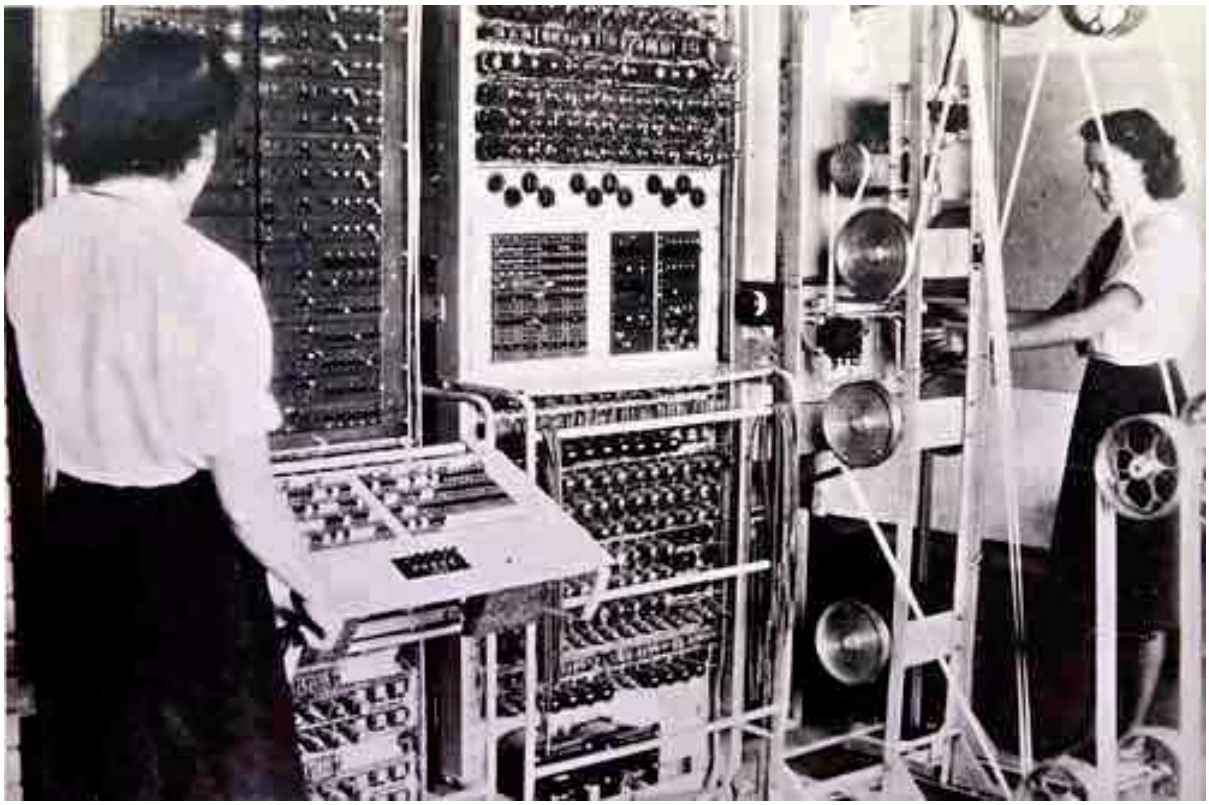

A study of code abstraction

Patrick Lambert

[http://dendory.net]

October 15, 2014

Abstract

Modern developers are shielded from the inner workings of computers and networks thanks to several layers of code abstraction. We'll dig into those layers from a single line of Perl code, down to the bytes that get produced at the bottom of the API stack.

1 Introduction

In the 80s and 90s, anyone who hoped to write a functional script of any kind would have to delve deep into the inner workings of the machine they were working on. For network code, it would be even more challenging, since there were many types of networks, and many systems that spoke different languages, had different file structures, and so on. The standards and APIs were just beginning to be written, and more often than not that meant your code, and in turn you as a developer, had to understand exactly what went on deep down in the system.

Now, things are obviously very different. As APIs matured, it made no sense for everyone to keep reinventing the wheel. As such, code abstraction became the norm. Whether you write in PHP, Perl, Python or Visual C#, you're typically dealing with functions that come from libraries or modules, which in turn talk to other functions, and so on until you end up with an unknown number of abstraction layers between what you write and what actually happens. This makes things easier by removing complexities, but it also removes us from understanding what really happens when we write a line of code, and creates more dependencies on other snippets of code which in turn may contain bugs.

In this document, I will take a single line of Perl code, and follow it down through the modules, all the way to the actual bytes going out on the network. I picked Perl because it's a language I know, because it's available for free on any system, and because it's fairly easy to dig into its various modules.

1.1 Audience

This document is intended for anyone interested in coding and in the inner workings of computer systems. It doesn't assume any familiarity with Perl or a specific language, although having some type of scripting or coding experience would be useful, along with some experience with web development. While you may not understand each snippet of code, the main purpose is to follow the flow all the way down to the lowest level and realize the amount of work that goes on from a single line of code.

Having Perl installed and following along isn't necessary, but it could provide further benefits to try and replicate each layer of abstraction, seeing how easy or hard it is to accomplish the same task with less and less dependencies.

2 Layers upon layers

The function I selected for this experiment is part of the *XML::Feed*[1] module and accomplishes much through a single line of code:

```
my $feed = XML::Feed->parse(URI->new("http://www.reddit.com/.rss"));
```

What this does is simply go out to the web and fetch an XML file, in this case an RSS stream from Reddit, and then returns it as a variable for you to parse. After importing the module and parsing the line, you can then access the information, in this case the latest news entries available on the site. Here is a more complete snippet of code you can try out for yourself to see the whole flow in action:

```
use XML::Feed;
use HTML::FormatText::WithLinks;
my $feed = XML::Feed->parse(URI->new("http://www.reddit.com/.rss"));
foreach my $i ($feed->entries)
{
  print "Title: " . $i->title . "\n";
  print "Time: " . $i->issued . "\n";
  print "Link: " . $i->link . "\n";
  $parsed = HTML::FormatText::WithLinks->new(before_link=>'', after_link=>'',
  footnote=>'');
  print $parsed->parse($i->content->body) . "\n\n";
}
```

It's not necessary to understand all of that, but this code basically loops around each entry gathered from that web site, and then displays the title, time, link and description of each news entry. It also uses the *HTML::FormatText::WithLinks* module to parse the description from HTML into plain text. For this experiment however, we will solely concern ourselves with the line showed above.

2.1 First layer: Parsing the XML

Digging into the Perl API is fairly easy. If you do an online search for *XML::Feed* you will soon find the page on the CPAN site with the documentation for that particular module. There, you can click the Source link which will show you the source code for that module. In our case we're interested in the *parse()* function.

The *parse()* function has 46 lines of code, so already after just the first layer, you can see how much is happening in order to accomplish

this one task. Here is the full source code:

```
sub parse {
    my $class = shift;
    my($stream, $specified_format) = @_;
    return $class->error("Stream parameter is required") unless $stream;
    my $feed = bless {}, $class;
    my $xml = '';
    if (UNIVERSAL::isa($stream, 'URI')) {
        my $ua = LWP::UserAgent->new;
        $ua->agent(__PACKAGE__ . "/$VERSION");
        $ua->env_proxy; # force allowing of proxies
        my $res = URI::Fetch->fetch($stream, UserAgent => $ua)
            or return $class->error(URI::Fetch->errstr);
        return $class->error("This feed has been permanently removed")
            if $res->status == URI::Fetch::URI_GONE();
        $xml = $res->content;
    } elsif (ref($stream) eq 'SCALAR') {
        $xml = $$stream;
    } elsif (ref($stream)) {
        while (read($stream, my($chunk), 8192)) {
            $xml .= $chunk;
        }
    } else {
        open my $fh, $stream
            or return $class->error("Can't open $stream: $!");
        while (read $fh, my($chunk), 8192) {
            $xml .= $chunk;
        }
        close $fh;
    }
    return $class->error("Can't get feed XML content from $stream")
        unless $xml;
    my $format;
    if ($specified_format) {
        $format = $specified_format;
    } else {
        $format = $feed->identify_format(\$xml)
         or return $class->error($feed->errstr);
    }

    my $format_class = join '::', __PACKAGE__, "Format", $format;
    eval "use $format_class";
    return $class->error("Unsupported format $format: $@") if $@;
    bless $feed, $format_class;
    $feed->init_string(\$xml) or return $class->error($feed->errstr);
    $feed;
}
```

A lot of that is to handle errors and possible edge cases, and then identify what kind of XML data it is. The actual parsing of the XML is done in other modules, namely *XML::Feed::Format::RSS* for RSS feeds, but we're not going to concern ourselves with that part. In reality, we're only interested in a small fraction of the function. What *parse()* actually does is use the *LWP::UserAgent*[2] module in order to make the connection on line 8, since it provides functions to handle proxy servers, create HTTP headers and more, then it uses *URI::Fetch*[3] on line 11 which is another module that provides a convenient way of reading web pages, including support for various features of the HTTP protocol like compression, caching, error codes and more. Then, it uses the built-in Perl *read()* function in order to read the incoming data line by line. Here are the relevant lines:

```
1  ...
2          my $ua = LWP::UserAgent->new;
3  ...
4          my $res = URI::Fetch->fetch($stream, UserAgent => $ua)
5  ...
6          open my $fh, $stream
7  ...
8          while (read $fh, my($chunk), 8192) {
9              $xml .= $chunk;
10         }
11 ...
```

Through these two additional modules, along with the native *read()* function, the first abstraction layer can accomplish much through what is still a fairly small amount of code. Now it's time to go down to the second layer.

2.2 Second layer: Setting up the connection

2.2.1 Setting a user agent

Before being able to read the file, a connection to the web server has to be made. If you recall from the last section, this is done by using the *new* function from the *LWP::UserAgent* function. While this module is fully able to actually go out and make the connection over the network, in this case the author of the previous layer selected to use *new()* and then do some more processing first. Let's look at the code:

```
1  sub new
2  {
3      # Check for common user mistake
```

```
    Carp::croak("Options to LWP::UserAgent should be key/value pairs, not hash
     reference")
        if ref($_[1]) eq 'HASH';

    my($class, %cnf) = @_;

    my $agent = delete $cnf{agent};
    my $from = delete $cnf{from};
    my $def_headers = delete $cnf{default_headers};
    my $timeout = delete $cnf{timeout};
    $timeout = 3*60 unless defined $timeout;
    my $local_address = delete $cnf{local_address};
    my $ssl_opts = delete $cnf{ssl_opts} || {};
    unless (exists $ssl_opts->{verify_hostname}) {
    # The processing of HTTPS_CA_* below is for compatibility with Crypt::SSLeay
    if (exists $ENV{PERL_LWP_SSL_VERIFY_HOSTNAME}) {
        $ssl_opts->{verify_hostname} = $ENV{PERL_LWP_SSL_VERIFY_HOSTNAME};
    }
    elsif ($ENV{HTTPS_CA_FILE} || $ENV{HTTPS_CA_DIR}) {
        # Crypt-SSLeay compatibility (verify peer certificate; but not the hostname)
        $ssl_opts->{verify_hostname} = 0;
        $ssl_opts->{SSL_verify_mode} = 1;
    }
    else {
        $ssl_opts->{verify_hostname} = 1;
    }
    }
    unless (exists $ssl_opts->{SSL_ca_file}) {
    if (my $ca_file = $ENV{PERL_LWP_SSL_CA_FILE} || $ENV{HTTPS_CA_FILE}) {
        $ssl_opts->{SSL_ca_file} = $ca_file;
    }
    }
    unless (exists $ssl_opts->{SSL_ca_path}) {
    if (my $ca_path = $ENV{PERL_LWP_SSL_CA_PATH} || $ENV{HTTPS_CA_DIR}) {
        $ssl_opts->{SSL_ca_path} = $ca_path;
    }
    }
    my $use_eval = delete $cnf{use_eval};
    $use_eval = 1 unless defined $use_eval;
    my $parse_head = delete $cnf{parse_head};
    $parse_head = 1 unless defined $parse_head;
    my $show_progress = delete $cnf{show_progress};
    my $max_size = delete $cnf{max_size};
    my $max_redirect = delete $cnf{max_redirect};
    $max_redirect = 7 unless defined $max_redirect;
    my $env_proxy = exists $cnf{env_proxy} ? delete $cnf{env_proxy} :
     $ENV{PERL_LWP_ENV_PROXY};

    my $cookie_jar = delete $cnf{cookie_jar};
```

```
    my $conn_cache = delete $cnf{conn_cache};
    my $keep_alive = delete $cnf{keep_alive};

    Carp::croak("Can't mix conn_cache and keep_alive")
      if $conn_cache && $keep_alive;

    my $protocols_allowed = delete $cnf{protocols_allowed};
    my $protocols_forbidden = delete $cnf{protocols_forbidden};

    my $requests_redirectable = delete $cnf{requests_redirectable};
    $requests_redirectable = ['GET', 'HEAD']
      unless defined $requests_redirectable;

    # Actually ""s are just as good as 0's, but for concision we'll just say:
    Carp::croak("protocols_allowed has to be an arrayref or 0, not
     \"$protocols_allowed\"!")
      if $protocols_allowed and ref($protocols_allowed) ne 'ARRAY';
    Carp::croak("protocols_forbidden has to be an arrayref or 0, not
     \"$protocols_forbidden\"!")
      if $protocols_forbidden and ref($protocols_forbidden) ne 'ARRAY';
    Carp::croak("requests_redirectable has to be an arrayref or 0, not
     \"$requests_redirectable\"!")
      if $requests_redirectable and ref($requests_redirectable) ne 'ARRAY';


    if (%cnf && $^W) {
    Carp::carp("Unrecognized LWP::UserAgent options: @{[sort keys %cnf]}");
    }

    my $self = bless {
             def_headers => $def_headers,
             timeout    => $timeout,
             local_address => $local_address,
             ssl_opts   => $ssl_opts,
             use_eval   => $use_eval,
                    show_progress=> $show_progress,
             max_size   => $max_size,
             max_redirect => $max_redirect,
                    proxy      => {},
             no_proxy   => [],
                    protocols_allowed  => $protocols_allowed,
                    protocols_forbidden => $protocols_forbidden,
                    requests_redirectable => $requests_redirectable,
            }, $class;

    $self->agent(defined($agent) ? $agent : $class->_agent)
    if defined($agent) || !$def_headers || !$def_headers->header("User-Agent");
    $self->from($from) if $from;
    $self->cookie_jar($cookie_jar) if $cookie_jar;
```

```
102     $self->parse_head($parse_head);
103     $self->env_proxy if $env_proxy;
104
105     $self->protocols_allowed( $protocols_allowed ) if $protocols_allowed;
106     $self->protocols_forbidden($protocols_forbidden) if $protocols_forbidden;
107
108     if ($keep_alive) {
109     $conn_cache ||= { total_capacity => $keep_alive };
110     }
111     $self->conn_cache($conn_cache) if $conn_cache;
112
113     return $self;
114 }
```

Okay, so the amount of code is starting to be staggering. Fortunately, again only a small portion is of interest for our purposes. Basically, the goal of this function is to create a new instance of the *LWP::UserAgent* class and configure various parameters to do with the upcoming connection, such as the user agent to pass to the server, how long the connection should stay active before timing out, which HTTP headers should be sent out, how to handle encryption in the case of SSL web sites, how to store any cookies that the site decides to send, and so on.

As you can imagine, if every developer had to worry about all of these things every time they wanted to fetch a file from a web site, it would be quite inconvenient. In our particular case, no parameter is passed out to the *new()* function so we're basically accepting all the defaults, then *parse()* above is making two additional changes, namely setting a custom user agent, and copying the system-wide proxy settings to the function:

```
1         $ua->agent(__PACKAGE__ . "/$VERSION");
2         $ua->env_proxy; # force allowing of proxies
```

Take note of what the user agent is being set to. When we get down through the layers and look at the code that actually goes out on the network, we'll get to see it in action.

2.2.2 Opening a stream

So far we've gotten a *LWP::UserAgent* object, and now we need to open up a stream. If you recall from section 2.1, the user agent variable is *$ua*, which is then passed to the *fetch()* function from the *URI::Fetch* module. Let's look at this latest one. Be ready to scroll:

```
1 sub fetch {
```

```
    my $class = shift;
    my($uri, %param) = @_;

    # get user parameters
    my $cache      = delete $param{Cache};
    my $ua         = delete $param{UserAgent};
    my $p_etag     = delete $param{ETag};
    my $p_lastmod  = delete $param{LastModified};
    my $content_hook = delete $param{ContentAlterHook};
    my $p_no_net   = delete $param{NoNetwork};
    my $p_cache_grep = delete $param{CacheEntryGrep};
    my $freeze     = delete $param{Freeze};
    my $thaw       = delete $param{Thaw};
    my $force      = delete $param{ForceResponse};
    croak("Unknown parameters: " . join(", ", keys %param))
        if %param;

    my $ref;
    if ($cache) {
        unless ($freeze && $thaw) {
            require Storable;
            $thaw = \&Storable::thaw;
            $freeze = \&Storable::freeze;
        }
        if (my $blob = $cache->get($uri)) {
            $ref = $thaw->($blob);
        }
    }

    # NoNetwork support (see pod docs below for logic clarification)
    if ($p_no_net) {
        croak("Invalid NoNetworkValue (negative)") if $p_no_net < 0;
        if ($ref && ($p_no_net == 1 || $ref->{CacheTime} > time() - $p_no_net)) {
            my $fetch = URI::Fetch::Response->new;
            $fetch->status(URI_OK);
            $fetch->content($ref->{Content});
            $fetch->etag($ref->{ETag});
            $fetch->last_modified($ref->{LastModified});
            $fetch->content_type($ref->{ContentType});
            return $fetch;
        }
        return undef if $p_no_net == 1;
    }

    $ua ||= do {
        my $ua = LWP::UserAgent->new;
        $ua->agent(join '/', $class, $class->VERSION);
        $ua->env_proxy;
        $ua;
```

```
    };

    my $req = HTTP::Request->new(GET => $uri);
    if ($HAS_ZLIB) {
        $req->header('Accept-Encoding', 'gzip');
    }
    if (my $etag = ($p_etag || $ref->{ETag})) {
        $req->header('If-None-Match', $etag);
    }
    if (my $ts = ($p_lastmod || $ref->{LastModified})) {
        $req->if_modified_since($ts);
    }

    my $res = $ua->request($req);
    my $fetch = URI::Fetch::Response->new;
    $fetch->uri($uri);
    $fetch->http_status($res->code);
    $fetch->http_response($res);
    $fetch->content_type($res->header('Content-Type'));
    if ($res->previous && $res->previous->code ==
     HTTP::Status::RC_MOVED_PERMANENTLY()) {
        $fetch->status(URI_MOVED_PERMANENTLY);
        $fetch->uri($res->previous->header('Location'));
    } elsif ($res->code == HTTP::Status::RC_GONE()) {
        $fetch->status(URI_GONE);
        $fetch->uri(undef);
        return $fetch;
    } elsif ($res->code == HTTP::Status::RC_NOT_MODIFIED()) {
        $fetch->status(URI_NOT_MODIFIED);
        $fetch->content($ref->{Content});
        $fetch->etag($ref->{ETag});
        $fetch->last_modified($ref->{LastModified});
        $fetch->content_type($ref->{ContentType});
        return $fetch;
    } elsif (!$res->is_success) {
        return $force ? $fetch : $class->error($res->message);

    } else {
        $fetch->status(URI_OK);
    }
    $fetch->last_modified($res->last_modified);
    $fetch->etag($res->header('ETag'));
    my $content = $res->content;
    if ($res->content_encoding && $res->content_encoding eq 'gzip') {
        $content = Compress::Zlib::memGunzip($content);
    }

    # let caller-defined transform hook modify the result that'll be
    # cached. perhaps the caller only wants the <head> section of
```

```
100     # HTML, or wants to change the content to a parsed datastructure
101     # already serialized with Storable.
102     if ($content_hook) {
103         croak("ContentAlterHook is not a subref") unless ref $content_hook eq
104          "CODE";
105         $content_hook->(\$content);
106     }
107
108     $fetch->content($content);
109
110     # cache by default, if there's a cache. but let callers cancel
111     # the cache action by defining a cache grep hook
112     if ($cache &&
113         ($p_cache_grep ? $p_cache_grep->($fetch) : 1)) {
114
115         $cache->set($uri, $freeze->({
116             ETag         => $fetch->etag,
117             LastModified => $fetch->last_modified,
118             Content      => $fetch->content,
119             CacheTime    => time(),
120             ContentType => $fetch->content_type,
121         }));
122     }
123     $fetch;
124 }
```

Let's break down what the function does. First, it accepts a number of parameters which get set after line 5. As we've seen in 2.1, only two get passed on in our case: the stream variable, which will be used to read the information from the network, and the user agent class. Then, this function has a number of conditional statements to deal with all of these potential parameters. In our case, most of them are ignored since we aren't dealing with cache, serialization, content handling, and so on. Instead, we go right into the interesting part at line 53 which deals with opening the network connection, and then after line 64, dealing with the response from the server. Here is the relevant code:

```
1     my $req = HTTP::Request->new(GET => $uri);
2 ...
3     my $res = $ua->request($req);
4     my $fetch = URI::Fetch::Response->new;
5     $fetch->uri($uri);
```

As you can see, once again this function doesn't actually have any network code. It creates a new object from the *HTTP::Request*[4] module, then uses the *request()* function from *LWP::UserAgent* to make the request on the opened connection, which will be our third abstraction layer. Then, we see another new module being used, *URI::Fetch::Response*[5]

in order to parse the various response codes. A web server can return a number of codes along with the normal HTTP headers, such as whether the connection was successful, if the file was moved, if the requested item doesn't exist, and so on. This is what we see happening in the rest of the function.

2.3 Third layer: Making the request

Let's recap. So far, we've been through two different abstraction layers. After using the *XML::Feed* module to parse the XML file from Reddit, we go down into *LWP::UserAgent* and *URI::Fetch* in order to make a new user agent object, and pass it to *fetch()* in order to get data from a site. Now, we go down one more layer into *HTTP::Request->new()* to prepare the request and *LWP::UserAgent->request()* to send it out.

2.3.1 Preparing the request

The first thing that the previous layer does is calling the *new()* function from *HTTP::Request*. Let's look at the code:

```
1 sub new
2 {
3     my($class, $method, $uri, $header, $content) = @_;
4     my $self = $class->SUPER::new($header, $content);
5     $self->method($method);
6     $self->uri($uri);
7     $self;
8 }
```

While this is a tiny function, the amount of work it does is deceiving. On line 4, it actually calls the *new()* function of its base module, *HTTP::Message*[6]. We'll skip that one because all it does is set the default headers for the upcoming connection. The next line sets the method for the connection, which is a parameter passed by the previous layer. If you remember, that method was *GET*. Any HTTP connection must have a valid method. *GET* is usually used to fetch information, while *POST* is used to send form data, such as logging into a web site. Finally, the *uri()* function simply parses the URL passed to make sure it's valid, returning various error messages in case it isn't.

2.3.2 Sending the request

After making a new object of the *HTTP::Request* type, the previous abstraction layer called the *request()* function from the *LWP::UserAgent* module. Let's see its source code:

```
sub request
{
    my($self, $request, $arg, $size, $previous) = @_;

    my $response = $self->simple_request($request, $arg, $size);
    $response->previous($previous) if $previous;

    if ($response->redirects >= $self->{max_redirect}) {
        $response->header("Client-Warning" =>
                       "Redirect loop detected (max_redirect =
                        $self->{max_redirect})");
        return $response;
    }

    if (my $req = $self->run_handlers("response_redirect", $response)) {
        return $self->request($req, $arg, $size, $response);
    }

    my $code = $response->code;

    if ($code == &HTTP::Status::RC_MOVED_PERMANENTLY or
    $code == &HTTP::Status::RC_FOUND or
    $code == &HTTP::Status::RC_SEE_OTHER or
    $code == &HTTP::Status::RC_TEMPORARY_REDIRECT)
    {
    my $referral = $request->clone;

    # These headers should never be forwarded
    $referral->remove_header('Host', 'Cookie');

    if ($referral->header('Referer') &&
        $request->uri->scheme eq 'https' &&
        $referral->uri->scheme eq 'http')
    {
        # RFC 2616, section 15.1.3.
        # https -> http redirect, suppressing Referer
        $referral->remove_header('Referer');
    }

    if ($code == &HTTP::Status::RC_SEE_OTHER ||
        $code == &HTTP::Status::RC_FOUND)
        {
        my $method = uc($referral->method);
```

```
        unless ($method eq "GET" || $method eq "HEAD") {
        $referral->method("GET");
        $referral->content("");
        $referral->remove_content_headers;
        }
    }

    # And then we update the URL based on the Location:-header.
    my $referral_uri = $response->header('Location');
    {
        # Some servers erroneously return a relative URL for redirects,
        # so make it absolute if it not already is.
        local $URI::ABS_ALLOW_RELATIVE_SCHEME = 1;
        my $base = $response->base;
        $referral_uri = "" unless defined $referral_uri;
        $referral_uri = $HTTP::URI_CLASS->new($referral_uri, $base)
                  ->abs($base);
    }
    $referral->uri($referral_uri);

    return $response unless $self->redirect_ok($referral, $response);
    return $self->request($referral, $arg, $size, $response);

    }
    elsif ($code == &HTTP::Status::RC_UNAUTHORIZED ||
         $code == &HTTP::Status::RC_PROXY_AUTHENTICATION_REQUIRED
        )
    {
    my $proxy = ($code == &HTTP::Status::RC_PROXY_AUTHENTICATION_REQUIRED);
    my $ch_header = $proxy || $request->method eq 'CONNECT'
        ?  "Proxy-Authenticate" : "WWW-Authenticate";
    my @challenge = $response->header($ch_header);
    unless (@challenge) {
        $response->header("Client-Warning" =>
                "Missing Authenticate header");
        return $response;
    }

    require HTTP::Headers::Util;
    CHALLENGE: for my $challenge (@challenge) {
        $challenge =~ tr/,/;/; # "," is used to separate auth-params!!
        ($challenge) = HTTP::Headers::Util::split_header_words($challenge);
        my $scheme = shift(@$challenge);
        shift(@$challenge); # no value
        $challenge = { @$challenge }; # make rest into a hash

        unless ($scheme =~ /^([a-z]+(?:-[a-z]+)*)$/) {
        $response->header("Client-Warning" =>
                "Bad authentication scheme '$scheme'");
```

```
93          return $response;
94          }
95          $scheme = $1; # untainted now
96          my $class = "LWP::Authen::\u$scheme";
97          $class =~ s/-/_/g;
98
99          no strict 'refs';
100         unless (%{"$class\::"}) {
101         # try to load it
102         eval "require $class";
103         if ($@) {
104             if ($@ =~ /^Cant locate/) {
105             $response->header("Client-Warning" =>
106                     "Unsupported authentication scheme '$scheme'");
107             }
108             else {
109             $response->header("Client-Warning" => $@);
110             }
111             next CHALLENGE;
112         }
113         }
114         unless ($class->can("authenticate")) {
115         $response->header("Client-Warning" =>
116                 "Unsupported authentication scheme '$scheme'");
117         next CHALLENGE;
118         }
119         return $class->authenticate($self, $proxy, $challenge, $response,
120                     $request, $arg, $size);
121     }
122     return $response;
123     }
124     return $response;
125 }
```

While this is a massive function, it's actually just half of the story. Its main purpose is to parse the headers received by the server and act on them. For example, line 21 checks whether the server said that the requested file was moved, and if so, makes another request to the new address. On line 72 it also handles the case where a proxy server is required, and whether that proxy needs authentication.

But before all of that can happen, we still need to open the actual connection, we need the function that tells the system to open a network socket. This happens on line 5. The *simple_request()* function actually does some more preparation tasks and then calls *send_request()*. That's the second half of the story for this layer of abstraction:

```
1 sub send_request
```

```
{
    my($self, $request, $arg, $size) = @_;
    my($method, $url) = ($request->method, $request->uri);
    my $scheme = $url->scheme;

    local($SIG{__DIE__}); # protect against user defined die handlers

    $self->progress("begin", $request);

    my $response = $self->run_handlers("request_send", $request);

    unless ($response) {
        my $protocol;

        {
            # Honor object-specific restrictions by forcing protocol objects
            #  into class LWP::Protocol::nogo.
            my $x;
            if($x = $self->protocols_allowed) {
                if (grep lc($_) eq $scheme, @$x) {
                }
                else {
                    require LWP::Protocol::nogo;
                    $protocol = LWP::Protocol::nogo->new;
                }
            }
            elsif ($x = $self->protocols_forbidden) {
                if(grep lc($_) eq $scheme, @$x) {
                    require LWP::Protocol::nogo;
                    $protocol = LWP::Protocol::nogo->new;
                }
            }
            # else fall thru and create the protocol object normally
        }

        # Locate protocol to use
        my $proxy = $request->{proxy};
        if ($proxy) {
            $scheme = $proxy->scheme;
        }

        unless ($protocol) {
            $protocol = eval { LWP::Protocol::create($scheme, $self) };
            if ($@) {
                $@ =~ s/ at .* line \d+.*//s; # remove file/line number
                $response = _new_response($request,
                 &HTTP::Status::RC_NOT_IMPLEMENTED, $@);
                if ($scheme eq "https") {
                    $response->message($response->message . "
```

```
                    (LWP::Protocol::https not installed)");
                $response->content_type("text/plain");
                $response->content(<<EOT);
LWP will support https URLs if the LWP::Protocol::https module
is installed.
EOT
            }
        }
    }

    if (!$response && $self->{use_eval}) {
        # we eval, and turn dies into responses below
        eval {
            $response = $protocol->request($request, $proxy, $arg, $size,
             $self->{timeout}) ||
        die "No response returned by $protocol";
        };
        if ($@) {
            if (UNIVERSAL::isa($@, "HTTP::Response")) {
                $response = $@;
                $response->request($request);
            }
            else {
                my $full = $@;
                (my $status = $@) =~ s/\n.*//s;
                $status =~ s/ at .* line \d+.*//s; # remove file/line number
                my $code = ($status =~ s/^(\d\d\d)\s+//) ? $1 :
                 &HTTP::Status::RC_INTERNAL_SERVER_ERROR;
                $response = _new_response($request, $code, $status, $full);
            }
        }
    }
    elsif (!$response) {
        $response = $protocol->request($request, $proxy,
                                       $arg, $size, $self->{timeout});
        # XXX: Should we die unless $response->is_success ???
    }
}

$response->request($request); # record request for reference
$response->header("Client-Date" => HTTP::Date::time2str(time));

$self->run_handlers("response_done", $response);

$self->progress("end", $response);
return $response;
}
```

Again, we see a lot more handling of the request, error handling, and various edge cases. Little is of interest in here, except for line 64. Here we get introduced to another module which will make up the next abstraction layer: *LWP::Protocol*[7]. All of the information we've dealt with so far, including the request from *HTTP::Request*, the user agent, optional arguments, are all passed onto that new module.

2.4 Fourth layer: Defining protocols

If you look at the *new()* function for the *LWP::Protocol* module, you may be left a bit confused:

```
1  sub create
2  {
3      my($scheme, $ua) = @_;
4      my $impclass = LWP::Protocol::implementor($scheme) or
5      Carp::croak("Protocol scheme '$scheme' is not supported");
6
7      # hand-off to scheme specific implementation sub-class
8      my $protocol = $impclass->new($scheme, $ua);
9
10     return $protocol;
11 }
```

This is supposed to be the key module to do all of the network stuff we've been looking for since the start. What this actually does is hand off all of the work to the proper subclass. This module has sub-modules for each type of request, including *LWP::Protocol::http* for HTTP requests, *LWP::Protocol:file* for files, and so on. But if you go on the CPAN site and try to look at those sub-modules, you may find them suspiciously missing. This is because for the first time so far, this layer has been hidden from us. While developers are expected to work with any of the previous modules, now we've finally delved deep enough that for normal use cases, it's been decided that we're now entering a layer deep enough that we shouldn't mess with it. Here be dragons...

Of course, Perl is open source, and the whole point of this experiment is to break away those layers, so we're not going to let that stop us. If you go on the source repository for the module[8], you can find what we need. Here's the code for the massive *request()* function:

```
1  sub request
2  {
3      my($self, $request, $proxy, $arg, $size, $timeout) = @_;
4
5      $size ||= 4096;
```

```
6
7       # check method
8       my $method = $request->method;
9       unless ($method =~ /^[A-Za-z0-9_!\#\$%&\'*+\-.^\`|~]+$/) { # HTTP token
10      return HTTP::Response->new( &HTTP::Status::RC_BAD_REQUEST,
11                  'Library does not allow method ' .
12                  "$method for 'http:' URLs");
13      }
14
15      my $url = $request->uri;
16      my($host, $port, $fullpath);
17
18      # Check if we're proxy'ing
19      if (defined $proxy) {
20      # $proxy is an URL to an HTTP server which will proxy this request
21      $host = $proxy->host;
22      $port = $proxy->port;
23      $fullpath = $method eq "CONNECT" ?
24                      ($url->host . ":" . $url->port) :
25                      $url->as_string;
26      }
27      else {
28      $host = $url->host;
29      $port = $url->port;
30      $fullpath = $url->path_query;
31      $fullpath = "/$fullpath" unless $fullpath =~ m,^/,;
32      }
33
34      # connect to remote site
35      my $socket = $self->_new_socket($host, $port, $timeout);
36
37      my $http_version = "";
38      if (my $proto = $request->protocol) {
39      if ($proto =~ /^(?:HTTP\/)?(1.\d+)$/) {
40          $http_version = $1;
41          $socket->http_version($http_version);
42          $socket->send_te(0) if $http_version eq "1.0";
43      }
44      }
45
46      $self->_check_sock($request, $socket);
47
48      my @h;
49      my $request_headers = $request->headers->clone;
50      $self->_fixup_header($request_headers, $url, $proxy);
51
52      $request_headers->scan(sub {
53                  my($k, $v) = @_;
54                  $k =~ s/^:://;
```

```
                $v =~ s/\n/ /g;
                push(@h, $k, $v);
            });

    my $content_ref = $request->content_ref;
    $content_ref = $$content_ref if ref($$content_ref);
    my $chunked;
    my $has_content;

    if (ref($content_ref) eq 'CODE') {
    my $clen = $request_headers->header('Content-Length');
    $has_content++ if $clen;
    unless (defined $clen) {
        push(@h, "Transfer-Encoding" => "chunked");
        $has_content++;
        $chunked++;
    }
    }
    else {
    # Set (or override) Content-Length header
    my $clen = $request_headers->header('Content-Length');
    if (defined($$content_ref) && length($$content_ref)) {
        $has_content = length($$content_ref);
        if (!defined($clen) || $clen ne $has_content) {
        if (defined $clen) {
            warn "Content-Length header value was wrong, fixed";
            hlist_remove(\@h, 'Content-Length');
        }
        push(@h, 'Content-Length' => $has_content);
        }
    }
    elsif ($clen) {
        warn "Content-Length set when there is no content, fixed";
        hlist_remove(\@h, 'Content-Length');
    }
    }

    my $write_wait = 0;
    $write_wait = 2
    if ($request_headers->header("Expect") || "") =~ /100-continue/;

    my $req_buf = $socket->format_request($method, $fullpath, @h);
    #print "------\n$req_buf\n------\n";

    if (!$has_content || $write_wait || $has_content > 8*1024) {
      WRITE:
        {
            # Since this just writes out the header block it should almost
            # always succeed to send the whole buffer in a single write call.
```

```
            my $n = $socket->syswrite($req_buf, length($req_buf));
            unless (defined $n) {
                redo WRITE if $!{EINTR};
                if ($!{EAGAIN}) {
                    select(undef, undef, undef, 0.1);
                    redo WRITE;
                }
                die "write failed: $!";
            }
            if ($n) {
                substr($req_buf, 0, $n, "");
            }
            else {
                select(undef, undef, undef, 0.5);
            }
            redo WRITE if length $req_buf;
        }
    }

    my($code, $mess, @junk);
    my $drop_connection;

    if ($has_content) {
    my $eof;
    my $wbuf;
    my $woffset = 0;
      INITIAL_READ:
    if ($write_wait) {
        # skip filling $wbuf when waiting for 100-continue
        # because if the response is a redirect or auth required
        # the request will be cloned and there is no way
        # to reset the input stream
        # return here via the label after the 100-continue is read
    }
    elsif (ref($content_ref) eq 'CODE') {
        my $buf = &$content_ref();
        $buf = "" unless defined($buf);
        $buf = sprintf "%x%s%s%s", length($buf), $CRLF, $buf, $CRLF
        if $chunked;
        substr($buf, 0, 0) = $req_buf if $req_buf;
        $wbuf = \$buf;
    }
    else {
        if ($req_buf) {
        my $buf = $req_buf . $$content_ref;
        $wbuf = \$buf;
        }
        else {
        $wbuf = $content_ref;
```

```
153         }
154         $eof = 1;
155     }
156 
157     my $fbits = '';
158     vec($fbits, fileno($socket), 1) = 1;
159 
160       WRITE:
161     while ($write_wait || $woffset < length($$wbuf)) {
162 
163         my $sel_timeout = $timeout;
164         if ($write_wait) {
165         $sel_timeout = $write_wait if $write_wait < $sel_timeout;
166         }
167         my $time_before;
168             $time_before = time if $sel_timeout;
169 
170         my $rbits = $fbits;
171         my $wbits = $write_wait ? undef : $fbits;
172             my $sel_timeout_before = $sel_timeout;
173           SELECT:
174             {
175                 my $nfound = select($rbits, $wbits, undef, $sel_timeout);
176                 if ($nfound < 0) {
177                     if ($!{EINTR} || $!{EAGAIN}) {
178                         if ($time_before) {
179                             $sel_timeout = $sel_timeout_before - (time -
180                              $time_before);
181                             $sel_timeout = 0 if $sel_timeout < 0;
182                         }
183                         redo SELECT;
184                     }
185                     die "select failed: $!";
186                 }
187         }
188 
189         if ($write_wait) {
190         $write_wait -= time - $time_before;
191         $write_wait = 0 if $write_wait < 0;
192         }
193 
194         if (defined($rbits) && $rbits =~ /[^\0]/) {
195         # readable
196         my $buf = $socket->_rbuf;
197         my $n = $socket->sysread($buf, 1024, length($buf));
198                 unless (defined $n) {
199                     die "read failed: $!" unless $!{EINTR} || $!{EAGAIN};
200                     # if we get here the rest of the block will do nothing
201                     # and we will retry the read on the next round
```

```
202                 }
203             elsif ($n == 0) {
204                     # the server closed the connection before we finished
205                     # writing all the request content. No need to write any more.
206                     $drop_connection++;
207                     last WRITE;
208             }
209             $socket->_rbuf($buf);
210             if (!$code && $buf =~ /\015?\012\015?\012/) {
211             # a whole response header is present, so we can read it without blocking
212                 ($code, $mess, @h) = $socket->read_response_headers(laxed => 1,
213                                         junk_out => \@junk,
214                                           );
215                 if ($code eq "100") {
216                 $write_wait = 0;
217                 undef($code);
218                 goto INITIAL_READ;
219                 }
220                 else {
221                 $drop_connection++;
222                 last WRITE;
223                 # XXX should perhaps try to abort write in a nice way too
224                 }
225             }
226             }
227             if (defined($wbits) && $wbits =~ /[^\0]/) {
228             my $n = $socket->syswrite($$wbuf, length($$wbuf), $woffset);
229                     unless (defined $n) {
230                         die "write failed: $!" unless $!{EINTR} || $!{EAGAIN};
231                         $n = 0; # will retry write on the next round
232                     }
233                     elsif ($n == 0) {
234                 die "write failed: no bytes written";
235             }
236             $woffset += $n;
237
238             if (!$eof && $woffset >= length($$wbuf)) {
239                 # need to refill buffer from $content_ref code
240                 my $buf = &$content_ref();
241                 $buf = "" unless defined($buf);
242                 $eof++ unless length($buf);
243                 $buf = sprintf "%x%s%s%s", length($buf), $CRLF, $buf, $CRLF
244                 if $chunked;
245                 $wbuf = \$buf;
246                 $woffset = 0;
247             }
248             }
249         } # WRITE
250         }
```

```
251
252     ($code, $mess, @h) = $socket->read_response_headers(laxed => 1,
253      junk_out => \@junk)
254     unless $code;
255     ($code, $mess, @h) = $socket->read_response_headers(laxed => 1,
256      junk_out => \@junk)
257     if $code eq "100";
258
259     my $response = HTTP::Response->new($code, $mess);
260     my $peer_http_version = $socket->peer_http_version;
261     $response->protocol("HTTP/$peer_http_version");
262     {
263     local $HTTP::Headers::TRANSLATE_UNDERSCORE;
264     $response->push_header(@h);
265     }
266     $response->push_header("Client-Junk" => \@junk) if @junk;
267
268     $response->request($request);
269     $self->_get_sock_info($response, $socket);
270
271     if ($method eq "CONNECT") {
272     $response->{client_socket} = $socket; # so it can be picked up
273     return $response;
274     }
275
276     if (my @te = $response->remove_header('Transfer-Encoding')) {
277     $response->push_header('Client-Transfer-Encoding', \@te);
278     }
279     $response->push_header('Client-Response-Num', scalar
280      $socket->increment_response_count);
281
282     my $complete;
283     $response = $self->collect($arg, $response, sub {
284     my $buf = ""; #prevent use of uninitialized value in SSLeay.xs
285     my $n;
286       READ:
287     {
288         $n = $socket->read_entity_body($buf, $size);
289             unless (defined $n) {
290                 redo READ if $!{EINTR} || $!{EAGAIN};
291                 die "read failed: $!";
292             }
293         redo READ if $n == -1;
294     }
295     $complete++ if !$n;
296         return \$buf;
297     } );
298     $drop_connection++ unless $complete;
299
```

```
300     @h = $socket->get_trailers;
301     if (@h) {
302     local $HTTP::Headers::TRANSLATE_UNDERSCORE;
303     $response->push_header(@h);
304     }
305
306     # keep-alive support
307     unless ($drop_connection) {
308     if (my $conn_cache = $self->{ua}{conn_cache}) {
309         my %connection = map { (lc($_) => 1) }
310                     split(/\s*,\s*/, ($response->header("Connection") || ""));
311         if (($peer_http_version eq "1.1" && !$connection{close}) ||
312         $connection{"keep-alive"})
313         {
314         $conn_cache->deposit($self->socket_type, "$host:$port", $socket);
315         }
316     }
317     }
318
319     $response;
320 }
```

The code starts by checking if the input parameters are valid, whether there's a proxy or not, and then on line 35, you can see a call to the *_new_socket()* function inside the same module. This function calls the *IO::Socket::INET*[9] module in order to create a new socket, which is a computer's way to open a logical connection to a network resource, such as a web server. After that, the function adds various information on that socket, including the version of the HTTP protocol used on line 40, and the default headers on line 49. It defines how much content it's sending on line 75, then actually writes out the headers on the socket starting at line 100 using the *syswrite()* function.

After that, it reads from the socket starting at line 130 using *sysread()*, and starts parsing the headers at line 252. The *syswrite()* and *sysread()* functions are built-in Perl functions that can read and write to a stream, such as the opened network socket. Of more interest is the *IO::Socket::INET* module which builds upon *IO::Socket*[10]. Let's go down another abstraction level.

2.5 Fifth layer: I/O Sockets

IO simply means input and output. With the *IO::Socket* module we're now deep enough for things to be generalized greatly. We're no longer talking about XML data, or even HTTP connections. We're now deep

enough on the OSI model[11] to be talking directly to network drivers in your operating system. This is where an Internet socket, which is an endpoint of an inter-process communication flow across a computer network, is created.

The first thing the previous layer did was call the *binding()* function of the *IO::Socket* module, which in turn calls the *ioctl()* Perl function, which is a system call to set whether the socket will be blocking (stops execution of the code while waiting for all the data to be sent or received) or non-blocking (allows the code to continue while partial data is on the line). Then, the actual socket is created.

To see what actually goes on when creating a socket, let's look at the *configure()* function which makes all the interesting initial network calls:

```
1  sub configure {
2      my($sock,$arg) = @_;
3      my($lport,$rport,$laddr,$raddr,$proto,$type);
4
5
6      $arg->{LocalAddr} = $arg->{LocalHost}
7      if exists $arg->{LocalHost} && !exists $arg->{LocalAddr};
8
9      ($laddr,$lport,$proto) = _sock_info($arg->{LocalAddr},
10                                         $arg->{LocalPort},
11                                         $arg->{Proto})
12             or return _error($sock, $!, $@);
13
14     $laddr = defined $laddr ? inet_aton($laddr)
15                             : INADDR_ANY;
16
17     return _error($sock, $EINVAL, "Bad hostname '",$arg->{LocalAddr},"'")
18     unless(defined $laddr);
19
20     $arg->{PeerAddr} = $arg->{PeerHost}
21     if exists $arg->{PeerHost} && !exists $arg->{PeerAddr};
22
23     unless(exists $arg->{Listen}) {
24     ($raddr,$rport,$proto) = _sock_info($arg->{PeerAddr},
25                                         $arg->{PeerPort},
26                                         $proto)
27             or return _error($sock, $!, $@);
28     }
29
30     $proto ||= _get_proto_number('tcp');
31
32     $type = $arg->{Type} || $socket_type{lc _get_proto_name($proto)};
33
```

```
34      my @raddr = ();
35
36      if(defined $raddr) {
37      @raddr = $sock->_get_addr($raddr, $arg->{MultiHomed});
38      return _error($sock, $EINVAL, "Bad hostname '",$arg->{PeerAddr},"'")
39          unless @raddr;
40      }
41
42      while(1) {
43
44      $sock->socket(AF_INET, $type, $proto) or
45          return _error($sock, $!, "$!");
46
47          if (defined $arg->{Blocking}) {
48          defined $sock->blocking($arg->{Blocking})
49          or return _error($sock, $!, "$!");
50      }
51
52      if ($arg->{Reuse} || $arg->{ReuseAddr}) {
53          $sock->sockopt(SO_REUSEADDR,1) or
54              return _error($sock, $!, "$!");
55      }
56
57      if ($arg->{ReusePort}) {
58          $sock->sockopt(SO_REUSEPORT,1) or
59              return _error($sock, $!, "$!");
60      }
61
62      if ($arg->{Broadcast}) {
63          $sock->sockopt(SO_BROADCAST,1) or
64              return _error($sock, $!, "$!");
65      }
66
67      if($lport || ($laddr ne INADDR_ANY) || exists $arg->{Listen}) {
68          $sock->bind($lport || 0, $laddr) or
69              return _error($sock, $!, "$!");
70      }
71
72      if(exists $arg->{Listen}) {
73          $sock->listen($arg->{Listen} || 5) or
74          return _error($sock, $!, "$!");
75          last;
76      }
77
78      # don't try to connect unless we're given a PeerAddr
79      last unless exists($arg->{PeerAddr});
80
81          $raddr = shift @raddr;
82
```

```
83      return _error($sock, $EINVAL, 'Cannot determine remote port')
84          unless($rport || $type == SOCK_DGRAM || $type == SOCK_RAW);
85
86      last
87          unless($type == SOCK_STREAM || defined $raddr);
88
89      return _error($sock, $EINVAL, "Bad hostname '",$arg->{PeerAddr},"'")
90          unless defined $raddr;
91
92      undef $@;
93          if ($sock->connect(pack_sockaddr_in($rport, $raddr))) {
94              return $sock;
95          }
96
97      return _error($sock, $!, $@ || "Timeout")
98          unless @raddr;
99      }
100
101     $sock;
102 }
```

Here we're dealing with actual network code. The first thing this function does is assign all the values needed for the socket, then it calls *_sock_info()* on line 24. This function uses various string parsing utilities to determine whether the protocol, host and port are valid. Here we're no longer talking about HTTP, but instead TCP, which is what HTTP rides over in the OSI model. The host should be a valid host name or IP address, and the port should be a valid port. HTTP runs on port 80, HTTPS on 443, and so on. TCP/IP is the basis for any stream connection over the Internet.

Once that's done, line 30 calls *_get_proto_number()* to get the actual number assigned to TCP, something we'll come back to in the next section. Similarly, line 37 converts a host name into an IP address if need be. The actual socket is created on line 44, which is a call to the function of the same name from *IO::Socket*. After that, various flags are set, potential errors are handled, and the socket is returned.

2.6 Sixth layer: Kernel drivers

So far we've gone through Perl code, but now, our code is talking directly to the system. But how is the operating system, whether it's Windows, OS X or Unix, actually sending bits over the network? The answer is the network driver. At this point, every driver will be different based on the OS you use along with your network card. This is

the beauty of APIs. Just like Perl modules give us APIs to their own functions, operating systems have system calls to talk to each type of hardware.

We're going to go briefly over what happens at the system level on Linux, because the source code is freely available. For an in-depth lecture I suggest the Linux Kernel Networking[12] presentation by Rami Rosen. The source of the socket code is available in socket.c[13] in the Kernel source tree. Here is the code that the Kernel uses to allocate a socket to an application:

```
static struct socket *sock_alloc(void)
{
        struct inode *inode;
        struct socket *sock;

        inode = new_inode_pseudo(sock_mnt->mnt_sb);
        if (!inode)
                return NULL;

        sock = SOCKET_I(inode);

        kmemcheck_annotate_bitfield(sock, type);
        inode->i_ino = get_next_ino();
        inode->i_mode = S_IFSOCK | S_IRWXUGO;
        inode->i_uid = current_fsuid();
        inode->i_gid = current_fsgid();
        inode->i_op = &sockfs_inode_ops;

        this_cpu_add(sockets_in_use, 1);
        return sock;
}
```

On Linux, socket are linked to inodes, which is an index on the file system. The Kernel keeps track of these inodes. The *this_cpu_add()* function is simply a way to add the number of sockets to an internal list. Finally, *connect()* is also defined in that file as a system call:

```
SYSCALL_DEFINE3(connect, int, fd, struct sockaddr __user *, uservaddr,
                int, addrlen)
{
        struct socket *sock;
        struct sockaddr_storage address;
        int err, fput_needed;

        sock = sockfd_lookup_light(fd, &err, &fput_needed);
        if (!sock)
                goto out;
        err = move_addr_to_kernel(uservaddr, addrlen, &address);
```

```
12          if (err < 0)
13                  goto out_put;
14
15          err =
16              security_socket_connect(sock, (struct sockaddr *)&address, addrlen);
17          if (err)
18                  goto out_put;
19
20          err = sock->ops->connect(sock, (struct sockaddr *)&address, addrlen,
21                                   sock->file->f_flags);
22  out_put:
23          fput_light(sock->file, fput_needed);
24  out:
25          return err;
26  }
```

The socket code is just part of the story however. Once the CPU knows how to accept socket calls, it needs to know what to send to the actual hardware, and that's done with network drivers. There are hundreds of drivers for everything from Ethernet cards, fiber optic connections, wireless, and so on. You can view the source of the fairly popular Intel PRO/100 Ethernet Card in e100.c[14] in the source tree.

If you dig into that code, you might realize that abstraction doesn't end here. Take a look for example at the *e100_write_flush()* function:

```
1  static inline void e100_write_flush(struct nic *nic)
2  {
3      (void)ioread8(&nic->csr->scb.status);
4  }
```

Here you can see that the driver calls a function called *ioread8()* which is a Kernel call that is defined in *iomap.h* which in turn calls *readb()* based on the architecture that Linux runs on, whether it's x86, arm, alpha and so on. For example, here is an implementation of *readb()* for the hexagon platform:

```
1  static inline u8 readb(const volatile void __iomem *addr)
2  {
3          u8 val;
4          asm volatile(
5                  "%0 = memb(%1);"
6                  : "=&r" (val)
7                  : "r" (addr)
8          );
9          return val;
10 }
```

This is what manually copies each character, in the form of bytes, to and from the network hardware. Seeing as this is assembly code, we're officially as low on the abstraction stack as we can go. After that, it's nothing but assembly commands going back and forth between the operating system, the CPU and the various hardware in your machine.

3 Network traffic

So far, we've been through five different layers of Perl code and a sixth layer of Kernel functions in order to find out what a single line did. We went from parsing XML data, to fetching raw data on an HTTP connection from a web server, down to the actual network sockets used to read and write at the system level. Now, it's time to see what the data actually is when looked at directly on the network.

To do this, I'll be using a packet capture utility to see exactly what is written on the socket by all of this code. First, this is the actual packet, in bytes (converted from binary to hexadecimal to make it more readable), as sent over the wire:

```
00 01 96 6A 21 02 00 04 23 44 1C DD 08 00 45 00
00 CB 48 28 40 00 80 06 33 F6 C0 A8 00 05 C0 A8
00 01 FC ED 00 50 4C 8E C0 3A 09 C4 F6 D1 50 18
01 00 CC BE 00 00 47 45 54 20 68 74 74 70 3A 2F
2F 77 77 77 2E 72 65 64 64 69 74 2E 63 6F 6D 2F
2E 72 73 73 20 48 54 54 50 2F 31 2E 31 0D 0A 54
45 3A 20 64 65 66 6C 61 74 65 2C 67 7A 69 70 3B
71 3D 30 2E 33 0D 0A 43 6F 6E 6E 65 63 74 69 6F
6E 3A 20 54 45 2C 20 63 6C 6F 73 65 0D 0A 41 63
63 65 70 74 2D 45 6E 63 6F 64 69 6E 67 3A 20 67
7A 69 70 0D 0A 48 6F 73 74 3A 20 77 77 77 2E 72
65 64 64 69 74 2E 63 6F 6D 0D 0A 55 73 65 72 2D
41 67 65 6E 74 3A 20 58 4D 4C 3A 3A 46 65 65 64
2F 30 2E 35 32 0D 0A 0D 0A
```

Oviously this is fairly pointless to us, so let's use a network utility to analyze it:

```
  Frame: Number = 183, Captured Frame Length = 217, MediaType = ETHERNET
- Ethernet: Etype = Internet IP (IPv4),DestinationAddress:[00-01-96-6A-21-02],
SourceAddress:[00-04-23-44-1C-DD]
  - DestinationAddress: 000196 6A2102 [00-01-96-6A-21-02]
     Rsv: (001001..)
     UL: (......0.) Universally Administered Address
     IG: (.......0) Individual address (unicast)
```

```
  - SourceAddress: 000423 441CDD [00-04-23-44-1C-DD]
     Rsv: (000000..)
     UL: (......0.) Universally Administered Address
     IG: (.......0) Individual address (unicast)
    EthernetType: Internet IP (IPv4), 2048(0x800)
- Ipv4: Src = 192.168.0.5, Dest = 192.168.0.1, Next Protocol = TCP, Packet ID =
18472, Total IP Length = 203
  - Versions: IPv4, Internet Protocol; Header Length = 20
     Version:    (0100....) IPv4, Internet Protocol
     HeaderLength: (....0101) 20 bytes (0x5)
  - DifferentiatedServicesField: DSCP: 0, ECN: 0
     DSCP: (000000..) Differentiated services codepoint 0
     ECT: (......0.) ECN-Capable Transport not set
     CE:  (.......0) ECN-CE not set
    TotalLength: 203 (0xCB)
    Identification: 18472 (0x4828)
  - FragmentFlags: 16384 (0x4000)
     Reserved: (0..............)
     DF:      (.1.............) Do not fragment
     MF:      (..0............) This is the last fragment
     Offset:  (...0000000000000) 0
    TimeToLive: 128 (0x80)
    NextProtocol: TCP, 6(0x6)
    Checksum: 13302 (0x33F6)
    SourceAddress: 192.168.0.5
    DestinationAddress: 192.168.0.1
- Tcp: Flags=...AP..., SrcPort=64749, DstPort=HTTP(80), PayloadLen=163,
Seq=1284423738 - 1284423901, Ack=163903221, Win=256 (scale factor 0x8) = 65536
    SrcPort: 64749
    DstPort: HTTP(80)
    SequenceNumber: 1284423738 (0x4C8EC03A)
    AcknowledgementNumber: 163903221 (0x9C4F7D1)
  - DataOffset: 80 (0x50)
     DataOffset: (0101....) 20 bytes
     Reserved: (....000.)
     NS:       (.......0) Nonce Sum not significant
  - Flags: ...AP...
     CWR:   (0.......) CWR not significant
     ECE:   (.0......) ECN-Echo not significant
     Urgent: (..0.....) Not Urgent Data
     Ack:   (...1....) Acknowledgement field significant
     Push:  (....1...) Push Function
     Reset: (.....0..) No Reset
     Syn:   (......0.) Not Synchronize sequence numbers
     Fin:   (.......0) Not End of data
    Window: 256 (scale factor 0x8) = 65536
    Checksum: 0xCCBE, Disregarded
    UrgentPointer: 0 (0x0)
    TCPPayload: SourcePort = 64749, DestinationPort = 80
```

```
57 - Http: Request, GET http://www.reddit.com/.rss
58     Command: GET
59   - URI: http://www.reddit.com/.rss
60      Location: http://www.reddit.com/.rss
61     ProtocolVersion: HTTP/1.1
62     TE: deflate,gzip;q=0.3
63     Connection: TE, close
64     Accept-Encoding: gzip
65     Host: www.reddit.com
66     UserAgent: XML::Feed/0.52
67     HeaderEnd: CRLF
```

This tree of information was generated by the Microsoft Network Monitor, but you can get such information from Wireshark, or any other packet capture utility. All of this represents a single packet. Needless to say, it would be quite a bit of work to generate something like that for each and every packet your application wants to send over the network, hence all the layers of abstraction we've been through. Let's take a look at what's contained here, so we can relate to the functions we've seen in the previous sections.

On line 1, we see that this is an Ethernet frame, so we know the driver that handled this request is an Ethernet driver. On line 2, we have the start of the Ethernet header. This is entirely filled up by the driver itself, including the MAC addresses of the source and destination. Then on line 13, we have the header for the IP part of TCP/IP, namely the source and destination addresses. In our case, we're dealing with IPv4 addresses. You can see there are a lot of flags, most of which have default values, and those are assigned by the socket code in Perl modules. If you notice line 30, the protocol number for TCP is actually 6, something we've seen in one of the previous layers.

Line 34 starts the TCP part of TCP/IP, which defines a stream connection. The destination port is 80, and then sequence numbers are shown, which is a way the system keeps track of packets. Line 46 starts TCP flags which are set by the various modules we've covered, again most are set to default values. Finally, we have the HTTP request starting at line 57. These lines are much higher in the stack than the previous parts. Here we have settings that can actually be set by accessible Perl functions. Line 57 has the URL of the HTTP request, and line 66 has the user agent, something we've seen as well, in our case set to the name of the library, *XML::Feed*, along with the version.

4 Conclusion

In this experiment, we started with a single line of code, and went down through the various Perl modules, down to C code for the operating system, and down onto the network to see exactly what went on from this one command. As you may have noticed, things get complicated very quickly. It's interesting to note that nothing we've seen is a black box, meaning that if you really wanted to, you could recreate the actual packet that was shown in the previous section. In Perl, that would require you to access the *IO::Socket* module directly which isn't all that difficult to do, and there are even modules for deeper coding. If you're interested in socket coding in Perl I recommend the Perl Socket Programming tutorial[15].

Hopefully this has been enlightening, or at least entertaining. As you can see, abstraction is everywhere in modern day coding. This has a lot of advantages, but it's good to sometimes break through those layers and explore that lies beneath.

5 References

References

[1] XML::Feed - Syndication feed parser and auto-discovery
http://search.cpan.org/~davecross/XML-Feed-0.52/lib/XML/Feed/Entry.pm

[2] LWP::UserAgent - Web user agent class
http://search.cpan.org/dist/libwww-perl/lib/LWP/UserAgent.pm

[3] URI::Fetch - Smart URI fetching/caching
http://search.cpan.org/~neilb/URI-Fetch-0.10/lib/URI/Fetch.pm

[4] HTTP::Request - HTTP style request message
http://search.cpan.org/~gaas/HTTP-Message-6.06/lib/HTTP/Request.pm

[5] URI::Fetch::Response - Feed response for URI::Fetch
http://search.cpan.org/~btrott/URI-Fetch-0.09/lib/URI/Fetch/Response.pm

[6] HTTP::Message - HTTP style message (base class)
http://search.cpan.org/~gaas/HTTP-Message-6.06/lib/HTTP/Message.pm

[7] LWP::Protocol - Base class for LWP protocols
http://search.cpan.org/~mschilli/libwww-perl/lib/LWP/Protocol.pm

[8] LWP::Protocol::http - Source code
https://metacpan.org/source/GAAS/libwww-perl-6.03/lib/LWP/Protocol/http.pm

[9] IO::Socket::INET - Object interface for AF_INET domain sockets
http://search.cpan.org/~gbarr/IO-1.25/lib/IO/Socket/INET.pm

[10] IO::Socket - Object interface to socket communications
http://search.cpan.org/~gbarr/IO-1.25/lib/IO/Socket.pm

[11] Wikipedia: OSI model
http://en.wikipedia.org/wiki/OSI_model

[12] Rami Rosen: Linux Kernel Networking
http://haifux.org/hebrew/lectures/217/netLec5.pdf

[13] source.c: Linux Cross Reference
http://lxr.free-electrons.com/source/net/socket.c

[14] e100.c: Linux Kernel Source Tree
https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/drivers/net/ethernet/intel/e100.c

[15] Tutorialspoint: Perl Socket Programming
http://www.tutorialspoint.com/perl/perl_socket_programming.htm