7. 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!

7.1. 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 <unistd.h>
#include <fcntl.h>
.
.
.
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 `EWOULDBLOCK`.

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 **`select()`**.

7.2. `select()`—Synchronous I/O Multiplexing

This function is somewhat strange, but it's very useful. Take the following situation: you are a server and you want to listen for incoming connections as well as keep reading from the connections you already have.

No problem, you say, just an **`accept()`** and a couple of **`recv()`**s. Not so fast, buster! What if you're blocking on an **`accept()`** call? How are you going to **`recv()`** data at the same time? “Use non-blocking sockets!” No way! You don't want to be a CPU hog. What, then?

`select()` gives you the power to monitor several sockets at the same time. It'll tell you which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions, if you really want to know that.

This being said, in modern times **`select()`**, though very portable, is one of the slowest methods for monitoring sockets. One possible alternative is libevent[24], or something similar, that encapsulates all the system-dependent stuff involved with getting socket notifications.

Without any further ado, I'll offer the synopsis of **`select()`**:

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

int select(int numfds, fd_set *readfds, fd_set *writefds,
```

[24] `http://www.monkey.org/~provos/libevent/`

```
           fd_set *exceptfds, struct timeval *timeout);
```

The function monitors “sets” of file descriptors; in particular *`readfds`*, *`writefds`*, and *`exceptfds`*. If you want to see if you can read from standard input and some socket descriptor, *`sockfd`*, just add the file descriptors `0` and *`sockfd`* to the set *`readfds`*. The parameter *`numfds`* should be set to the values of the highest file descriptor plus one. In this example, it should be set to *`sockfd+1`*, since it is assuredly higher than standard input (`0`).

When **`select()`** returns, *`readfds`* will be modified to reflect which of the file descriptors you selected which is ready for reading. You can test them with the macro **`FD_ISSET()`**, below.

Before progressing much further, I'll talk about how to manipulate these sets. Each set is of the type `fd_set`. The following macros operate on this type:

`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`*.

Finally, what is this weirded out `struct timeval`? Well, sometimes you don't want to wait forever for someone to send you some data. Maybe every 96 seconds you want to print “Still Going...” to the terminal even though nothing has happened. This time structure allows you to specify a timeout period. If the time is exceeded and **`select()`** still hasn't found any ready file descriptors, it'll return so you can continue processing.

The `struct timeval` has the follow fields:

```
struct timeval {
    int tv_sec;     // seconds
    int tv_usec;    // microseconds
};
```

Just set *`tv_sec`* to the number of seconds to wait, and set *`tv_usec`* to the number of microseconds to wait. Yes, that's *micro*seconds, not milliseconds. There are 1,000 microseconds in a millisecond, and 1,000 milliseconds in a second. Thus, there are 1,000,000 microseconds in a second. Why is it “usec”? The “u” is supposed to look like the Greek letter μ (Mu) that we use for “micro”. Also, when the function returns, *`timeout`* *might* be updated to show the time still remaining. This depends on what flavor of Unix you're running.

Yay! We have a microsecond resolution timer! Well, don't count on it. You'll probably have to wait some part of your standard Unix timeslice no matter how small you set your `struct timeval`.

Other things of interest: If you set the fields in your `struct timeval` to `0`, **`select()`** will timeout immediately, effectively polling all the file descriptors in your sets. If you set the parameter *`timeout`* to NULL, it will never timeout, and will wait until the first file descriptor is ready. Finally, if you don't care about waiting for a certain set, you can just set it to NULL in the call to **`select()`**.

The following code snippet[25] waits 2.5 seconds for something to appear on standard input:

```
/*
** select.c -- a select() demo
*/

#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

#define STDIN 0  // file descriptor for standard input

int main(void)
{
    struct timeval tv;
    fd_set readfds;
```

[25] `http://beej.us/guide/bgnet/examples/select.c`

```
    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    // don't care about writefds and exceptfds:
    select(STDIN+1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

If you're on a line buffered terminal, the key you hit should be RETURN or it will time out anyway.

Now, some of you might think this is a great way to wait for data on a datagram socket—and you are right: it *might* be. Some Unices can use select in this manner, and some can't. You should see what your local man page says on the matter if you want to attempt it.

Some Unices update the time in your `struct timeval` to reflect the amount of time still remaining before a timeout. But others do not. Don't rely on that occurring if you want to be portable. (Use **`gettimeofday()`** if you need to track time elapsed. It's a bummer, I know, but that's the way it is.)

What happens if a socket in the read set closes the connection? Well, in that case, **`select()`** returns with that socket descriptor set as “ready to read”. When you actually do **`recv()`** from it, **`recv()`** will return `0`. That's how you know the client has closed the connection.

One more note of interest about **`select()`**: if you have a socket that is **`listen()`**ing, you can check to see if there is a new connection by putting that socket's file descriptor in the *`readfds`* set.

And that, my friends, is a quick overview of the almighty **`select()`** function.

But, by popular demand, here is an in-depth example. Unfortunately, the difference between the dirt-simple example, above, and this one here is significant. But have a look, then read the description that follows it.

This program[26] acts like a simple multi-user chat server. Start it running in one window, then **telnet** to it (“**telnet hostname 9034**”) from multiple other windows. When you type something in one **telnet** session, it should appear in all the others.

```
/*
** selectserver.c -- a cheezy multiperson chat server
*/

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

#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);
    }

```

[26] `http://beej.us/guide/bgnet/examples/selectserver.c`

```
    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\n", 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\n");
        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; i++) {
            if (FD_ISSET(i, &read_fds)) { // we got one!!
                if (i == listener) {
                    // handle new connections
                    addrlen = sizeof remoteaddr;
                    newfd = accept(listener,
                        (struct sockaddr *)&remoteaddr,
                        &addrlen);

                    if (newfd == -1) {
                        perror("accept");
                    } else {
                        FD_SET(newfd, &master); // add to master set
                        if (newfd > fdmax) {    // keep track of the max
                            fdmax = newfd;
                        }
                        printf("selectserver: new connection from %s on "
                            "socket %d\n",
                            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)) <= 0) {
                        // got error or connection closed by client
                        if (nbytes == 0) {
                            // connection closed
                            printf("selectserver: socket %d hung up\n", i);
                        } else {
                            perror("recv");
                        }
                        close(i); // bye!
                        FD_CLR(i, &master); // remove from master set
                    } else {
                        // we got some data from a client
                        for(j = 0; j <= fdmax; j++) {
                            // send to everyone!
                            if (FD_ISSET(j, &master)) {
                                // except the listener and ourselves
                                if (j != listener && j != i) {
                                    if (send(j, buf, nbytes, 0) == -1) {
                                        perror("send");
                                    }
                                }
                            }
                        }
                    }
                } // END handle data from client
            } // END got new incoming connection
        } // END looping through file descriptors
    } // END for(;;)--and you thought it would never end!

    return 0;
}
```

Notice I have two file descriptor sets in the code: *master* and *read_fds*. The first, *master*, holds all the socket descriptors that are currently connected, as well as the socket descriptor that is listening for new connections.

The reason I have the *master* set is that **select()** actually *changes* the set you pass into it to reflect which sockets are ready to read. Since I have to keep track of the connections from one call of **select()** to the next, I must store these safely away somewhere. At the last minute, I copy the *master* into the *read_fds*, and then call **select()**.

But doesn't this mean that every time I get a new connection, I have to add it to the *master* set? Yup! And every time a connection closes, I have to remove it from the *master* set? Yes, it does.

Notice I check to see when the *listener* socket is ready to read. When it is, it means I have a new connection pending, and I **accept()** it and add it to the *master* set. Similarly, when a client connection is ready to read, and **recv()** returns `0`, I know the client has closed the connection, and I must remove it from the *master* set.

If the client **recv()** returns non-zero, though, I know some data has been received. So I get it, and then go through the *master* list and send that data to all the rest of the connected clients.

And that, my friends, is a less-than-simple overview of the almighty **select()** function.

In addition, here is a bonus afterthought: there is another function called **poll()** which behaves much the same way **select()** does, but with a different system for managing the file descriptor sets. Check it out!

7.3. Handling Partial send()s

Remember back in the section about **send()**, above, when I said that **send()** might not send all the bytes you asked it to? That is, you want it to send 512 bytes, but it returns 412. What happened to the remaining 100 bytes?

Well, they're still in your little buffer waiting to be sent out. Due to circumstances beyond your control, the kernel decided not to send all the data out in one chunk, and now, my friend, it's up to you to get the data out there.

You could write a function like this to do it, too:

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

int sendall(int s, char *buf, int *len)
{
    int total = 0;        // how many bytes we've sent
    int bytesleft = *len; // how many we have left to send
    int n;

    while(total < *len) {
        n = send(s, buf+total, bytesleft, 0);
        if (n == -1) { break; }
        total += n;
        bytesleft -= n;
    }

    *len = total; // return number actually sent here

    return n==-1?-1:0; // return -1 on failure, 0 on success
}
```

In this example, *s* is the socket you want to send the data to, *buf* is the buffer containing the data, and *len* is a pointer to an `int` containing the number of bytes in the buffer.

The function returns `-1` on error (and *errno* is still set from the call to **send()**.) Also, the number of bytes actually sent is returned in *len*. This will be the same number of bytes you asked it to send, unless there was an error. **`sendall()`** will do it's best, huffing and puffing, to send the data out, but if there's an error, it gets back to you right away.

For completeness, here's a sample call to the function:

```
char buf[10] = "Beej!";
int len;

len = strlen(buf);
if (sendall(s, buf, &len) == -1) {
    perror("sendall");
    printf("We only sent %d bytes because of the error!\n", len);
}
```

What happens on the receiver's end when part of a packet arrives? If the packets are variable length, how does the receiver know when one packet ends and another begins? Yes, real-world scenarios are a royal pain in the donkeys. You probably have to *encapsulate* (remember that from the data encapsulation section way back there at the beginning?) Read on for details!

7.4. Serialization—How to Pack Data

It's easy enough to send text data across the network, you're finding, but what happens if you want to send some “binary” data like `int`s or `float`s? It turns out you have a few options.

1. Convert the number into text with a function like **`sprintf()`**, then send the text. The receiver will parse the text back into a number using a function like **`strtol()`**.
2. Just send the data raw, passing a pointer to the data to **`send()`**.
3. Encode the number into a portable binary form. The receiver will decode it.

Sneak preview! Tonight only!

[*Curtain raises*]

Beej says, “I prefer Method Three, above!”

[*THE END*]

(Before I begin this section in earnest, I should tell you that there are libraries out there for doing this, and rolling your own and remaining portable and error-free is quite a challenge. So hunt around and do your homework before deciding to implement this stuff yourself. I include the information here for those curious about how things like this work.)

Actually all the methods, above, have their drawbacks and advantages, but, like I said, in general, I prefer the third method. First, though, let's talk about some of the drawbacks and advantages to the other two.

The first method, encoding the numbers as text before sending, has the advantage that you can easily print and read the data that's coming over the wire. Sometimes a human-readable protocol is excellent to use in a non-bandwidth-intensive situation, such as with Internet Relay Chat (IRC)[27]. However, it has the disadvantage that it is slow to convert, and the results almost always take up more space than the original number!

Method two: passing the raw data. This one is quite easy (but dangerous!): just take a pointer to the data to send, and call send with it.

```
double d = 3490.15926535;

send(s, &d, sizeof d, 0);  /* DANGER--non-portable! */
```

The receiver gets it like this:

```
double d;

recv(s, &d, sizeof d, 0);  /* DANGER--non-portable! */
```

[27] `http://en.wikipedia.org/wiki/Internet_Relay_Chat`

Fast, simple—what's not to like? Well, it turns out that not all architectures represent a `double` (or `int` for that matter) with the same bit representation or even the same byte ordering! The code is decidedly non-portable. (Hey—maybe you don't need portability, in which case this is nice and fast.)

When packing integer types, we've already seen how the **`htons()`**-class of functions can help keep things portable by transforming the numbers into Network Byte Order, and how that's the Right Thing to do. Unfortunately, there are no similar functions for `float` types. Is all hope lost?

Fear not! (Were you afraid there for a second? No? Not even a little bit?) There is something we can do: we can pack (or "marshal", or "serialize", or one of a thousand million other names) the data into a known binary format that the receiver can unpack on the remote side.

What do I mean by "known binary format"? Well, we've already seen the **`htons()`** example, right? It changes (or "encodes", if you want to think of it that way) a number from whatever the host format is into Network Byte Order. To reverse (unencode) the number, the receiver calls **`ntohs()`**.

But didn't I just get finished saying there wasn't any such function for other non-integer types? Yes. I did. And since there's no standard way in C to do this, it's a bit of a pickle (that a gratuitous pun there for you Python fans).

The thing to do is to pack the data into a known format and send that over the wire for decoding. For example, to pack `float`s, here's something quick and dirty with plenty of room for improvement:[28]

```
#include <stdint.h>

uint32_t htonf(float f)
{
    uint32_t p;
    uint32_t sign;

    if (f < 0) { sign = 1; f = -f; }
    else { sign = 0; }

    p = ((((uint32_t)f)&0x7fff)<<16) | (sign<<31); // whole part and sign
    p |= (uint32_t)(((f - (int)f) * 65536.0f))&0xffff; // fraction

    return p;
}

float ntohf(uint32_t p)
{
    float f = ((p>>16)&0x7fff); // whole part
    f += (p&0xffff) / 65536.0f; // fraction

    if (((p>>31)&0x1) == 0x1) { f = -f; } // sign bit set

    return f;
}
```

The above code is sort of a naive implementation that stores a `float` in a 32-bit number. The high bit (31) is used to store the sign of the number ("1" means negative), and the next seven bits (30-16) are used to store the whole number portion of the `float`. Finally, the remaining bits (15-0) are used to store the fractional portion of the number.

Usage is fairly straightforward:

```
#include <stdio.h>

int main(void)
{
    float f = 3.1415926, f2;
    uint32_t netf;

    netf = htonf(f);  // convert to "network" form
    f2 = ntohf(netf); // convert back to test

    printf("Original: %f\n", f);        // 3.141593
```

[28] `http://beej.us/guide/bgnet/examples/pack.c`

```
    printf(" Network: 0x%08X\n", netf); // 0x0003243F
    printf("Unpacked: %f\n", f2);       // 3.141586

    return 0;
}
```

On the plus side, it's small, simple, and fast. On the minus side, it's not an efficient use of space and the range is severely restricted—try storing a number greater-than 32767 in there and it won't be very happy! You can also see in the above example that the last couple decimal places are not correctly preserved.

What can we do instead? Well, *The* Standard for storing floating point numbers is known as IEEE-754[29]. Most computers use this format internally for doing floating point math, so in those cases, strictly speaking, conversion wouldn't need to be done. But if you want your source code to be portable, that's an assumption you can't necessarily make. (On the other hand, if you want things to be fast, you should optimize this out on platforms that don't need to do it! That's what **`htons()`** and its ilk do.)

Here's some code that encodes floats and doubles into IEEE-754 format[30]. (Mostly—it doesn't encode NaN or Infinity, but it could be modified to do that.)

```
#define pack754_32(f) (pack754((f), 32, 8))
#define pack754_64(f) (pack754((f), 64, 11))
#define unpack754_32(i) (unpack754((i), 32, 8))
#define unpack754_64(i) (unpack754((i), 64, 11))

uint64_t pack754(long double f, unsigned bits, unsigned expbits)
{
    long double fnorm;
    int shift;
    long long sign, exp, significand;
    unsigned significandbits = bits - expbits - 1; // -1 for sign bit

    if (f == 0.0) return 0; // get this special case out of the way

    // check sign and begin normalization
    if (f < 0) { sign = 1; fnorm = -f; }
    else { sign = 0; fnorm = f; }

    // get the normalized form of f and track the exponent
    shift = 0;
    while(fnorm >= 2.0) { fnorm /= 2.0; shift++; }
    while(fnorm < 1.0) { fnorm *= 2.0; shift--; }
    fnorm = fnorm - 1.0;

    // calculate the binary form (non-float) of the significand data
    significand = fnorm * ((1LL<<significandbits) + 0.5f);

    // get the biased exponent
    exp = shift + ((1<<(expbits-1)) - 1); // shift + bias

    // return the final answer
    return (sign<<(bits-1)) | (exp<<(bits-expbits-1)) | significand;
}

long double unpack754(uint64_t i, unsigned bits, unsigned expbits)
{
    long double result;
    long long shift;
    unsigned bias;
    unsigned significandbits = bits - expbits - 1; // -1 for sign bit

    if (i == 0) return 0.0;

    // pull the significand
```

[29] http://en.wikipedia.org/wiki/IEEE_754
[30] http://beej.us/guide/bgnet/examples/ieee754.c

```
    result = (i&((1LL<<significandbits)-1)); // mask
    result /= (1LL<<significandbits); // convert back to float
    result += 1.0f; // add the one back on

    // deal with the exponent
    bias = (1<<(expbits-1)) - 1;
    shift = ((i>>significandbits)&((1LL<<expbits)-1)) - bias;
    while(shift > 0) { result *= 2.0; shift--; }
    while(shift < 0) { result /= 2.0; shift++; }

    // sign it
    result *= (i>>(bits-1))&1? -1.0: 1.0;

    return result;
}
```

I put some handy macros up there at the top for packing and unpacking 32-bit (probably a `float`) and 64-bit (probably a `double`) numbers, but the **`pack754()`** function could be called directly and told to encode *`bits`*-worth of data (*`expbits`* of which are reserved for the normalized number's exponent.)

Here's sample usage:

```

#include <stdio.h>
#include <stdint.h> // defines uintN_t types
#include <inttypes.h> // defines PRIx macros

int main(void)
{
    float f = 3.1415926, f2;
    double d = 3.14159265358979323, d2;
    uint32_t fi;
    uint64_t di;

    fi = pack754_32(f);
    f2 = unpack754_32(fi);

    di = pack754_64(d);
    d2 = unpack754_64(di);

    printf("float before : %.7f\n", f);
    printf("float encoded: 0x%08" PRIx32 "\n", fi);
    printf("float after  : %.7f\n\n", f2);

    printf("double before : %.20lf\n", d);
    printf("double encoded: 0x%016" PRIx64 "\n", di);
    printf("double after  : %.20lf\n", d2);

    return 0;
}
```

The above code produces this output:

```
float before : 3.1415925
float encoded: 0x40490FDA
float after  : 3.1415925

double before : 3.14159265358979311600
double encoded: 0x400921FB54442D18
double after  : 3.14159265358979311600
```

Another question you might have is how do you pack `struct`s? Unfortunately for you, the compiler is free to put padding all over the place in a `struct`, and that means you can't portably send the whole thing over the wire in one chunk. (Aren't you getting sick of hearing “can't do this”, “can't do that”? Sorry! To quote a friend, “Whenever anything goes wrong, I always blame Microsoft.” This one might not be Microsoft's fault, admittedly, but my friend's statement is completely true.)

Back to it: the best way to send the `struct` over the wire is to pack each field independently and then unpack them into the `struct` when they arrive on the other side.

That's a lot of work, is what you're thinking. Yes, it is. One thing you can do is write a helper function to help pack the data for you. It'll be fun! Really!

In the book “The Practice of Programming[31]” by Kernighan and Pike, they implement **`printf()`**-like functions called **`pack()`** and **`unpack()`** that do exactly this. I'd link to them, but apparently those functions aren't online with the rest of the source from the book.

(The Practice of Programming is an excellent read. Zeus saves a kitten every time I recommend it.)

At this point, I'm going to drop a pointer to the BSD-licensed Typed Parameter Language C API[32] which I've never used, but looks completely respectable. Python and Perl programmers will want to check out their language's **`pack()`** and **`unpack()`** functions for accomplishing the same thing. And Java has a big-ol' Serializable interface that can be used in a similar way.

But if you want to write your own packing utility in C, K&P's trick is to use variable argument lists to make **`printf()`**-like functions to build the packets. Here's a version I cooked up[33] on my own based on that which hopefully will be enough to give you an idea of how such a thing can work.

(This code references the **`pack754()`** functions, above. The **`packi*()`** functions operate like the familiar **`htons()`** family, except they pack into a `char` array instead of another integer.)

```
#include <ctype.h>
#include <stdarg.h>
#include <string.h>
#include <stdint.h>
#include <inttypes.h>

// various bits for floating point types--
// varies for different architectures
typedef float float32_t;
typedef double float64_t;

/*
** packi16() -- store a 16-bit int into a char buffer (like htons())
*/
void packi16(unsigned char *buf, unsigned int i)
{
    *buf++ = i>>8; *buf++ = i;
}

/*
** packi32() -- store a 32-bit int into a char buffer (like htonl())
*/
void packi32(unsigned char *buf, unsigned long i)
{
    *buf++ = i>>24; *buf++ = i>>16;
    *buf++ = i>>8;  *buf++ = i;
}

/*
** unpacki16() -- unpack a 16-bit int from a char buffer (like ntohs())
*/
unsigned int unpacki16(unsigned char *buf)
{
    return (buf[0]<<8) | buf[1];
}

/*
** unpacki32() -- unpack a 32-bit int from a char buffer (like ntohl())
*/
unsigned long unpacki32(unsigned char *buf)
{
    return (buf[0]<<24) | (buf[1]<<16) | (buf[2]<<8) | buf[3];
}
```

[31] http://cm.bell-labs.com/cm/cs/tpop/
[32] http://tpl.sourceforge.net/
[33] http://beej.us/guide/bgnet/examples/pack2.c

```
/*
** pack() -- store data dictated by the format string in the buffer
**
**  h - 16-bit              l - 32-bit
**  c - 8-bit char          f - float, 32-bit
**  s - string (16-bit length is automatically prepended)
*/
int32_t pack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int16_t h;
    int32_t l;
    int8_t c;
    float32_t f;
    char *s;
    int32_t size = 0, len;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-bit
            size += 2;
            h = (int16_t)va_arg(ap, int); // promoted
            packi16(buf, h);
            buf += 2;
            break;

        case 'l': // 32-bit
            size += 4;
            l = va_arg(ap, int32_t);
            packi32(buf, l);
            buf += 4;
            break;

        case 'c': // 8-bit
            size += 1;
            c = (int8_t)va_arg(ap, int); // promoted
            *buf++ = (c>>0)&0xff;
            break;

        case 'f': // float
            size += 4;
            f = (float32_t)va_arg(ap, double); // promoted
            l = pack754_32(f); // convert to IEEE 754
            packi32(buf, l);
            buf += 4;
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = strlen(s);
            size += len + 2;
            packi16(buf, len);
            buf += 2;
            memcpy(buf, s, len);
            buf += len;
            break;
        }
    }

    va_end(ap);

    return size;
}

/*
```

```
** unpack() -- unpack data dictated by the format string into the buffer
*/
void unpack(unsigned char *buf, char *format, ...)
{
    va_list ap;
    int16_t *h;
    int32_t *l;
    int32_t pf;
    int8_t *c;
    float32_t *f;
    char *s;
    int32_t len, count, maxstrlen=0;

    va_start(ap, format);

    for(; *format != '\0'; format++) {
        switch(*format) {
        case 'h': // 16-bit
            h = va_arg(ap, int16_t*);
            *h = unpacki16(buf);
            buf += 2;
            break;

        case 'l': // 32-bit
            l = va_arg(ap, int32_t*);
            *l = unpacki32(buf);
            buf += 4;
            break;

        case 'c': // 8-bit
            c = va_arg(ap, int8_t*);
            *c = *buf++;
            break;

        case 'f': // float
            f = va_arg(ap, float32_t*);
            pf = unpacki32(buf);
            buf += 4;
            *f = unpack754_32(pf);
            break;

        case 's': // string
            s = va_arg(ap, char*);
            len = unpacki16(buf);
            buf += 2;
            if (maxstrlen > 0 && len > maxstrlen) count = maxstrlen - 1;
            else count = len;
            memcpy(s, buf, count);
            s[count] = '\0';
            buf += len;
            break;

        default:
            if (isdigit(*format)) { // track max str len
                maxstrlen = maxstrlen * 10 + (*format-'0');
            }
        }

        if (!isdigit(*format)) maxstrlen = 0;
    }

    va_end(ap);
}
```

And here is a demonstration program[34] of the above code that packs some data into *`buf`* and then unpacks it into variables. Note that when calling **`unpack()`** with a string argument (format specifier "`s`"), it's wise to put a maximum length count in front of it to prevent a buffer overrun, e.g. "`96s`". Be wary when unpacking data you get over the network—a malicious user might send badly-constructed packets in an effort to attack your system!

```
#include <stdio.h>

// various bits for floating point types--
// varies for different architectures
typedef float float32_t;
typedef double float64_t;

int main(void)
{
    unsigned char buf[1024];
    int8_t magic;
    int16_t monkeycount;
    int32_t altitude;
    float32_t absurdityfactor;
    char *s = "Great unmitigated Zot!  You've found the Runestaff!";
    char s2[96];
    int16_t packetsize, ps2;

    packetsize = pack(buf, "chhlsf", (int8_t)'B', (int16_t)0, (int16_t)37,
            (int32_t)-5, s, (float32_t)-3490.6677);
    packi16(buf+1, packetsize); // store packet size in packet for kicks

    printf("packet is %" PRId32 " bytes\n", packetsize);

    unpack(buf, "chhl96sf", &magic, &ps2, &monkeycount, &altitude, s2,
        &absurdityfactor);

    printf("'%c' %" PRId32" %" PRId16 " %" PRId32
            " \"%s\" %f\n", magic, ps2, monkeycount,
            altitude, s2, absurdityfactor);

    return 0;
}
```

Whether you roll your own code or use someone else's, it's a good idea to have a general set of data packing routines for the sake of keeping bugs in check, rather than packing each bit by hand each time.

When packing the data, what's a good format to use? Excellent question. Fortunately, RFC 4506[35], the External Data Representation Standard, already defines binary formats for a bunch of different types, like floating point types, integer types, arrays, raw data, etc. I suggest conforming to that if you're going to roll the data yourself. But you're not obligated to. The Packet Police are not right outside your door. At least, I don't *think* they are.

In any case, encoding the data somehow or another before you send it is the right way of doing things!

7.5. Son of Data Encapsulation

What does it really mean to encapsulate data, anyway? In the simplest case, it means you'll stick a header on there with either some identifying information or a packet length, or both.

What should your header look like? Well, it's just some binary data that represents whatever you feel is necessary to complete your project.

Wow. That's vague.

Okay. For instance, let's say you have a multi-user chat program that uses `SOCK_STREAM`s. When a user types ("says") something, two pieces of information need to be transmitted to the server: what was said and who said it.

[34] `http://beej.us/guide/bgnet/examples/pack2.c`
[35] `http://tools.ietf.org/html/rfc4506`

So far so good? “What's the problem?” you're asking.

The problem is that the messages can be of varying lengths. One person named “tom” might say, “Hi”, and another person named “Benjamin” might say, “Hey guys what is up?”

So you **`send()`** all this stuff to the clients as it comes in. Your outgoing data stream looks like this:

```
t o m H i B e n j a m i n H e y g u y s w h a t i s u p ?
```

And so on. How does the client know when one message starts and another stops? You could, if you wanted, make all messages the same length and just call the **`sendall()`** we implemented, above. But that wastes bandwidth! We don't want to **`send()`** 1024 bytes just so “tom” can say “Hi”.

So we *encapsulate* the data in a tiny header and packet structure. Both the client and server know how to pack and unpack (sometimes referred to as “marshal” and “unmarshal”) this data. Don't look now, but we're starting to define a *protocol* that describes how a client and server communicate!

In this case, let's assume the user name is a fixed length of 8 characters, padded with `'\0'`. And then let's assume the data is variable length, up to a maximum of 128 characters. Let's have a look a sample packet structure that we might use in this situation:

1. `len` (1 byte, unsigned)—The total length of the packet, counting the 8-byte user name and chat data.
2. `name` (8 bytes)—The user's name, NUL-padded if necessary.
3. `chatdata` (*n*-bytes)—The data itself, no more than 128 bytes. The length of the packet should be calculated as the length of this data plus 8 (the length of the name field, above).

Why did I choose the 8-byte and 128-byte limits for the fields? I pulled them out of the air, assuming they'd be long enough. Maybe, though, 8 bytes is too restrictive for your needs, and you can have a 30-byte name field, or whatever. The choice is up to you.

Using the above packet definition, the first packet would consist of the following information (in hex and ASCII):

```
   0A     74 6F 6D 00 00 00 00 00      48 69
(length)  T  o  m    (padding)         H  i
```

And the second is similar:

```
   18     42 65 6E 6A 61 6D 69 6E      48 65 79 20 67 75 79 73 20 77 ...
(length)  B  e  n  j  a  m  i  n       H  e  y     g  u  y  s     w  ...
```

(The length is stored in Network Byte Order, of course. In this case, it's only one byte so it doesn't matter, but generally speaking you'll want all your binary integers to be stored in Network Byte Order in your packets.)

When you're sending this data, you should be safe and use a command similar to **`sendall()`**, above, so you know all the data is sent, even if it takes multiple calls to **`send()`** to get it all out.

Likewise, when you're receiving this data, you need to do a bit of extra work. To be safe, you should assume that you might receive a partial packet (like maybe we receive “`18 42 65 6E 6A`” from Benjamin, above, but that's all we get in this call to **`recv()`**). We need to call **`recv()`** over and over again until the packet is completely received.

But how? Well, we know the number of bytes we need to receive in total for the packet to be complete, since that number is tacked on the front of the packet. We also know the maximum packet size is 1+8+128, or 137 bytes (because that's how we defined the packet.)

There are actually a couple things you can do here. Since you know every packet starts off with a length, you can call **`recv()`** just to get the packet length. Then once you have that, you can call it again specifying exactly the remaining length of the packet (possibly repeatedly to get all the data) until you have the complete packet. The advantage of this method is that you only need a buffer large enough for one packet, while the disadvantage is that you need to call **`recv()`** at least twice to get all the data.

Another option is just to call **`recv()`** and say the amount you're willing to receive is the maximum number of bytes in a packet. Then whatever you get, stick it onto the back of a buffer, and finally check to see if the packet is complete. Of course, you might get some of the next packet, so you'll need to have room for that.

What you can do is declare an array big enough for two packets. This is your work array where you will reconstruct packets as they arrive.

Every time you **`recv()`** data, you'll append it into the work buffer and check to see if the packet is complete. That is, the number of bytes in the buffer is greater than or equal to the length specified in the header (+1, because the length in the header doesn't include the byte for the length itself.) If the number of bytes in the buffer is less than 1, the packet is not complete, obviously. You have to make a special case for this, though, since the first byte is garbage and you can't rely on it for the correct packet length.

Once the packet is complete, you can do with it what you will. Use it, and remove it from your work buffer.

Whew! Are you juggling that in your head yet? Well, here's the second of the one-two punch: you might have read past the end of one packet and onto the next in a single **`recv()`** call. That is, you have a work buffer with one complete packet, and an incomplete part of the next packet! Bloody heck. (But this is why you made your work buffer large enough to hold *two* packets—in case this happened!)

Since you know the length of the first packet from the header, and you've been keeping track of the number of bytes in the work buffer, you can subtract and calculate how many of the bytes in the work buffer belong to the second (incomplete) packet. When you've handled the first one, you can clear it out of the work buffer and move the partial second packet down the to front of the buffer so it's all ready to go for the next **`recv()`**.

(Some of you readers will note that actually moving the partial second packet to the beginning of the work buffer takes time, and the program can be coded to not require this by using a circular buffer. Unfortunately for the rest of you, a discussion on circular buffers is beyond the scope of this article. If you're still curious, grab a data structures book and go from there.)

I never said it was easy. Ok, I did say it was easy. And it is; you just need practice and pretty soon it'll come to you naturally. By Excalibur I swear it!

7.6. Broadcast Packets—Hello, World!

So far, this guide has talked about sending data from one host to one other host. But it is possible, I insist, that you can, with the proper authority, send data to multiple hosts *at the same time*!

With UDP (only UDP, not TCP) and standard IPv4, this is done through a mechanism called *broadcasting*. With IPv6, broadcasting isn't supported, and you have to resort to the often superior technique of *multicasting*, which, sadly I won't be discussing at this time. But enough of the starry-eyed future—we're stuck in the 32-bit present.

But wait! You can't just run off and start broadcasting willy-nilly; You have to set the socket option `SO_BROADCAST` before you can send a broadcast packet out on the network. It's like a one of those little plastic covers they put over the missile launch switch! That's just how much power you hold in your hands!

But seriously, though, there is a danger to using broadcast packets, and that is: every system that receives a broadcast packet must undo all the onion-skin layers of data encapsulation until it finds out what port the data is destined to. And then it hands the data over or discards it. In either case, it's a lot of work for each machine that receives the broadcast packet, and since it is all of them on the local network, that could be a lot of machines doing a lot of unnecessary work. When the game Doom first came out, this was a complaint about its network code.

Now, there is more than one way to skin a cat... wait a minute. Is there really more than one way to skin a cat? What kind of expression is that? Uh, and likewise, there is more than one way to send a broadcast packet. So, to get to the meat and potatoes of the whole thing: how do you specify the destination address for a broadcast message? There are two common ways:

1. Send the data to a specific subnet's broadcast address. This is the subnet's network number with all one-bits set for the host portion of the address. For instance, at home my network is 192.168.1.0, my netmask is 255.255.255.0, so the last byte of the address is my host number (because the first three bytes, according to the netmask, are the network number). So my broadcast address is 192.168.1.255. Under Unix, the **ifconfig** command will actually give you all this data. (If you're curious, the bitwise logic to get your broadcast address is *`network_number`* OR (NOT *`netmask`*).) You can send this type of broadcast packet to remote

networks as well as your local network, but you run the risk of the packet being dropped by the destination's router. (If they didn't drop it, then some random smurf could start flooding their LAN with broadcast traffic.)

2. Send the data to the “global” broadcast address. This is 255.255.255.255, aka `INADDR_BROADCAST`. Many machines will automatically bitwise AND this with your network number to convert it to a network broadcast address, but some won't. It varies. Routers do not forward this type of broadcast packet off your local network, ironically enough.

So what happens if you try to send data on the broadcast address without first setting the `SO_BROADCAST` socket option? Well, let's fire up good old **talker** and **listener** and see what happens.

```
$ talker 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ talker 192.168.1.255 foo
sendto: Permission denied
$ talker 255.255.255.255 foo
sendto: Permission denied
```

Yes, it's not happy at all...because we didn't set the `SO_BROADCAST` socket option. Do that, and now you can **`sendto()`** anywhere you want!

In fact, that's the *only difference* between a UDP application that can broadcast and one that can't. So let's take the old **talker** application and add one section that sets the `SO_BROADCAST` socket option. We'll call this program *`broadcaster.c`*[36]:

```
/*
** broadcaster.c -- a datagram "client" like talker.c, except
**                  this one can broadcast
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define SERVERPORT 4950 // the port users will be connecting to

int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in their_addr; // connector's address information
    struct hostent *he;
    int numbytes;
    int broadcast = 1;
    //char broadcast = '1'; // if that doesn't work, try this

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

    if ((he=gethostbyname(argv[1])) == NULL) {  // get the host info
        perror("gethostbyname");
        exit(1);
    }

    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
```

[36] `http://beej.us/guide/bgnet/examples/broadcaster.c`

```
        perror("socket");
        exit(1);
    }

    // this call is what allows broadcast packets to be sent:
    if (setsockopt(sockfd, SOL_SOCKET, SO_BROADCAST, &broadcast,
        sizeof broadcast) == -1) {
        perror("setsockopt (SO_BROADCAST)");
        exit(1);
    }

    their_addr.sin_family = AF_INET;     // host byte order
    their_addr.sin_port = htons(SERVERPORT); // short, network byte order
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    memset(their_addr.sin_zero, '\0', sizeof their_addr.sin_zero);

    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
            (struct sockaddr *)&their_addr, sizeof their_addr)) == -1) {
        perror("sendto");
        exit(1);
    }

    printf("sent %d bytes to %s\n", numbytes,
        inet_ntoa(their_addr.sin_addr));

    close(sockfd);

    return 0;
}
```

What's different between this and a “normal” UDP client/server situation? Nothing! (With the exception of the client being allowed to send broadcast packets in this case.) As such, go ahead and run the old UDP **listener** program in one window, and **broadcaster** in another. You should be now be able to do all those sends that failed, above.

```
$ broadcaster 192.168.1.2 foo
sent 3 bytes to 192.168.1.2
$ broadcaster 192.168.1.255 foo
sent 3 bytes to 192.168.1.255
$ broadcaster 255.255.255.255 foo
sent 3 bytes to 255.255.255.255
```

And you should see **listener** responding that it got the packets. (If **listener** doesn't respond, it could be because it's bound to an IPv6 address. Try changing the `AF_UNSPEC` in *`listener.c`* to `AF_INET` to force IPv4.)

Well, that's kind of exciting. But now fire up **listener** on another machine next to you on the same network so that you have two copies going, one on each machine, and run **broadcaster** again with your broadcast address... Hey! Both **listener**s get the packet even though you only called **`sendto()`** once! Cool!

If the **listener** gets data you send directly to it, but not data on the broadcast address, it could be that you have a firewall on your local machine that is blocking the packets. (Yes, Pat and Bapper, thank you for realizing before I did that this is why my sample code wasn't working. I told you I'd mention you in the guide, and here you are. So *nyah*.)

Again, be careful with broadcast packets. Since every machine on the LAN will be forced to deal with the packet whether it **`recvfrom()`**s it or not, it can present quite a load to the entire computing network. They are definitely to be used sparingly and appropriately.