Source: http://www.gilesthomas.com/2013/08/writing-a-reverse-proxyloadbalancer-from-the-ground-up-in-c-part-0/

Writing a reverse proxy/loadbalancer from the ground up in C, part 0: introduction

8 August 2013
0 Comments

We're spending a lot of time on nginx configuration at PythonAnywhere. We're a platform-as-a-service, and a lot of people host their websites with us, so it's important that we have a reliable load-balancer to receive all of the incoming web traffic and appropriately distribute it around backend web-server nodes.

nginx is a fantastic, possibly unbeatable tool for this. It's fast, reliable, and lightweight in terms of CPU resources. We're using the OpenResty variant of it, which adds a number of useful modules — most importantly for us, one for Lua scripting, which means that we can dynamically work out where to send traffic as the hits come in.

It's also quite simple to configure at a basic level. You want all incoming requests for site X to go to backend Y? Just write something like this:

```
server {
    server_name X
    listen 80;

    location / {
        proxy_set_header Host $host;
        proxy_pass Y;
    }
}
```

Simple enough. Lua scripting is pretty easy to add — you just put an extra directive before the `proxy_pass` that provides some Lua code to run, and then variables you set in the code can be accessed from the `proxy_pass`.

But there are many more complicated options. `worker_connections`, `tcp_nopush`, `sendfile`, `types_hash_max_size`... Some are reasonably easy to understand with a certain amount of reading, some are harder.

I'm a big believer that the best way to understand something complex is to try to build your own simple version of it. So, in my copious free time, I'm going to start putting together a simple loadbalancer in C. The aim isn't to rewrite nginx or OpenResty; it's to write enough equivalent functionality that I can better understand what they are really doing under the hood, in the same way as writing a compiler for a toy language gives you a better understanding of how proper compilers work. I'll get a good grasp on some underlying OS concepts that I have only a vague appreciation of now. It's also going to be quite fun coding in C again. I've not really written any since 1997.

Anyway, I'll document the steps I take here on this blog; partly because there's a faint chance that it might be interesting to other experienced Python programmers whose C is rusty or nonexistent and want to get a view under the hood, but mostly because the best way to be sure you really understand it is to try to explain it to other people.

I hope it'll be interesting!

Here's a link to the first post in the series: Writing a reverse proxy/loadbalancer from the ground up in C, part 1: a trivial one-shot proxy

Category: Programming
Post navigation
Writing a reverse proxy/loadbalancer from the ground up in C, part 1: a trivial single-threaded proxy →

Writing a reverse proxy/loadbalancer from the ground up in C, part 1: a trivial single-threaded proxy

12 August 2013
0 Comments

This is the first step along my road to building a simple C-based reverse proxy/loadbalancer so that I can understand how nginx/OpenResty works — more explanation here. It's called `rsp`, for Really Simple Proxy. This version listens for connections on a particular port, specified on the command line; when one is made it sends the request down to a backend — another server with an associated port, also specified on the command line — and sends whatever comes back from the backend back to the person who made the original connection. It can only handle one connection at a time — while it's handling one, it just queues up others, and it handles them in turn. This will, of course, change later.

I'm posting this in the hope that it might help people who know Python, and some basic C, but want to learn more about how the OS-level networking stuff works. I'm also vaguely hoping that any readers who code in C day to day might take a look and tell me what I'm doing wrong :-)

The code that I'll be describing is hosted on GitHub as a project called rsp, for "Really Simple Proxy". It's MIT licensed, and the version of it I'll be walking through in this blog post is as of commit f214f5a. I'll copy and paste the code that I'm describing into this post anyway, so if you're following along there's no need to do any kind of complicated checkout.

The repository has this structure:

```
+- README.md
+- LICENSE.md
+- setup-env.sh
+- run_integration_tests
+- promote_to_live
+- handle_integration_error
+- .gitignore
+- fts
   \--- test_can_proxy_http_request_to_backend.py
+- src
   \--- Makefile
```

```
    +-- rsp.c
```

`README.md`, `LICENSE.md` and `.gitignore` are pretty self-explanatory.

`setup-env.sh` contains a few shell commands that, when run on a fresh Ubuntu machine, will install all of the dependencies for compiling rsp.

`fts`, short for Functional Tests, contains one Python script; this creates a simple Python web server on a specific port on localhost, then starts rsp configured so that all requests that come in on its own port get forwarded to that backend. It then sends a request to rsp and checks that the response that comes back is the one from the backend, then does another to make sure it can handle multiple requests. This is the most trivial test I could think of for a first cut at rsp, and this blog post contains an explanation of how the minimal C code to make that test pass works. Over time, I'll add more test scripts to the repository to check each incremental improvement in rsp's functionality. Naturally, I'm writing all of this test-first (though I'm too lazy to write unit tests right now — this may well come back to bite me later).

`src` contains a `Makefile` that knows how to build rsp, and the code for the proxy itself, `rsp.c`. As you might expect, most of the rest of this post will focus on the latter.

Finally, there's `run_integration_tests` (which runs all of the Python tests in `fts`), `promote_to_live`, which pushes the repository it's in to `origin/master`, and `handle_integration_error`. This is boilerplate code for the not-quite-continuous-integration system I use for my own projects, leibniz — basically just a git repository setup and a hook that makes it impossible for me to push stuff that doesn't pass functional tests to GitHub. You can probably ignore all of those, though if you do check out the repository then `run_integration_tests` is a convenient way to run the FTs in one go.

So, that's the structure. I won't go into a description of how the Python functional test works, as I expect most readers here will understand it pretty well. And I'm assuming that you have at least a nodding acquaintance with Makefiles, so I won't explain that bit. So, on to the C code!

`rsp.c` contains code for an incredibly basic proxy. It's actually best understood by working from the top down, so let's go. First, some pretty standard header includes:

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

A couple of constants for later use:

```
#define MAX_LISTEN_BACKLOG 1
#define BUFFER_SIZE 4096
```

And on to our first function — one that, given a connection to a client (ie. browser) that's hit the proxy, connects to a backend, sends the client's request to it, then gets the response from the backend and sends it back to the client. This is simpler than the code that handles the process of listening for incoming connections, so it's worth running through it first.

```
void handle_client_connection(int client_socket_fd,
                              char *backend_host,
                              char *backend_port_str)
{
```

The first parameter is an integer *file descriptor* for the client socket connection (which has already been established by the time we get here). A file descriptor is the low-level C way of describing an open file, or a file-like object like a socket connection. The kernel has some kind of big array, where each item in the array describes all of the inner details about an open file-like thing, and the integer file descriptor is ultimately an index into that array.

We also take a string specifying the address of the backend we're going to connect to, and an another specifying the port on the backend. We accept a string for the port (rather than an integer) because one of the the OS system calls we're going to use accepts string *services* rather than ports — the most immediate advantage of which is that we can just specify `"http"` and let it work out that means port 80. Hardly a huge win, but neat enough.

Right, next our local variable definitions — these need to go at the start of the function — an annoying requirement in C [Update: turns out that it hasn't been a requirement since the 1999 C standard, so later posts update this] — but we'll backtrack to talk about what each one's used for as we use them.

```
    struct addrinfo hints;
    struct addrinfo *addrs;
    struct addrinfo *addrs_iter;
    int getaddrinfo_error;

    int backend_socket_fd;

    char buffer[BUFFER_SIZE];
    int bytes_read;
```

So now it's time to actually do something. Our first step is to convert the hostname/service descriptor strings we have into something that we can use to make a network connection. Historically one might have used the `gethostbyname` system call, but that's apparently frowned upon these days; it's non-reentrant and makes it hard to support both IPv4 and IPv6. The hip way to get host information is by using `getaddrinfo`, so that's what we'll do.

`getaddrinfo` needs three things; the hostname and service we're connecting to, and some *hints* telling us what kind of thing we're interested in hearing about — for example, in our case we only want to know about addresses of machines that can handle streaming sockets which we can read to and write from like files, rather than datagrams where we send lumps of data back and forth, one lump at a time.

We already have the hostname and service passed in as arguments, so our first step is to set up a structure to represent these hints. We have the local variable that was defined earlier as:

```
    struct addrinfo hints;
```

So we need to set some values on it. In C code, a struct that's allocated on the stack as a local variable like that has completely undefined contents, which means that we need to clear it out by setting everything in it to zeros using `memset`, like this:

```
    memset(&hints, 0, sizeof(struct addrinfo));
```

…and once that's done, we can fill in the things we're interested in:

```
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
```

`AF_UNSPEC` means that we're happy with either IPv4 or IPv6 results, and `SOCK_STREAM` means that we want something that supports streaming sockets.

Now we've set up our hints structure, we can call `getaddrinfo`. It returns zero if it succeeds, or an error code if it doesn't, and the real address information results are returned in a parameter — we pass a pointer to a pointer to a `struct addrinfo` in, and it puts a pointer to the first in a list of results into the pointer that the pointer points to. Lovely, pointers to pointers and we're only a dozen lines in…

```
    getaddrinfo_error = getaddrinfo(backend_host, backend_port_str, &hints, 
&addrs);
    if (getaddrinfo_error != 0) {
        fprintf(stderr, "Couldn't find backend: %s\n", 
gai_strerror(getaddrinfo_error));
        exit(1);
    }
```

So now we have, in our variable `addrs`, a pointer to the first item in a linked list of possible addresses that we can connect to. Each item in the list has the associated kinds of family (IPv4 or IPv6, basically), socket type (restricted to streaming sockets because that's what we asked for in the hints), and protocol. We want to find one that we can connect to, so we loop through them:

```
    for (addrs_iter = addrs;
         addrs_iter != NULL;
         addrs_iter = addrs_iter->ai_next)
    {
```

For each one, we try to create a socket using the system `socket` call, passing in the details of the address that we're trying, and if that fails we move on to the next one:

```
        backend_socket_fd = socket(addrs_iter->ai_family,
                                   addrs_iter->ai_socktype,
                                   addrs_iter->ai_protocol);
        if (backend_socket_fd == -1) {
            continue;
        }
```

If it succeeded, we try to `connect` to the address using that socket, and if that succeeds we break out of the loop:

```
        if (connect(backend_socket_fd,
                    addrs_iter->ai_addr,
                    addrs_iter->ai_addrlen) != -1) {
            break;
        }
```

If, on the other hand, the connect failed, we close the socket (to tidy up) and move on to the next one in the loop:

```
        close(backend_socket_fd);
    }
```

Once we’re out of the loop, we need to check if we ever managed to do a successful socket creation and connect — if we don’t, we bomb out.

```
    if (addrs_iter == NULL) {
        fprintf(stderr, "Couldn't connect to backend");
        exit(1);
    }
```

Otherwise, we free the list of addresses that we got back from `getaddrinfo` (ah, the joys of manual memory management…)

```
    freeaddrinfo(addrs);
```

…and finally we do a really simple bit of code to actually do the proxying. For this first cut, I’ve assumed that a single `read` on the file descriptor that is connected to the client is enough to pull down all of the client’s headers, and that we never want to send anything from the client to the backend after those headers. So we just do one read from the client, and send everything we get from that read down to the backend:

```
    bytes_read = read(client_socket_fd, buffer, BUFFER_SIZE);
    write(backend_socket_fd, buffer, bytes_read);
```

…then we just go into a loop that reads everything it can from the backend until the `read` call returns zero bytes (which means end-of-file) and `write`s everything that it reads down the socket to the client.

```
    while (bytes_read = read(backend_socket_fd, buffer, BUFFER_SIZE)) {
        write(client_socket_fd, buffer, bytes_read);
    }
```

Then we close the client socket, and that’s the total of our client-handling code.

```
    close(client_socket_fd);
}
```

The code we use to create a socket to listen for incoming client connections and pass them off to the function we’ve just gone through is actually pretty similar, but with a few interesting twists. It lives (as you might expect) in the program’s `main` function:

```
int main(int argc, char *argv[]) {
```

…which we start off with our local variables again:

```
    char *server_port_str;
    char *backend_addr;
    char *backend_port_str;

    struct addrinfo hints;
    struct addrinfo *addrs;
    struct addrinfo *addr_iter;
    int getaddrinfo_error;

    int server_socket_fd;
    int client_socket_fd;

    int so_reuseaddr;
```

The first step is just to check that we have the right number of command-line arguments and to put them into some meaningfully-named variables:

```
    if (argc != 4) {
        fprintf(stderr,
                "Usage: %s   \n",
                argv[0]);
        exit(1);
    }
    server_port_str = argv[1];
    backend_addr = argv[2];
    backend_port_str = argv[3];
```

Now, the next step is to get the address of localhost. We do that with the same kind of `getaddrinfo` call that we did on the client-connection handling side, but this time we add one extra value to the hints, and pass in NULL as the first parameter to the call:

```
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;

    getaddrinfo_error = getaddrinfo(NULL, server_port_str, &hints, &addrs);
```

The `ai_flags` structure member being set to `AI_PASSIVE`, combined with the `NULL` first parameter, tells `getaddrinfo` that we want to be able to run a server socket on this address — we want to be able to listen for incoming connections, accept them, and handle them.

Once we’ve got the list of appropriate addresses, we iterate through them again, and for each one we create a socket like we did before, but now instead of trying to `connect` to them to make an outgoing connection, we try to `bind` so that we can accept incoming connections:

```
    for (addr_iter = addrs; addr_iter != NULL; addr_iter = addr_iter->ai_next) {
        server_socket_fd = socket(addr_iter->ai_family,
                                  addr_iter->ai_socktype,
                                  addr_iter->ai_protocol);
        if (server_socket_fd == -1) {
            continue;
        }

        so_reuseaddr = 1;
        setsockopt(server_socket_fd, SOL_SOCKET, SO_REUSEADDR, &so_reuseaddr, 
sizeof(so_reuseaddr));

        if (bind(server_socket_fd,
                 addr_iter->ai_addr,
                 addr_iter->ai_addrlen) == 0)
        {
            break;
        }

        close(server_socket_fd);
    }
```

Binding basically says “I own this socket and I’m going to listen for incoming connections on it”.

There’s also a second little tweak in that code — the call to `setsockopt`. This is useful when you’re working on something like this. The main loop for rsp never exits, so of course you need to use control-C or `kill` to quit it. The problem is that this means we never close our server socket, so the operating system is never told "we're not listening on this port any more". The OS has timeouts, and if it notices that the program that was listening on a particular port has gone away, it will free it up for use by other programs. But this can take a few minutes, so if you're debugging and starting and stopping the server frequently, you can wind up with errors trying to bind when you start it. The `SO_REUSEADDR` flag that we're associating with the socket is just a way of saying "I'm happy to share this socket with other people", which mitigates this problem.

Anyway, once we've bound (or if we were unable to bind) then we handle errors and tidy up just as we did before:

```
    if (addr_iter == NULL) {
        fprintf(stderr, "Couldn't bind\n");
        exit(1);
    }

    freeaddrinfo(addrs);
```

Finally, we need to mark the socket so that it's "passive" — that is, it's one that will listen for incoming connections instead of making outgoing connections. This is done using the slightly-confusingly-named `listen` call, which doesn't actually listen for anything but simply marks the socket appropriately:

```
    listen(server_socket_fd, MAX_LISTEN_BACKLOG);
```

The second parameter says that we're going to allow a certain number of incoming connections to build up while we're handling stuff.

Now we've got our server socket ready, and the next code is the endless loop that actually does the proxying.

```
    while (1) {
```

In it, we need to wait for incoming connections, using `accept`, which blocks until someone connects to us.

```
        client_socket_fd = accept(server_socket_fd, NULL, NULL);
        if (client_socket_fd == -1) {
            perror("Could not accept");
            exit(1);
        }
```

`accept` takes three parameters; the server socket's file descriptor (which you'd expect given that it needs to know what to work on) and also some pointers into which it can put information about the incoming client connection. We're likely to need something like that later, but right now we don't need it so we won't worry about it — passing in `NULL` is the appropriate way to tell `accept` that we don't care.

After the `accept` has been done, we have a file descriptor that describes the client connection, so we can hand off to the function that we described earlier:

```
        handle_client_connection(client_socket_fd, backend_addr,
backend_port_str);
```

And off we go, around the loop again:

```
    }

}
```

Phew. So that was a bit harder than it would have been in Python. But not too scary. Hopefully it was all reasonably clear — and if it wasn't, please let me know in the comments. And if any C experts have been reading — thank you for putting up with the slow pace, and if you have any suggestions then I'd love to hear them!

The next step, I think, is to make this a more useful proxy by making it no longer single-shot, and instead accept multiple simultaneous client connections and proxy them back to the backend. We can then add multiple backends, and start looking at selecting which one to proxy to based on the `Host` HTTP header. And, as I'm aiming to produce a cut-down version of OpenResty, then adding some Lua scripting would help too.

But multiple connections first. Here's how I handle them.

Some acknowledgements: obviously the Linux man pages at linux.die.net were invaluable in putting this together. An earlier version of this proxy used code from this socket server example at tutorialspoint and its associated socket client example, but the code there (on examination) turned out to use quite a few deprecated functions, so in fact most of it wound up getting rewritten using the man page for `getaddrinfo`.

Category: Programming
Post navigation
← Writing a reverse proxy/loadbalancer from the ground up in C, part 0: introduction Writing a reverse proxy/loadbalancer from the ground up in C, part 2: handling multiple connections with epoll →

Writing a reverse proxy/loadbalancer from the ground up in C, part 2: handling multiple connections with epoll

7 September 2013
0 Comments

This is the second step along my road to building a simple C-based reverse proxy/loadbalancer so that I can understand how nginx/OpenResty works — more background here. Here's a link to the first part, where I showed the basic networking code required to write a proxy that could handle one incoming connection at a time and connect it with a single backend.

This (rather long) post describes a version that uses Linux's epoll API to handle multiple simultaneous connections — but it still just sends all of them down to the same backend server. I've tested it using the Apache `ab` server benchmarking tool, and over a million requests, 100 running concurrently, it adds about 0.1ms to the average request time as compared to a direct connection to the web server, which is pretty good going at this early stage. It also doesn't appear to leak memory, which is doubly good going for someone who's not coded in C since the late 90s. I'm pretty sure it's not totally stupid code, though obviously comments and corrections would be much appreciated!

[UPDATE: there's definitely one bug in this version — it doesn't gracefully handle cases when the we can't send data to the client as fast as we're receiving it from the backend. More info here.]

Just like before, the code that I'll be describing is hosted on GitHub as a project called rsp, for "Really Simple Proxy". It's MIT licensed, and the version of it I'll be walking through in this blog post is as of commit f51950b213. I'll copy and paste the code that I'm describing into this post anyway, so if you're following along there's no need to do any kind of complicated checkout.

Before we dive into the code, though, it's worth talking about epoll a bit.

You'll remember that the code for the server in my last post went something like this:

```
while True:
    wait for a new incoming connection from a client
    handle the client connection
```

…where the code to handle the client connection was basically:

```
connect to the backend
read a block's worth of stuff from the client
send it to the backend
while there's still stuff to be read from the backend:
    send it to the client
```

Now, all of those steps to read stuff, or to wait for incoming connections, were blocking calls — we made the call, and when there was data for us to process, the call returned. So handling multiple connections would have been impossible, as (say) while we were waiting for data from one backend we would also have to be waiting for new incoming connections, and perhaps reading from other incoming connections or backends. We'd be trying to do several things at once.

That sounds like the kind of problem threads, or even cooperating processes, were made for. That's a valid solution, and was the normal way of doing it for a long time. But that's changed (at least in part). To see why, think about what would happen on a very busy server, handling hundreds or thousands of concurrent connections. You'd have hundreds or thousands of threads or processes — which isn't a huge deal in and of itself, but they'd all be spending most of their time just sitting there using up memory while they were waiting for data to come in. Processes, or even threads, consume a non-trivial amount of machine resources at that kind of scale, and while there's still a place for them, they're an inefficient way to do this kind of work.

A very popular metaphor for network servers like this these days is to use non-blocking IO. It's a bit of a misnomer, but there's logic behind it. The theory is that instead of having your "read from the backend server" or "wait for an incoming connection" call just sit there and not return until something's available, it doesn't block at all — if there's nothing there to return, it just returns a code saying "nothing for you right now".

Now obviously, you can't write your code so that it's constantly running through a list of the things you're waiting for saying "anything there for me yet?" because that would suck up CPU cycles to no real benefit. So what the non-blocking model does in practice is provide you with a way to register a whole bunch of things you're interested in, and then gives you a blocking (told you it was a misnomer) function that basically says "let me know as soon as there's anything interesting happening on *any of these*“. The “things” that you’re waiting for stuff on are file descriptors. So the previous loop could be rewritten using this model to look something like this:

```
    add the "incoming client connection" file descriptor to the list of things
I'm interested in
    while True:
        wait for an event on the list of things I'm interested in
        if it's an incoming client connection:
            get the file descriptor for the client connection, add it to the
list
            connect to the backend
            add the backend's file descriptor to the list
        else if it's data from an client connection
            send it to the associated backend
        else if it's data from a backend
            send it to the associated client connection
```

…with a bit of extra logic to handle closing connections.

> It’s worth noting that this updated version can not only process multiple connections with just a single thread — it can also handle bidirectional communication between the client and the backend. The previous version read once from the client, sent the result of that read down to the backend, and from then on only sent data from the backend to the client. The code above keeps sending stuff in both directions, so if the client sends something after the initial block of data, while the backend is already replying, then it gets sent to the backend. This isn’t super-useful for simple HTTP requests, but for persistent connections (like WebSockets) it’s essential.

There have been many system calls that been the “wait for an event on the list of things I’m interested in” call in Unix/POSIX-like environments over the years — `select` and `poll`, for example — but they had poor performance as the number of file descriptors got large.

The popular solution, in Linux at least, is epoll. It can handle huge numbers of file descriptors with minimal reduction in performance. (The equivalent in FreeBSB and Mac OS X is kqueue, and according to Wikipedia there’s something similar in Windows and Solaris called “I/O Completion Ports“.)

The rest of this post shows the code I wrote to use epoll in a way that (a) makes sense to me and feels like it will keep making sense as I add more stuff to rsp, and (b) works pretty efficiently.

Just as before, I’ll explain it by working through the code. There are a bunch of different files now, but the main one is still `rsp.c`, which now has a main routine that starts like this:

```
int main(int argc, char* argv[])
{
    if (argc != 4) {
        fprintf(stderr,
                "Usage: %s   \n",
                argv[0]);
        exit(1);
    }
    char* server_port_str = argv[1];
    char* backend_addr = argv[2];
    char* backend_port_str = argv[3];
```

So, some pretty normal initialisation stuff to check the command-line parameters and put them into meaningfully-named variables. (Sharp-eyed readers will have noticed that I’ve updated my code formatting — I'm now putting the `*` to represent a pointer next to the type to which it points, which makes more sense to me than splitting the type definition with a space, and I've also discovered that the C99 standard allows you to declare variables anywhere inside a function, which I think makes the code much more readable.)

Now, the first epoll-specific code:

```
    int epoll_fd = epoll_create1(0);
    if (epoll_fd == -1) {
        perror("Couldn't create epoll FD");
        exit(1);
    }
```

epoll not only allows you to wait for stuff to happen on multiple file descriptors at a time — it's also controlled by its own special type of file descriptor. You can have multiple epoll FDs in a program, each of which gives you the ability to wait for changes on a different set of normal FDs. A specific normal FD could be in several different epoll FDs' sets of things to listen to. You can even add one epoll FD to the list of FDs another epoll FD is watching if you're so inclined. But we're not doing anything quite that complicated here.

You create a special epoll FD using either `epoll_create` or `epoll_create1`. `epoll_create` is pretty much deprecated now (see the link for details) so we just use `epoll_create1` in its simplest form, and bomb out if it returns an error value.

So now we have an epoll instance ready to go, and we need to register some file descriptors with it to listen to. The first one we need is the one that will wait for incoming connections from clients. Here's the code that does that in `rsp.c`

```
    struct epoll_event_handler* server_socket_event_handler;
    server_socket_event_handler = create_server_socket_handler(epoll_fd,
                                                               server_port_str,
                                                               backend_addr,

backend_port_str);

    add_epoll_handler(epoll_fd, server_socket_event_handler, EPOLLIN);
```

This is all code that's using an abstraction I've built on top of epoll that makes it easy to provide callback functions that are called when an event happens on a file descriptor, so it's worth explaining that now. Let's switch to the file `epollinterface.c`. This defines a function `add_epoll_handler` that looks like this:

```
void add_epoll_handler(int epoll_fd, struct epoll_event_handler* handler,
uint32_t event_mask)
{
    struct epoll_event event;

    event.data.ptr = handler;
    event.events = event_mask;
    if (epoll_ctl(epoll_fd, EPOLL_CTL_ADD, handler->fd, &event) == -1) {
        perror("Couldn't register server socket with epoll");
        exit(-1);
    }
}
```

The important system call in there is `epoll_ctl`. This is the function that allows you to add, modify and delete file descriptors from the list that a particular epoll file descriptor is watching. You give it the epoll file descriptor, an operation (`EPOLL_CTL_ADD`, `EPOLL_CTL_MOD` or `EPOLL_CTL_DEL`), the normal file descriptor you're interested in events for, and a pointer to a `struct epoll_event`. The event you pass in has two fields: an event mask saying which events on the file descriptor you're interested in, and some data.

The data is interesting. When you do the "block until something interesting has happened on one or more of this epoll FD's file descriptors" call, it returns a list of results. Obviously, you want to be able to work out for each event where it came from so that you can work out what to do with it. Now, this could have been done by simply returning the file descriptor for each. But epoll's designers were a bit cleverer than that.

The thing is, if all epoll gave you was a set of file descriptors that have had something happen to them, then you would need to maintain some kind of control logic saying "this file descriptor needs to be handled by that bit of code over there, and this one by that code", and so on. That could get complicated quickly. You only need to look at the code of some of the epoll examples on the net to see that while it might make sample code easier to understand at a glance, it won't scale. (I should make it clear that this isn't a criticism of the examples, especially the one I linked to, which is excellent — just my opinion that non-trivial non-sample code needs a different pattern.)

So, when epoll tells you that something's happened on one of the file descriptors you're interested in, it gives you an epoll event just like the one you registered the FD with, with the `events` field set to the bitmask of the events you've received (rather than the set of the events you're interested in) and the `data` field set to whatever it was you gave it at registration time.

The type of the `data` field is a union, and it looks like this:

```
typedef union epoll_data {
    void    *ptr;
    int      fd;
    uint32_t u32;
    uint64_t u64;
} epoll_data_t;
```

> Aside for people newish to C — this was something I had to refresh myself on — a C union is a type that allows you to put any value from a set of types into a variable. So in a variable with the type specification above, you can store *either* a pointer to something (`void*`), an integer, *or* one of two different types of specifically-sized integers. When you retrieve the value, you have to use the field name appropriate to the type of thing you put in there — for example, if you were to store a 32-bit integer in the data using the `u32` name and then retrieve it using the `ptr` variable, the result would be undefined. (Well, on a 32-bit machine it would probably be a pointer to whatever memory address was represented by that 32-bit integer, but that's unlikely to be what you wanted.)

In this case, we're using the `data` pointer inside the union, and we're setting it to a pointer to a `struct epoll_event_handler`. This is a structure I've created to provide callback-like functionality from epoll. Let's take a look — it's in `epollinterface.h`:

```
struct epoll_event_handler {
    int fd;
    void (*handle)(struct epoll_event_handler*, uint32_t);
    void* closure;
};
```

So, an `epoll_event_handler` stores:

- The file descriptor it's associated with
- A callback function to handle an epoll event which takes a pointer to a `epoll_event_handler` structure, and a `uint32_t` which will hold the bitmask representing the events that need to be handled
- And a pointer to something called `closure`; basically, a place to store any data the callback function needs to do its job.

Right. Now we have a function called `add_epoll_handler` that knows how to add a file descriptor and an associated structure to an epoll FD's list of things it's interested in so that it's possible to do a callback with data when an event happens on the epoll FD. Let's go back to the code in `rsp.c` that was calling this. Here it is again:

```
    struct epoll_event_handler* server_socket_event_handler;
    server_socket_event_handler = create_server_socket_handler(epoll_fd,
                                                               server_port_str,
                                                               backend_addr,

backend_port_str);

    add_epoll_handler(epoll_fd, server_socket_event_handler, EPOLLIN);
```

This presumably now makes sense — we're creating a special handler to handle events on the server socket (that is, the thing that listens for incoming client connections) and we're then adding that to our epoll FD, with an event mask that says that we're interested in hearing from it when there's something to read on it — that is, a client connection has come in.

Let's put aside how that server socket handler works for a moment, and finish with `rsp.c`. The next lines look like this:

```
    printf("Started.  Listening on port %s.\n", server_port_str);
    do_reactor_loop(epoll_fd);

    return 0;
}
```

Pretty simple. We print out our status, then call this `do_reactor_loop` function, then return. `do_reactor_loop` is obviously the interesting bit; it's another part of the epoll abstraction layer, and it basically does the "while True" loop in the pseudocode above — it waits for incoming events on the epoll FD, and when they arrive it extracts the appropriate handler, and calls its callback with its closure data. Let's take a look, back in `epollinterface.c`:

```
void do_reactor_loop(int epoll_fd)
{
    struct epoll_event current_epoll_event;

    while (1) {
        struct epoll_event_handler* handler;

        epoll_wait(epoll_fd, ¤t_epoll_event, 1, -1);
        handler = (struct epoll_event_handler*) current_epoll_event.data.ptr;
        handler->handle(handler, current_epoll_event.events);
    }

}
```

It's simple enough. We go into a never-ending loop, and each time around we call `epoll_wait`, which, as you'd expect, is the magic function that blocks until events are available on any one of the file descriptors that our epoll FD is interested in. It takes an epoll FD to wait on, a place to store incoming events, a maximum number of events to receive right now, and a timeout. As we're saying "no timeout" with that `-1` as the last parameter, then when it returns, we know we have an event — so we extract its handler, and call it with the appropriate data. And back around the loop again.

> One interesting thing here: as you'd expect from the parameters, `epoll_wait` can actually get multiple events at once; the `1` we're passing in as the penultimate parameter is to say "just give us one", and we're passing in a pointer to a single `struct epoll_event`. If we wanted more than one then we'd pass in an array of `struct epoll_event`s, with a penultimate parameter saying how long it is so that `epoll_wait` knew the maximum number to get in this batch. When you call `epoll_wait` with a smaller "maximum events to get" parameter than the number that are actually pending, it will return the maximum number you asked for, and then the next time you call it will give you the next ones in its queue immediately, so the only reason to get lots of them in one go is efficiency. But I've noticed no performance improvements from getting multiple epoll events in one go, and only accepting one event at a time has one advantage. Imagine that you're processing an event on a backend socket's FD, which tells you that the backend has closed the connection. You then close the associated client socket, and you free up the memory for both the backend and the client socket's handlers and closures. Closing the client socket means that you'll never get any more events on that client socket (it automatically removes if from any epoll FDs' lists that it's on). But what if there was already an event for the client socket in the event array that was returned from your last call to `epoll_wait`, and you'd just not got to it yet? If that happened, then when you did try to process it, you'd try to get its handler and closure data, which had already been freed. This would almost certainly cause the server to crash. Handling this kind of situation would make the code significantly more complicated, so I've dodged it for now, especially given that it doesn't seem to harm the proxy's speed.

So that's our reactor loop (the name "reactor" comes from Twisted and I've doubtless completely misused the word). The code that remains unexplained is in the event-handlers. Let's start off by looking at the one we skipped over earlier — the `server_socket_event_handler` that we created back in `rsp.c` to listen for incoming connections. It's in `server_socket.c`, and the `create_server_socket_handler` function called from `rsp.c` looks like this:

```
struct epoll_event_handler* create_server_socket_handler(int epoll_fd,
                                                         char* server_port_str,
                                                         char* backend_addr,
                                                         char* backend_port_str)
{

    int server_socket_fd;
    server_socket_fd = create_and_bind(server_port_str);
```

So, we create and bind to a server socket, to get a file descriptor for it. You'll remember that terminology from the last post, and in fact the `create_and_bind` function (also defined in `server_socket.c`) is exactly the same code as we had to do the same job in the original single-connection server.

Now, we do our first new thing — we tell the system to make our server socket non-blocking, which is obviously important if we don't want calls to get data from it to block:

```
    make_socket_non_blocking(server_socket_fd);
```

This isn't a system call, unfortunately — it's a utility function, defined in `netutils.c`, and let's jump over and take a look:

```
void make_socket_non_blocking(int socket_fd)
{
    int flags;

    flags = fcntl(socket_fd, F_GETFL, 0);
    if (flags == -1) {
        perror("Couldn't get socket flags");
        exit(1);
    }

    flags |= O_NONBLOCK;
    if (fcntl(socket_fd, F_SETFL, flags) == -1) {
        perror("Couldn't set socket flags");
        exit(-1);
    }
}
```

Each socket has a number of flags associated with it that control various aspects of it. One of these is whether or not it's non-blocking. So this code simply gets the bitmask that represents the current set of flags associated with a socket, ORs in the "non-blocking" bit, and then applies the new bitmask to the socket. Easy enough, and thanks to Banu Systems for a neatly-encapsulated function for that in their excellent epoll example.

Let's get back to the `create_server_socket_handler` function in `server_socket.c`.

```
    listen(server_socket_fd, MAX_LISTEN_BACKLOG);
```

You'll remember this line from the last post, too. One slight difference — in the first example, we had `MAX_LISTEN_BACKLOG` set to 1. Now it's much higher, at 4096. This came out of the Apache Benchmarker tests I was doing with large numbers of simultaneous connections. If you're running a server, and it gets lots of incoming connection and goes significantly past its backlog, then the OS can assume someone's running a SYN flood denial of service attack against you. You'll see stuff like this in `syslog`:

```
Sep  4 23:09:27 localhost kernel: [3036520.232354] TCP: TCP: Possible SYN
flooding on port 8000. Sending cookies.  Check SNMP counters.
```

Thanks to Erik Dubbelboer for an excellent writeup on how this happens and why. A value of 4096 for the maximum backlog seems to be fine in terms of memory usage and allows this proxy to work well enough for the amount of connections I've tested it with so far.

Moving on in the code:

```
    struct server_socket_event_data* closure = malloc(sizeof(struct
server_socket_event_data));
    closure->epoll_fd = epoll_fd;
    closure->backend_addr = backend_addr;
    closure->backend_port_str = backend_port_str;
```

We create a closure of a special structure type that will contain all of the information that our "there's an incoming client connection" callback will need to do its job, and fill it in appropriately.

```
    struct epoll_event_handler* result = malloc(sizeof(struct
epoll_event_handler));
    result->fd = server_socket_fd;
    result->handle = handle_server_socket_event;
    result->closure = closure;

    return result;
}
```

…then we create a `struct epoll_event_handler` with the FD, the handler function, and the closure, and return it.

That's how we create a server socket that can listen for incoming client connections, which when added to the event loop that the code in `epollinterface.c` defined, will call an appropriate function with the appropriate data.

Next, let's look at that callback. It's called `handle_server_socket_event`, and it's also in `server_socket.c`.

```
void handle_server_socket_event(struct epoll_event_handler* self, uint32_t
events)
{
    struct server_socket_event_data* closure = (struct
server_socket_event_data*) self->closure;
```

We need to be able to extract information from the closure we set up originally for this handler, so we start off by casting it to the appropriate type. Next, we need to accept any incoming connections. We loop through all of them, accepting them one at a time; we don't know up front how many there will be to accept so we just do an infinite loop that we can break out of:

```
    int client_socket_fd;
    while (1) {
        client_socket_fd = accept(self->fd, NULL, NULL);
```

There are two conditions under which an accept will fail (under which circumstances the call to `accept` will return `-1`):

```
        if (client_socket_fd == -1) {
```

Firstly if there's nothing left to accept. If that's the case, we break out of our loop:

```
            if ((errno == EAGAIN) || (errno == EWOULDBLOCK)) {
                break;
```

Secondly, if there's some kind of weird internal error. For now, this means that we just exit the program with an appropriate error message.

```
            } else {
                perror("Could not accept");
                exit(1);
            }
        }
```

If we were able to accept an incoming client connection, we need to create a handler to look after it, which we'll have to add to our central epoll handler. This is done by a new function, `handle_client_connection`

```
        handle_client_connection(closure->epoll_fd,
                                 client_socket_fd,
                                 closure->backend_addr,
                                 closure->backend_port_str);
```

Once that's done, we go back around our accept loop:

```
    }
```

And once the loop is done, we return from this function:

```
}
```

So, in summary — when we get a message from the server socket file descriptor saying that there are one or more incoming connections, we call `handle_server_socket_event`, which accepts all of them, calling `handle_client_connection` for each one. We have to make sure that we accept them all, as we won't be told about them again. (This is actually slightly surprising, for reasons I'll go into later.)

All this means that our remaining unexplained code is what happens from `handle_client_connection` onwards. This is also in `server_socket.c`, and is really simple:

```
void handle_client_connection(int epoll_fd,
                              int client_socket_fd,
                              char* backend_host,
                              char* backend_port_str)
{

    struct epoll_event_handler* client_socket_event_handler;
    client_socket_event_handler = create_client_socket_handler(client_socket_fd,
                                                               epoll_fd,
                                                               backend_host,

backend_port_str);
    add_epoll_handler(epoll_fd, client_socket_event_handler, EPOLLIN |
EPOLLRDHUP);

}
```

We just create a new kind of handler, one for handling events on client sockets, and register it with our epoll loop saying that we're interested in events when data comes in, and when the remote end of the connection is closed.

Onward to the client connection handler code, then! `create_client_socket_handler` is defined in `client_socket.c`, and looks like this:

```
struct epoll_event_handler* create_client_socket_handler(int client_socket_fd,
                                                         int epoll_fd,
                                                         char* backend_host,
                                                        char* backend_port_str)
{

    make_socket_non_blocking(client_socket_fd);

    struct client_socket_event_data* closure = malloc(sizeof(struct
client_socket_event_data));

    struct epoll_event_handler* result = malloc(sizeof(struct
epoll_event_handler));
    result->fd = client_socket_fd;
    result->handle = handle_client_socket_event;
    result->closure = closure;

    closure->backend_handler = connect_to_backend(result, epoll_fd,
backend_host, backend_port_str);

    return result;
}
```

This code should be pretty clear by now. We make the client socket non-blocking, create a closure to store data for callbacks relating to it (in this case, the client handler needs to know about the backend so that it can send data to it), then we set up the event handler object, create the backend connection, and return the handler. There are two new functions being used here — `handle_client_socket_event` and `connect_to_backend`, both of which do exactly what they say they do.

Let's consider `connect_to_backend` first. It's also in `client_socket.c`, and I won't copy it all in here, because it's essentially exactly the same code as was used in the last post to connect to a backend. Once it's done all of the messing around to get the `addrinfo`, connect to the backend, and get an FD that refers to that backend connection, it does a few things that should be pretty clear:

```
    struct epoll_event_handler* backend_socket_event_handler;
    backend_socket_event_handler =
create_backend_socket_handler(backend_socket_fd, client_handler);
    add_epoll_handler(epoll_fd, backend_socket_event_handler, EPOLLIN |
EPOLLRDHUP);
```

The same pattern as before — create a handler to look after that FD, passing in information for the closure (in this case, just as the client handler needed to know about the backend, the backend needs to know about this client), then we add the handler to the epoll event loop, once again saying that we're interested in knowing about incoming data and when the remote end closes the connection.

The only remaining client connection code that we've not gone over is `handle_client_socket_event`. Here it is:

```
void handle_client_socket_event(struct epoll_event_handler* self, uint32_t
events)
{
    struct client_socket_event_data* closure = (struct client_socket_event_data*
) self->closure;

    char buffer[BUFFER_SIZE];
    int bytes_read;
```

After our initial setup, we work out what to do based on the event bitmask that we were provided. Firstly, if there's some data coming in, we read it:

```
    if (events & EPOLLIN) {
        bytes_read = read(self->fd, buffer, BUFFER_SIZE);
```

There are two possible error cases here. Firstly, perhaps we were misinformed and there's no data. If that's the case then we do nothing.

```
        if (bytes_read == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
            return;
        }
```

Secondly, perhaps the remote end has closed the connection. We don't always get an official "remote hung up" if this happens, so we explicitly close the connection if that happens, also closing our connection to the backend.

```
        if (bytes_read == 0 || bytes_read == -1) {
            close_backend_socket(closure->backend_handler);
            close_client_socket(self);
            return;
        }
```

Finally, if we did successfully read some data, we send it down to the backend:

```
        write(closure->backend_handler->fd, buffer, bytes_read);
    }
```

Now, the event bitmask can (of course) contain multiple events. So the following code might also be executed for an event that triggered the above code:

```
    if ((events & EPOLLERR) | (events & EPOLLHUP) | (events & EPOLLRDHUP)) {
        close_backend_socket(closure->backend_handler);
        close_client_socket(self);
        return;
    }
```

…or in other words, if there's been some kind of error or the remote end hung up, we unceremoniously close the connection to the backend and the client connection itself.

```
}
```

And that's the sum of our event handling from client connections.

There's one interesting but perhaps non-obvious thing happening in that code. You'll remember that when we were handling the "incoming client connection" event, we had to carefully accept every incoming connection because we weren't going to be informed about it again. In this handler, however, we read a maximum of `BUFFER_SIZE` bytes (currently 4096). What if there were more than 4096 bytes to read?

Explaining this requires a little more background on epoll. Epoll can operate in two different modes — *edge-triggered* and *level-triggered*. Level-triggered is the default, so it's what we're using here. In level-triggered mode, if you receive an epoll notification that there's data waiting for you, and only read some of it, then epoll notes that there's still unhandled data waiting, and schedules another event to be delivered later. By contrast, edge-triggered mode only informs you once about incoming data. If you don't process it all, it won't tell you again.

So because we're using level-triggered epoll, we don't need to make sure we read everything — we know that epoll will tell us later if there's some stuff we didn't read. And doing things this way gives us a nice way to make sure that when we are handling lots of connections, we time-slice between them reasonably well. After all, if every time we got data from a client, we processed it all in the handler, then if a client sent us lots of data in one go, we'd sit there processing it and ignoring other clients in the meantime. Remember, we're not multi-threading.

That's all very well, but if it's the case and we can use it when processing data from a client, why did we have to be careful to accept all incoming client connections? Surely we could only accept the first one, then rely on epoll to tell us again later that there were still more to handle?

To be honest, I don't know. It seems really odd to me. But I tried changing the accept code to only accept one client connection, and it didn't work — we never got informed about the ones we didn't accept. Someone else got the same behaviour and reported it as a bug in the kernel back in 2006. But it's super-unlikely that something like this is a kernel bug, especially after seven years, so it must be something odd about my code, or deliberate defined behaviour that I've just not found the documentation for. Either way, the thread continuing from that bug report has comments from people saying that regardless of whether you're running edge- or level-triggered, if you want to handle lots of connections then accepting them all in one go is a good idea. So I'll stick with that for the time being, and if anyone more knowledgable than me wants to clarify things in the comments then I'd love to hear more!

So, what's left? Well, there's the code to close the client socket handler:

```
void close_client_socket(struct epoll_event_handler* self)
{
    close(self->fd);
    free(self->closure);
    free(self);
}
```

Simple enough — we close the socket, then free the memory associated with the closure and the handler.

There's a little bit of extra complexity here, in how we call this close function from the `handle_client_socket_event` function. It's all to do with memory management, like most nasty things in C programs. But it's worth having a quick look at the backend-handling code first. As you'd expect, it's in `backend_socket.c`, and it probably looks rather familiar. We have a function to create a backend handler:

```
struct epoll_event_handler* create_backend_socket_handler(int backend_socket_fd,
                                                          struct
epoll_event_handler* client_handler)
{
    make_socket_non_blocking(backend_socket_fd);

    struct backend_socket_event_data* closure = malloc(sizeof(struct
backend_socket_event_data));
    closure->client_handler = client_handler;

    struct epoll_event_handler* result = malloc(sizeof(struct
epoll_event_handler));
    result->fd = backend_socket_fd;
    result->handle = handle_backend_socket_event;
    result->closure = closure;

    return result;
}
```

Essentially the same as its client socket equivalent, but it doesn't need to create a client handler for its closure because one was passed in.

There's a function to handle backend events, which also won't have many surprises:

```
void handle_backend_socket_event(struct epoll_event_handler* self, uint32_t
events)
{
    struct backend_socket_event_data* closure = (struct
backend_socket_event_data*) self->closure;

    char buffer[BUFFER_SIZE];
    int bytes_read;

    if (events & EPOLLIN) {
        bytes_read = read(self->fd, buffer, BUFFER_SIZE);
        if (bytes_read == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
            return;
        }

        if (bytes_read == 0 || bytes_read == -1) {
            close_client_socket(closure->client_handler);
            close_backend_socket(self);
            return;
        }

        write(closure->client_handler->fd, buffer, bytes_read);
    }

    if ((events & EPOLLERR) | (events & EPOLLHUP) | (events & EPOLLRDHUP)) {
        close_client_socket(closure->client_handler);
        close_backend_socket(self);
        return;
    }

}
```

And finally, code to close the backend socket:

```
void close_backend_socket(struct epoll_event_handler* self)
{
    close(self->fd);
    free(self->closure);
    free(self);
}
```

There's a lot of duplication there. Normally I'd refactor to make as much common code as possible between client and backend connections. But the next steps into making this a useful proxy are likely to change the structure enough that it's not worth doing that right now, only to undo it a few commits later. So there it remains, for now.

That's all of the code! The only thing remaining to explain is the memory management weirdness I mentioned in the close handling.

Here's the problem: when a connection is closed, we need to free the memory allocated to the `epoll_event_handler` structure and its associated closure. So our `handle_client_socket_event` function, which is the one notified when the remote end is closed, needs to have access to the handler structure in order to close it. If you were wondering why the epoll interface abstraction passes the handler structure into the callback function (instead of just the closure, which would be more traditional for a callback interface like this) then there’s the explanation — so that it can be freed when the connection closes.

But, you might ask, why don’t we just put the memory management for the handler structure in the epoll event loop, `do_reactor_loop`? When an event comes in, we could handle it as normal and then if the event said that the connection had closed, we could free the handler’s memory. Indeed, we could even handle more obscure cases — perhaps the handler could returns a value saying “I’m done, you can free my handler”.

But it doesn’t work, because we’re not only closing the connection for the FD the handler is handling. When a client connection closes, we need to close the backend, and vice versa. Now, when the remote end closes a connection, we get an event from epoll. But if we close it ourselves, then we don’t. For most normal use, that doesn’t matter — after all, we just closed it, so we should know that we’ve done so and tidy up appropriately.

But when in a client connection handler we’re told that the remote end has disconnected, we need to not only free the client connection (and thus free its handler and its closure), we also need to close the backend and free its stuff up. Which means that the client connection needs to have a reference not just to the backend connection’s FD to send events — it also needs to know about the backend connection’s handler and closure structures because it needs to free them up too.

So there’s the explanation. There are other ways we could do this kind of thing — I’ve tried a bunch — but they all require non-trivial amounts of accounting code to keep track of things. As the system itself is pretty simple right now (notwithstanding the length of this blog post) then I think it would be an unnecessary complication. But it is something that will almost certainly require revisiting later.

So, on that note — that’s it! That’s the code for a trivial epoll-based proxy that connects all incoming connections to a backend. It can handle hundreds of simultaneous connections — indeed, with appropriate ulimit changes to increase the maximum number of open file descriptors, it can handle thousands — and it adds very little overhead.

In the next step, I’m going to integrate Lua scripting. This is how the proxy will ultimately handle backend selection (so that client connections can be delegated to appropriate backends based on the hostname they’re for) but initially I just want to get it integrated for something much simpler. Here’s a link to the post.

Thanks again to Banu Systems for their awesome epoll example, which was invaluable.

Category: Linux | Programming
Post navigation
← Writing a reverse proxy/loadbalancer from the ground up in C, part 1: a trivial single-threaded proxy Writing a reverse proxy/loadbalancer from the ground up in C, part 3: Lua-based configuration →

Writing a reverse proxy/loadbalancer from the ground up in C, part 3: Lua-based configuration

11 September 2013

0 Comments

This is the third step along my road to building a simple C-based reverse proxy/loadbalancer so that I can understand how nginx/OpenResty works — more background here. Here's a link to the first part, where I showed the basic networking code required to write a proxy that could handle one incoming connection at a time and connect it with a single backend, and to the second part, where I added the code to handle multiple connections by using epoll.

This post is much shorter than the last one. I wanted to make the minimum changes to introduce some Lua-based scripting — specifically, I wanted to keep the same proxy with the same behaviour, and just move the stuff that was being configured via command-line parameters into a Lua script, so that just the name of that script would be specified on the command line. It was really easy :-) — but obviously I may have got it wrong, so as ever, any comments and corrections would be much appreciated.

Just like before, the code that I'll be describing is hosted on GitHub as a project called rsp, for "Really Simple Proxy". It's MIT licensed, and the version of it I'll be walking through in this blog post is as of commit 615d20d9a0. I'll copy and paste the code that I'm describing into this post anyway, so if you're following along there's no need to do any kind of complicated checkout.

The first thing I should probably explain, though, is why I picked Lua for this. I'm founder of a company called PythonAnywhere, so why not Python? Well, partly it's a kind of cargo-cult thing. nginx (and particularly OpenResty) use Lua for all of their scripting, so I probably should too (especially if that's what I'm trying to emulate.

Another reason is that Lua is really, really easy to integrate into C programs — it was one of the design goals. Python is reasonably easy to embed, but as soon as you want to get objects out, you have to do a lot of memory management and it can get hairy (just scroll down that page a bit to see what I mean). From what I've read, Lua makes this kind of thing easier. I may learn better later.

Finally, there's the fact that Lua is just very very fast. As a language, I think it's not as nice as Python. But perhaps the things that look like language flaws to me were important tradeoffs in making it so fast. LuaJIT, in particular, is apparently blindingly fast. I've seen the words "alien technology" floating around to refer to it…

So there we go. Let's look at how it can be integrated into the proxy from the last post. Remember, all we're doing at this stage is using it as a glorified config file parser; more interesting stuff will come later.

The first step is to get hold of a Lua library to use. Inspired by the whole alien technology thing, I went for LuaJIT, and installed it thusly:

```
git clone http://luajit.org/git/luajit-2.0.git
cd luajit-2.0
make && sudo make install
```

Now, we need to build it into the program. A few changes to the Makefile; LuaJIT installs itself to `/usr/local/`, so:

```
INCLUDE_DIRS := -I/usr/local/include/luajit-2.0/
LDFLAGS := -Wl,-rpath,/usr/local/lib
LIBS := -lluajit-5.1

...

%.o: %.c $(HEADER_FILES)
       $(CC) -c -Wall -std=gnu99 $(INCLUDE_DIRS) $<

...

rsp: $(OBJ_FILES)
       $(CC) -o $@ $(OBJ_FILES) $(LDFLAGS) $(LIBS)
```

Note the use of `-Wl,-rpath,/usr/local/lib` in the `LDFLAGS` instead of the more traditional `-L/usr/local/lib`. This bakes the location of the library into the rsp executable, so that it knows to look in `/usr/local/lib` at runtime rather than relying on us always setting `LD_LIBRARY_PATH` to tell it where to find the `libluajit-5.1.so` file.

Now, some code to use it. For now, it's all in `rsp.c`. At the top, we need to include the headers:

```
#include <luajit.h>
#include <lauxlib.h>
```

And now we can use it. Jumping down to `main`, there's this line:

```
lua_State *L = lua_open();
```

That's all you need to do to create a new Lua interpreter. The capital `L` seems to be the tradition amongst Lua programmers when embedding the interpreter, so we'll stick with that for now. Next, we want to load our config file (with an appropriate error if it doesn't exist or if it's not valid Lua, both of which conditions are reported to us with appropriate error messages by the Lua interpreter):

```
    if (luaL_dofile(L, argv[1]) != 0) {
        fprintf(stderr, "Error parsing config file: %s\n", lua_tostring(L, -1));
        exit(1);
    }
```

That call to `lua_tostring` with its `-1` parameter is worth a bit more discussion. All data that's passed from Lua to C goes across a stack, which is maintained by the interpreter. `lua_tostring(L, -1)` means "get the top thing from the interpreter's C communication stack, assume it's a string, and put it in a `char*` for me". The function (actually, macro) `luaL_dofile`, if it fails, returns a non-zero code and pushes an error message on to the stack -- so we can use that to extract the error message.

So, once that code's been run, we have a Lua interpreter in which our config file has been run. Now we need to extract the configuration values we want from it. The code that does this in `main` uses a simple utility function, `get_config_opt`:

```
    char* server_port_str = get_config_opt(L, "listenPort");
    char* backend_addr = get_config_opt(L, "backendAddress");
    char* backend_port_str = get_config_opt(L, "backendPort");
```

...and `get_config_opt` looks like this:

```
char* get_config_opt(lua_State* L, char* name) {
    lua_getglobal(L, name);
    if (!lua_isstring(L, -1)) {
        fprintf(stderr, "%s must be a string", name);
        exit(1);
    }
    return (char*) lua_tostring(L, -1);
}
```

Again, that stack: `lua_getglobal` gets the value of the given global variable and puts it on the stack. In Lua, if a global is not defined, it has the value `nil`, so this won't break here; instead, we next ask the interpreter if the thing on the top of the stack is a string using `lua_isstring` -- this covers the `nil` case, and also any other cases where something weird has been put in the variable. Once we've determined that the thing on the top of the stack is a string, we extract it using the `lua_tostring` function we used before.

So, what's the next step? There is no next step! That was all that we needed to do to use Lua to configure the proxy.

Now it's time to do some serious stuff -- parsing the HTTP headers so that we can delegate incoming client connections to backends based on their host header (and perhaps other things). Stay tuned!

[UPDATE: actually, the next post is going to be about fixing a bug in the previous version; more here]

Category: Linux | Programming
Post navigation
← Writing a reverse proxy/loadbalancer from the ground up in C, part 2: handling multiple connections with epoll Writing a reverse proxy/loadbalancer from the ground up in C, pause to regroup: non-blocking output →

Writing a reverse proxy/loadbalancer from the ground up in C, pause to regroup: non-blocking output

28 September 2013
0 Comments

Before moving on to the next step in my from-scratch reverse proxy, I thought it would be nice to install it on the machine where this blog runs, and proxy all access to the blog through it. It would be useful dogfooding and might show any non-obvious errors in the code. And it did.

I found that while short pages were served up perfectly well, longer pages were corrupted and interrupted halfway through. Using curl gave various weird errors, eg. `curl: (56) Problem (3) in the Chunked-Encoded data`, which is a general error saying that it’s receiving chunked data and the chunking is invalid.

Doubly strangely, these problems didn't happen when I ran the proxy on the machine where I'm developing it and got it to proxy the blog; only when I ran it on the same machine as the blog. They're different versions of Ubuntu, the blog server being slightly older, but not drastically so — and none of the stuff I'm using is that new, so it seemed unlikely to be a bug in the blog server's OS. And anyway, select isn't broken.

After a ton of debugging with `printf`s here there and everywhere, I tracked it down. You'll remember that our code to transfer data from the backend to the client looks like this:

```
void handle_backend_socket_event(struct epoll_event_handler* self, uint32_t
events)
{
    struct backend_socket_event_data* closure = (struct
backend_socket_event_data*) self->closure;

    char buffer[BUFFER_SIZE];
    int bytes_read;

    if (events & EPOLLIN) {
        bytes_read = read(self->fd, buffer, BUFFER_SIZE);
        if (bytes_read == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
            return;
        }

        if (bytes_read == 0 || bytes_read == -1) {
            close_client_socket(closure->client_handler);
            close_backend_socket(self);
            return;
        }

        write(closure->client_handler->fd, buffer, bytes_read);
    }

    if ((events & EPOLLERR) | (events & EPOLLHUP) | (events & EPOLLRDHUP)) {
        close_client_socket(closure->client_handler);
        close_backend_socket(self);
        return;
    }

}
```

If you look closely, there's a system call there where I'm not checking the return value — always risky. It's this:

```
        write(closure->client_handler->fd, buffer, bytes_read);
```

The `write` function returns the number of bytes it managed to write, or an error code. The debugging code revealed that sometimes it was returning -1, and `errno` was set to EAGAIN, meaning that the operation would have blocked on a non-blocking socket.

This makes a lot of sense. Sending stuff out over the network is a fairly complex process. There are kernel buffers of stuff to send, and as we're using TCP, which is connection-based, I imagine there's a possibility that the client being slow or transmission of data over the Internet might be causing things to back up. Possibly sometimes it was returning a non-error code, too, but was still not able to write all of the bytes I asked it to write, so stuff was getting skipped.

So that means that even for this simple example of an epoll-based proxy to work properly, we need to do some kind of buffering in the server to handle cases where we're getting stuff from the backend faster than we can send it to the client. And possibly vice versa. It's possible to get epoll events on an FD when it's ready to accept output, so that's probably the way to go — but it will need a bit of restructuring. So the next step will be to implement that, rather than the multiple-backend handling stuff I was planning.

This is excellent. Now I know a little more about why writing something like nginx is hard, and have a vague idea of why I sometimes see stuff in its logs along the lines of `an upstream response is buffered to a temporary file`. Which is entirely why I started writing this stuff in the first place :-)

Here's a run-through of the code I had to write to fix the bug.

Category: Linux | Programming

Post navigation
← Writing a reverse proxy/loadbalancer from the ground up in C, part 3: Lua-based configuration
Writing a reverse proxy/loadbalancer from the ground up in C, pause to regroup: fixed it! →

Writing a reverse proxy/loadbalancer from the ground up in C, pause to regroup: fixed it!

30 September 2013
2 Comments

It took a bit of work, but the bug is fixed: rsp now handles correctly the case when it can't write as much as it wants to the client side. I *think* this is enough for it to properly work as a front-end for this website, so it's installed and running here. If you're reading this (and I've not had to switch it off in the meantime) then the pages you're reading were served over rsp. Which is very pleasing :-)

The code needs a bit of refactoring before I can present it, and the same bug still exists on the communicating-to-backends side (which is one of the reasons it needs refactoring — this is something I should have been able to fix in one place only) so I'll do that over the coming days, and then do another post.

Category: Linux | Programming

Post navigation
← Writing a reverse proxy/loadbalancer from the ground up in C, pause to regroup: non-blocking output