from. Or, you could wrap the entire functionality in a single function, like this:

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

int recvtimeout(int s, char *buf, int len, int timeout)
{
    fd_set fds;
    int n;
    struct timeval tv;

    // set up the file descriptor set
    FD_ZERO(&fds);
    FD_SET(s, &fds);

    // set up the struct timeval for the timeout
    tv.tv_sec = timeout;
    tv.tv_usec = 0;

    // wait until timeout or data received
    n = select(s+1, &fds, NULL, NULL, &tv);
    if (n == 0) return -2; // timeout!
```

```
    if (n == -1) return -1; // error

    // data must be here, so do a normal recv()
    return recv(s, buf, len, 0);
}
.
.
.
// Sample call to recvtimeout():
n = recvtimeout(s, buf, sizeof(buf), 10); // 10 second timeout

if (n == -1) {
    // error occurred
    perror("recvtimeout");
}
else if (n == -2) {
    // timeout occurred
} else {
    // got some data in buf
}
.
.
.
```

Notice that `recvtimeout()` returns **`-2`** in case of a timeout. Why not return **`0`**? Well, if you recall, a return value of **`0`** on a call to `recv()` means that the remote side closed the connection. So that return value is already spoken for, and **`-1`** means "error", so I chose **`-2`** as my timeout indicator.

How do I encrypt or compress the data before sending it through the socket?

One easy way to do encryption is to use SSL (secure sockets layer), but that's beyond the scope of this guide.

But assuming you want to plug in or implement your own compressor or encryption system, it's just a matter of thinking of your data as running through a sequence of steps between both ends. Each step changes the data in some way.

1. server reads data from file (or whereever)
2. server encrypts data (you add this part)
3. server `send()`s encrypted data

Now the other way around:

1. client `recv()`s encrypted data
2. client decrypts data (you add this part)
3. client writes data to file (or whereever)

You can also do compression at the same point that you do the encryption/decryption, above. Or you could do both! Just remember to compress before you encrypt. `:)`

Just as long as the client properly undoes what the server does, the data will be fine in the end no matter how many intermediate steps you add.

So all you need to do to use my code is to find the place between where the data is read and the data is sent (using `send()`) over the network, and stick some code in there that does the encryption.

What is this "`PF_INET`" I keep seeing? Is it related to `AF_INET`?

Yes, yes it is. See the section on `socket()` for details.

How can I write a server that accepts shell commands from a client and executes them?

For simplicity, lets say the client `connect()`s, `send()`s, and `close()`s the connection (that is, there are no subsequent system calls without the client connecting again.)

The process the client follows is this:

1. `connect()` to server
2. `send("/sbin/ls > /tmp/client.out")`
3. `close()` the connection

Meanwhile, the server is handling the data and executing it:

1. `accept()` the connection from the client
2. `recv(str)` the command string
3. `close()` the connection
4. `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, that's what!

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 (!strcmp(str, "foobar")) {
    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++.

How do I send data securely with TCP/IP using encryption?

Check out the OpenSSL project.

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. `;-)`

Prev Contents Next