NAME

AnyEvent::Intro − an introductory tutorial to AnyEvent

Introduction to AnyEvent

This is a tutorial that will introduce you to the features of AnyEvent.

The first part introduces the core AnyEvent module (after swamping you a bit in evangelism), which might already provide all you ever need: If you are only interested in AnyEvent's event handling capabilities, read no further.

The second part focuses on network programming using sockets, for which AnyEvent offers a lot of support you can use, and a lot of workarounds around portability quirks.

What is AnyEvent?

If you don't care for the whys and want to see code, skip this section!

AnyEvent is first of all just a framework to do event-based programming. Typically such frameworks are an all-or-nothing thing: If you use one such framework, you can't (easily, or even at all) use another in the same program.

AnyEvent is different − it is a thin abstraction layer on top of other event loops, just like DBI is an abstraction of many different database APIs. Its main purpose is to move the choice of the underlying framework (the event loop) from the module author to the program author using the module.

That means you can write code that uses events to control what it does, without forcing other code in the same program to use the same underlying framework as you do − i.e. you can create a Perl module that is event-based using AnyEvent, and users of that module can still choose between using Gtk2, Tk, Event (or run inside Irssi or rxvt-unicode) or any other supported event loop. AnyEvent even comes with its own pure-perl event loop implementation, so your code works regardless of other modules that might or might not be installed. The latter is important, as AnyEvent does not have any hard dependencies to other modules, which makes it easy to install, for example, when you lack a C compiler. No matter what environment, AnyEvent will just cope with it.

A typical limitation of existing Perl modules such as Net::IRC is that they come with their own event loop: In Net::IRC, a program which uses it needs to start the event loop of Net::IRC. That means that one cannot integrate this module into a Gtk2 GUI for instance, as that module, too, enforces the use of its own event loop (namely Glib).

Another example is LWP: it provides no event interface at all. It's a pure blocking HTTP (and FTP etc.) client library, which usually means that you either have to start another process or have to fork for a HTTP request, or use threads (e.g. Coro::LWP), if you want to do something else while waiting for the request to finish.

The motivation behind these designs is often that a module doesn't want to depend on some complicated XS-module (Net::IRC), or that it doesn't want to force the user to use some specific event loop at all (LWP), out of fear of severely limiting the usefulness of the module: If your module requires Glib, it will not run in a Tk program.

AnyEvent solves this dilemma, by **not** forcing module authors to either:

− write their own event loop (because it guarantees the availability of an event loop everywhere − even on windows with no extra modules installed).
− choose one specific event loop (because AnyEvent works with most event loops available for Perl).

If the module author uses AnyEvent for all his (or her) event needs (IO events, timers, signals, ...) then all other modules can just use his module and don't have to choose an event loop or adapt to his event loop. The choice of the event loop is ultimately made by the program author who uses all the modules and writes the main program. And even there he doesn't have to choose, he can just let AnyEvent choose the most efficient event loop available on the system.

Read more about this in the main documentation of the AnyEvent module.

Introduction to Event-Based Programming

So what exactly is programming using events? It quite simply means that instead of your code actively waiting for something, such as the user entering something on STDIN:

```
$| = 1; print "enter your name> ";

my $name = <STDIN>;
```

You instead tell your event framework to notify you in the event of some data being available on STDIN, by using a callback mechanism:

```
use AnyEvent;

$| = 1; print "enter your name> ";

my $name;

my $wait_for_input = AnyEvent->io (
   fh   => \*STDIN, # which file handle to check
   poll => "r",     # which event to wait for ("r"ead data)
   cb   => sub {    # what callback to execute
      $name = <STDIN>; # read it
   }
);

# do something else here
```

Looks more complicated, and surely is, but the advantage of using events is that your program can do something else instead of waiting for input (side note: combining AnyEvent with a thread package such as Coro can recoup much of the simplicity, effectively getting the best of two worlds).

Waiting as done in the first example is also called ‘‘blocking’’ the process because you ‘‘block’’/keep your process from executing anything else while you do so.

The second example avoids blocking by only registering interest in a read event, which is fast and doesn’t block your process. The callback will be called only when data is available and can be read without blocking.

The ‘‘interest’’ is represented by an object returned by `AnyEvent->io` called a ‘‘watcher’’ object – thus named because it ‘‘watches’’ your file handle (or other event sources) for the event you are interested in.

In the example above, we create an I/O watcher by calling the `AnyEvent->io` method. A lack of further interest in some event is expressed by simply forgetting about its watcher, for example by `undef`–ing the only variable it is stored in. AnyEvent will automatically clean up the watcher if it is no longer used, much like Perl closes your file handles if you no longer use them anywhere.

A short note on callbacks

A common issue that hits people is the problem of passing parameters to callbacks. Programmers used to languages such as C or C++ are often used to a style where one passes the address of a function (a function reference) and some data value, e.g.:

```
sub callback {
   my ($arg) = @_;

   $arg->method;
}

my $arg = ...;

call_me_back_later \&callback, $arg;
```

This is clumsy, as the place where behaviour is specified (when the callback is registered) is often far away from the place where behaviour is implemented. It also doesn't use Perl syntax to invoke the code. There is also an abstraction penalty to pay as one has to *name* the callback, which often is unnecessary and leads to nonsensical or duplicated names.

In Perl, one can specify behaviour much more directly by using *closures*. Closures are code blocks that take a reference to the enclosing scope(s) when they are created. This means lexical variables in scope when a closure is created can be used inside the closure:

```
my $arg = ...;

call_me_back_later sub { $arg->method };
```

Under most circumstances, closures are faster, use fewer resources and result in much clearer code than the traditional approach. Faster, because parameter passing and storing them in local variables in Perl is relatively slow. Fewer resources, because closures take references to existing variables without having to create new ones, and clearer code because it is immediately obvious that the second example calls the `method` method when the callback is invoked.

Apart from these, the strongest argument for using closures with AnyEvent is that AnyEvent does not allow passing parameters to the callback, so closures are the only way to achieve that in most cases :–>

A little hint to catch mistakes

AnyEvent does not check the parameters you pass in, at least not by default. to enable checking, simply start your program with `AE_STRICT=1` in the environment, or put `use AnyEvent::Strict` near the top of your program:

```
AE_STRICT=1 perl myprogram
```

You can find more info on this and additional debugging aids later in this introduction.

Condition Variables

Back to the I/O watcher example: The code is not yet a fully working program, and will not work as-is. The reason is that your callback will not be invoked out of the blue; you have to run the event loop first. Also, event-based programs need to block sometimes too, such as when there is nothing to do, and everything is waiting for new events to arrive.

In AnyEvent, this is done using condition variables. Condition variables are named ‘‘condition variables’’ because they represent a condition that is initially false and needs to be fulfilled.

You can also call them ‘‘merge points’’, ‘‘sync points’’, ‘‘rendezvous ports’’ or even callbacks and many other things (and they are often called these names in other frameworks). The important point is that you can create them freely and later wait for them to become true.

Condition variables have two sides – one side is the ‘‘producer’’ of the condition (whatever code detects and flags the condition), the other side is the ‘‘consumer’’ (the code that waits for that condition).

In our example in the previous section, the producer is the event callback and there is no consumer yet – let's change that right now:

```
use AnyEvent;

$| = 1; print "enter your name> ";

my $name;

my $name_ready = AnyEvent->condvar;

my $wait_for_input = AnyEvent->io (
   fh   => \*STDIN,
   poll => "r",
   cb   => sub {
```

```
      $name = <STDIN>;
      $name_ready->send;
   }
);

# do something else here

# now wait until the name is available:
$name_ready->recv;

undef $wait_for_input; # watcher no longer needed

print "your name is $name\n";
```

This program creates an AnyEvent condvar by calling the `AnyEvent->condvar` method. It then creates a watcher as usual, but inside the callback it `send`s the `$name_ready` condition variable, which causes whoever is waiting on it to continue.

The ''whoever'' in this case is the code that follows, which calls `$name_ready->recv`: The producer calls `send`, the consumer calls `recv`.

If there is no `$name` available yet, then the call to `$name_ready->recv` will halt your program until the condition becomes true.

As the names `send` and `recv` imply, you can actually send and receive data using this, for example, the above code could also be written like this, without an extra variable to store the name in:

```
use AnyEvent;

$| = 1; print "enter your name> ";

my $name_ready = AnyEvent->condvar;

my $wait_for_input = AnyEvent->io (
   fh => \*STDIN, poll => "r",
   cb => sub { $name_ready->send (scalar <STDIN>) }
);

# do something else here

# now wait and fetch the name
my $name = $name_ready->recv;

undef $wait_for_input; # watcher no longer needed

print "your name is $name\n";
```

You can pass any number of arguments to `send`, and every subsequent call to `recv` will return them.

The "main loop"

Most event-based frameworks have something called a ''main loop'' or ''event loop run function'' or something similar.

Just like in `recv` AnyEvent, these functions need to be called eventually so that your event loop has a chance of actually looking for the events you are interested in.

For example, in a Gtk2 program, the above example could also be written like this:

```
use Gtk2 -init;
use AnyEvent;
```

```
##########################################
# create a window and some label

my $window = new Gtk2::Window "toplevel";
$window->add (my $label = new Gtk2::Label "soon replaced by name");

$window->show_all;

##########################################
# do our AnyEvent stuff

$| = 1; print "enter your name> ";

my $name_ready = AnyEvent->condvar;

my $wait_for_input = AnyEvent->io (
   fh => \*STDIN, poll => "r",
   cb => sub {
      # set the label
      $label->set_text (scalar <STDIN>);
      print "enter another name> ";
   }
);

##########################################
# Now enter Gtk2's event loop

main Gtk2;
```

No condition variable anywhere in sight – instead, we just read a line from STDIN and replace the text in the label. In fact, since nobody `undefs` `$wait_for_input` you can enter multiple lines.

Instead of waiting for a condition variable, the program enters the Gtk2 main loop by calling `Gtk2->main`, which will block the program and wait for events to arrive.

This also shows that AnyEvent is quite flexible – you didn't have to do anything to make the AnyEvent watcher use Gtk2 (actually Glib) – it just worked.

Admittedly, the example is a bit silly – who would want to read names from standard input in a Gtk+ application? But imagine that instead of doing that, you make an HTTP request in the background and display its results. In fact, with event-based programming you can make many HTTP requests in parallel in your program and still provide feedback to the user and stay interactive.

And in the next part you will see how to do just that – by implementing an HTTP request, on our own, with the utility modules AnyEvent comes with.

Before that, however, let's briefly look at how you would write your program using only AnyEvent, without ever calling some other event loop's run function.

In the example using condition variables, we used those to start waiting for events, and in fact, condition variables are the solution:

```
my $quit_program = AnyEvent->condvar;

# create AnyEvent watchers (or not) here

$quit_program->recv;
```

If any of your watcher callbacks decide to quit (this is often called an ‘‘unloop’’ in other frameworks), they

can just call `$quit_program->send`. Of course, they could also decide not to and call `exit` instead, or they could decide never to quit (e.g. in a long-running daemon program).

If you don't need some clean quit functionality and just want to run the event loop, you can do this:

```
AnyEvent->condvar->recv;
```

And this is, in fact, the closest to the idea of a main loop run function that AnyEvent offers.

Timers and other event sources

So far, we have used only I/O watchers. These are useful mainly to find out whether a socket has data to read, or space to write more data. On sane operating systems this also works for console windows/terminals (typically on standard input), serial lines, all sorts of other devices, basically almost everything that has a file descriptor but isn't a file itself. (As usual, ''sane'' excludes windows – on that platform you would need different functions for all of these, complicating code immensely – think ''socket only'' on windows).

However, I/O is not everything – the second most important event source is the clock. For example when doing an HTTP request you might want to time out when the server doesn't answer within some predefined amount of time.

In AnyEvent, timer event watchers are created by calling the `AnyEvent->timer` method:

```
use AnyEvent;

my $cv = AnyEvent->condvar;

my $wait_one_and_a_half_seconds = AnyEvent->timer (
   after => 1.5,  # after how many seconds to invoke the cb?
   cb    => sub { # the callback to invoke
      $cv->send;
   },
);

# can do something else here

# now wait till our time has come
$cv->recv;
```

Unlike I/O watchers, timers are only interested in the amount of seconds they have to wait. When (at least) that amount of time has passed, AnyEvent will invoke your callback.

Unlike I/O watchers, which will call your callback as many times as there is data available, timers are normally one-shot: after they have ''fired'' once and invoked your callback, they are dead and no longer do anything.

To get a repeating timer, such as a timer firing roughly once per second, you can specify an `interval` parameter:

```
my $once_per_second = AnyEvent->timer (
   after => 0,    # first invoke ASAP
   interval => 1, # then invoke every second
   cb    => sub { # the callback to invoke
      $cv->send;
   },
);
```

More esoteric sources

AnyEvent also has some other, more esoteric event sources you can tap into: signal, child and idle watchers.

Signal watchers can be used to wait for ''signal events'', which means your process was sent a signal (such as `SIGTERM` or `SIGUSR1`).

Child-process watchers wait for a child process to exit. They are useful when you fork a separate process and need to know when it exits, but you do not want to wait for that by blocking.

Idle watchers invoke their callback when the event loop has handled all outstanding events, polled for new events and didn't find any, i.e., when your process is otherwise idle. They are useful if you want to do some non-trivial data processing that can be done when your program doesn't have anything better to do.

All these watcher types are described in detail in the main AnyEvent manual page.

Sometimes you also need to know what the current time is: `AnyEvent->now` returns the time the event toolkit uses to schedule relative timers, and is usually what you want. It is often cached (which means it can be a bit outdated). In that case, you can use the more costly `AnyEvent->time` method which will ask your operating system for the current time, which is slower, but also more up to date.

Network programming and AnyEvent

So far you have seen how to register event watchers and handle events.

This is a great foundation to write network clients and servers, and might be all that your module (or program) ever requires, but writing your own I/O buffering again and again becomes tedious, not to mention that it attracts errors.

While the core AnyEvent module is still small and self-contained, the distribution comes with some very useful utility modules such as AnyEvent::Handle, AnyEvent::DNS and AnyEvent::Socket. These can make your life as a non-blocking network programmer a lot easier.

Here is a quick overview of these three modules:

AnyEvent::DNS

This module allows fully asynchronous DNS resolution. It is used mainly by AnyEvent::Socket to resolve hostnames and service ports for you, but is a great way to do other DNS resolution tasks, such as reverse lookups of IP addresses for log files.

AnyEvent::Handle

This module handles non-blocking IO on (socket–, pipe– etc.) file handles in an event based manner. It provides a wrapper object around your file handle that provides queueing and buffering of incoming and outgoing data for you.

It also implements the most common data formats, such as text lines, or fixed and variable-width data blocks.

AnyEvent::Socket

This module provides you with functions that handle socket creation and IP address magic. The two main functions are `tcp_connect` and `tcp_server`. The former will connect a (streaming) socket to an internet host for you and the later will make a server socket for you, to accept connections.

This module also comes with transparent IPv6 support, this means: If you write your programs with this module, you will be IPv6 ready without doing anything special.

It also works around a lot of portability quirks (especially on the windows platform), which makes it even easier to write your programs in a portable way (did you know that windows uses different error codes for all socket functions and that Perl does not know about these? That "Unknown error 10022" (which is `WSAEINVAL`) can mean that our `connect` call was successful? That unsuccessful TCP connects might never be reported back to your program? That `WSAEINPROGRESS` means your `connect` call was ignored instead of being in progress? AnyEvent::Socket works around all of these Windows/Perl bugs for you).

Implementing a parallel finger client with non-blocking connects and AnyEvent::Socket

The finger protocol is one of the simplest protocols in use on the internet. Or in use in the past, as almost nobody uses it anymore.

It works by connecting to the finger port on another host, writing a single line with a user name and then reading the finger response, as specified by that user. OK, RFC 1288 specifies a vastly more complex protocol, but it basically boils down to this:

```
# telnet freebsd.org finger
Trying 8.8.178.135...
Connected to freebsd.org (8.8.178.135).
Escape character is '^]'.
larry
Login: lile                             Name: Larry Lile
Directory: /home/lile                   Shell: /usr/local/bin/bash
No Mail.
Mail forwarded to: lile@stdio.com
No Plan.
```

So let's write a little AnyEvent function that makes a finger request:

```
use AnyEvent;
use AnyEvent::Socket;

sub finger($$) {
   my ($user, $host) = @_;

   # use a condvar to return results
   my $cv = AnyEvent->condvar;

   # first, connect to the host
   tcp_connect $host, "finger", sub {
      # the callback receives the socket handle - or nothing
      my ($fh) = @_
         or return $cv->send;

      # now write the username
      syswrite $fh, "$user\015\012";

      my $response;

      # register a read watcher
      my $read_watcher; $read_watcher = AnyEvent->io (
         fh   => $fh,
         poll => "r",
         cb   => sub {
            my $len = sysread $fh, $response, 1024, length $response;

            if ($len <= 0) {
               # we are done, or an error occurred, lets ignore the latter
               undef $read_watcher; # no longer interested
               $cv->send ($response); # send results
            }
         },
      );
   };

   # pass $cv to the caller
   $cv
}
```

That's a mouthful! Let's dissect this function a bit, first the overall function and execution flow:

```
sub finger($$) {
   my ($user, $host) = @_;

   # use a condvar to return results
   my $cv = AnyEvent->condvar;

   # first, connect to the host
   tcp_connect $host, "finger", sub {
      ...
   };

   $cv
}
```

This isn't too complicated, just a function with two parameters that creates a condition variable `$cv`, initiates a TCP connect to `$host`, and returns `$cv`. The caller can use the returned `$cv` to receive the finger response, but one could equally well pass a third argument, a callback, to the function.

Since we are programming event'ish, we do not wait for the connect to finish – it could block the program for a minute or longer!

Instead, we pass `tcp_connect` a callback to invoke when the connect is done. The callback is called with the socket handle as its first argument if the connect succeeds, and no arguments otherwise. The important point is that it will always be called as soon as the outcome of the TCP connect is known.

This style of programming is also called "continuation style": the "continuation" is simply the way the program continues – normally at the next line after some statement (the exception is loops or things like `return`). When we are interested in events, however, we instead specify the "continuation" of our program by passing a closure, which makes that closure the "continuation" of the program.

The `tcp_connect` call is like saying "return now, and when the connection is established or the attempt failed, continue there".

Now let's look at the callback/closure in more detail:

```
# the callback receives the socket handle - or nothing
my ($fh) = @_
   or return $cv->send;
```

The first thing the callback does is to save the socket handle in `$fh`. When there was an error (no arguments), then our instinct as expert Perl programmers would tell us to `die`:

```
my ($fh) = @_
   or die "$host: $!";
```

While this would give good feedback to the user (if he happens to watch standard error), our program would probably stop working here, as we never report the results to anybody, certainly not the caller of our `finger` function, and most event loops continue even after a `die`!

This is why we instead `return`, but also call `$cv->send` without any arguments to signal to the condvar consumer that something bad has happened. The return value of `$cv->send` is irrelevant, as is the return value of our callback. The `return` statement is used for the side effect of, well, returning immediately from the callback. Checking for errors and handling them this way is very common, which is why this compact idiom is so handy.

As the next step in the finger protocol, we send the username to the finger daemon on the other side of our connection (the kernel.org finger service doesn't actually wait for a username, but the net is running out of finger servers fast):

```
syswrite $fh, "$user\015\012";
```

Note that this isn't 100% clean socket programming – the socket could, for whatever reasons, not accept our data. When writing a small amount of data like in this example it doesn't matter, as a socket buffer is

almost always big enough for a mere "username", but for real-world cases you might need to implement some kind of write buffering – or use AnyEvent::Handle, which handles these matters for you, as shown in the next section.

What we *do* have to do is implement our own read buffer – the response data could arrive late or in multiple chunks, and we cannot just wait for it (event-based programming, you know?).

To do that, we register a read watcher on the socket which waits for data:

```
my $read_watcher; $read_watcher = AnyEvent->io (
   fh   => $fh,
   poll => "r",
```

There is a trick here, however: the read watcher isn't stored in a global variable, but in a local one – if the callback returns, it would normally destroy the variable and its contents, which would in turn unregister our watcher.

To avoid that, we refer to the watcher variable in the watcher callback. This means that, when the `tcp_connect` callback returns, perl thinks (quite correctly) that the read watcher is still in use – namely inside the inner callback – and thus keeps it alive even if nothing else in the program refers to it anymore (it is much like Baron Münchhausen keeping himself from dying by pulling himself out of a swamp).

The trick, however, is that instead of:

```
my $read_watcher = AnyEvent->io (...
```

The program does:

```
my $read_watcher; $read_watcher = AnyEvent->io (...
```

The reason for this is a quirk in the way Perl works: variable names declared with `my` are only visible in the *next* statement. If the whole `AnyEvent->io` call, including the callback, would be done in a single statement, the callback could not refer to the `$read_watcher` variable to `undef`ine it, so it is done in two statements.

Whether you'd want to format it like this is of course a matter of style. This way emphasizes that the declaration and assignment really are one logical statement.

The callback itself calls `sysread` for as many times as necessary, until `sysread` returns either an error or end-of-file:

```
cb   => sub {
   my $len = sysread $fh, $response, 1024, length $response;

   if ($len <= 0) {
```

Note that `sysread` has the ability to append data it reads to a scalar if we specify an offset, a feature which we make use of in this example.

When `sysread` indicates we are done, the callback `undef`ines the watcher and then `send`s the response data to the condition variable. All this has the following effects:

Undefining the watcher destroys it, as our callback was the only one still having a reference to it. When the watcher gets destroyed, it destroys the callback, which in turn means the `$fh` handle is no longer used, so that gets destroyed as well. The result is that all resources will be nicely cleaned up by perl for us.

Using the finger client

Now, we could probably write the same finger client in a simpler way if we used `IO::Socket::INET`, ignored the problem of multiple hosts and ignored IPv6 and a few other things that `tcp_connect` handles for us.

But the main advantage is that we can not only run this finger function in the background, we even can run multiple sessions in parallel, like this:

```
my $f1 = finger "kuriyama", "freebsd.org";
my $f2 = finger "icculus?listarchives=1", "icculus.org";
my $f3 = finger "mikachu", "icculus.org";

print "kuriyama's gpg key\n"    , $f1->recv, "\n";
print "icculus' plan archive\n" , $f2->recv, "\n";
print "mikachu's plan zomgn\n"  , $f3->recv, "\n";
```

It doesn't look like it, but in fact all three requests run in parallel. The code waits for the first finger request to finish first, but that doesn't keep it from executing them parallel: when the first `recv` call sees that the data isn't ready yet, it serves events for all three requests automatically, until the first request has finished.

The second `recv` call might either find the data is already there, or it will continue handling events until that is the case, and so on.

By taking advantage of network latencies, which allows us to serve other requests and events while we wait for an event on one socket, the overall time to do these three requests will be greatly reduced, typically all three are done in the same time as the slowest of the three requests.

By the way, you do not actually have to wait in the `recv` method on an AnyEvent condition variable – after all, waiting is evil – you can also register a callback:

```
$f1->cb (sub {
   my $response = shift->recv;
   # ...
});
```

The callback will be invoked only when `send` is called. In fact, instead of returning a condition variable you could also pass a third parameter to your finger function, the callback to invoke with the response:

```
sub finger($$$) {
   my ($user, $host, $cb) = @_;
```

How you implement it is a matter of taste – if you expect your function to be used mainly in an event-based program you would normally prefer to pass a callback directly. If you write a module and expect your users to use it "synchronously" often (for example, a simple http-get script would not really care much for events), then you would use a condition variable and tell them "simply `->recv` the data".

Problems with the implementation and how to fix them

To make this example more real-world-ready, we would not only implement some write buffering (for the paranoid, or maybe denial-of-service aware security expert), but we would also have to handle timeouts and maybe protocol errors.

Doing this quickly gets unwieldy, which is why we introduce AnyEvent::Handle in the next section, which takes care of all these details for you and lets you concentrate on the actual protocol.

Implementing simple HTTP and HTTPS GET requests with AnyEvent::Handle

The AnyEvent::Handle module has been hyped quite a bit in this document so far, so let's see what it really offers.

As finger is such a simple protocol, let's try something slightly more complicated: HTTP/1.0.

An HTTP GET request works by sending a single request line that indicates what you want the server to do and the URI you want to act it on, followed by as many "header" lines (`Header: data`, same as e–mail headers) as required for the request, followed by an empty line.

The response is formatted very similarly, first a line with the response status, then again as many header lines as required, then an empty line, followed by any data that the server might send.

Again, let's try it out with `telnet` (I condensed the output a bit – if you want to see the full response, do it yourself).

```
# telnet www.google.com 80
Trying 209.85.135.99...
Connected to www.google.com (209.85.135.99).
Escape character is '^]'.
GET /test HTTP/1.0

HTTP/1.0 404 Not Found
Date: Mon, 02 Jun 2008 07:05:54 GMT
Content-Type: text/html; charset=UTF-8

<html><head>
[...]
Connection closed by foreign host.
```

The `GET ...` and the empty line were entered manually, the rest of the telnet output is google's response, in this case a `404 not found` one.

So, here is how you would do it with `AnyEvent::Handle`:

```
sub http_get {
   my ($host, $uri, $cb) = @_;

   # store results here
   my ($response, $header, $body);

   my $handle; $handle = new AnyEvent::Handle
      connect  => [$host => 'http'],
      on_error => sub {
         $cb->("HTTP/1.0 500 $!");
         $handle->destroy; # explicitly destroy handle
      },
      on_eof   => sub {
         $cb->($response, $header, $body);
         $handle->destroy; # explicitly destroy handle
      };

   $handle->push_write ("GET $uri HTTP/1.0\015\012\015\012");

   # now fetch response status line
   $handle->push_read (line => sub {
      my ($handle, $line) = @_;
      $response = $line;
   });

   # then the headers
   $handle->push_read (line => "\015\012\015\012", sub {
      my ($handle, $line) = @_;
      $header = $line;
   });

   # and finally handle any remaining data as body
   $handle->on_read (sub {
      $body .= $_[0]->rbuf;
      $_[0]->rbuf = "";
   });
}
```

And now let's go through it step by step. First, as usual, the overall `http_get` function structure:

```
sub http_get {
   my ($host, $uri, $cb) = @_;

   # store results here
   my ($response, $header, $body);

   my $handle; $handle = new AnyEvent::Handle
      ... create handle object

   ... push data to write

   ... push what to expect to read queue
}
```

Unlike in the finger example, this time the caller has to pass a callback to `http_get`. Also, instead of passing a URL as one would expect, the caller has to provide the hostname and URI – normally you would use the `URI` module to parse a URL and separate it into those parts, but that is left to the inspired reader :)

Since everything else is left to the caller, all `http_get` does is initiate the connection by creating the AnyEvent::Handle object (which calls `tcp_connect` for us) and leave everything else to its callback.

The handle object is created, unsurprisingly, by calling the `new` method of AnyEvent::Handle:

```
   my $handle; $handle = new AnyEvent::Handle
      connect  => [$host => 'http'],
      on_error => sub {
         $cb->("HTTP/1.0 500 $!");
         $handle->destroy; # explicitly destroy handle
      },
      on_eof   => sub {
         $cb->($response, $header, $body);
         $handle->destroy; # explicitly destroy handle
      };
```

The `connect` argument tells AnyEvent::Handle to call `tcp_connect` for the specified host and service/port.

The `on_error` callback will be called on any unexpected error, such as a refused connection, or unexpected end-of-file while reading headers.

Instead of having an extra mechanism to signal errors, connection errors are signalled by crafting a special "response status line", like this:

```
HTTP/1.0 500 Connection refused
```

This means the caller cannot distinguish (easily) between locally-generated errors and server errors, but it simplifies error handling for the caller a lot.

The error callback also destroys the handle explicitly, because we are not interested in continuing after any errors. In AnyEvent::Handle callbacks you have to call `destroy` explicitly to destroy a handle. Outside of those callbacks you can just forget the object reference and it will be automatically cleaned up.

Last but not least, we set an `on_eof` callback that is called when the other side indicates it has stopped writing data, which we will use to gracefully shut down the handle and report the results. This callback is only called when the read queue is empty – if the read queue expects some data and the handle gets an EOF from the other side this will be an error – after all, you did expect more to come.

If you wanted to write a server using AnyEvent::Handle, you would use `tcp_accept` and then create the AnyEvent::Handle with the `fh` argument.

The write queue

The next line sends the actual request:

```
$handle->push_write ("GET $uri HTTP/1.0\015\012\015\012");
```

No headers will be sent (this is fine for simple requests), so the whole request is just a single line followed by an empty line to signal the end of the headers to the server.

The more interesting question is why the method is called `push_write` and not just write. The reason is that you can *always* add some write data without blocking, and to do this, AnyEvent::Handle needs some write queue internally – and `push_write` pushes some data onto the end of that queue, just like Perl's `push` pushes data onto the end of an array.

The deeper reason is that at some point in the future, there might be `unshift_write` as well, and in any case, we will shortly meet `push_read` and `unshift_read`, and it's usually easiest to remember if all those functions have some symmetry in their name. So `push` is used as the opposite of `unshift` in AnyEvent::Handle, not as the opposite of `pull` – just like in Perl.

Note that we call `push_write` right after creating the AnyEvent::Handle object, before it has had time to actually connect to the server. This is fine, pushing the read and write requests will queue them in the handle object until the connection has been established. Alternatively, we could do this "on demand" in the `on_connect` callback.

If `push_write` is called with more than one argument, then you can do *formatted* I/O. For example, this would JSON-encode your data before pushing it to the write queue:

```
$handle->push_write (json => [1, 2, 3]);
```

This pretty much summarises the write queue, there is little else to it.

Reading the response is far more interesting, because it involves the more powerful and complex *read queue*:

The read queue

The response consists of three parts: a single line with the response status, a single paragraph of headers ended by an empty line, and the request body, which is the remaining data on the connection.

For the first two, we push two read requests onto the read queue:

```
# now fetch response status line
$handle->push_read (line => sub {
   my ($handle, $line) = @_;
   $response = $line;
});

# then the headers
$handle->push_read (line => "\015\012\015\012", sub {
   my ($handle, $line) = @_;
   $header = $line;
});
```

While one can just push a single callback to parse all the data on the queue, formatted I/O really comes to our aid here, since there is a ready-made "read line" read type. The first read expects a single line, ended by `\015\012` (the standard end-of-line marker in internet protocols).

The second "line" is actually a single paragraph – instead of reading it line by line we tell `push_read` that the end-of-line marker is really `\015\012\015\012`, which is an empty line. The result is that the whole header paragraph will be treated as a single line and read. The word "line" is interpreted very freely, much like Perl itself does it.

Note that push read requests are pushed immediately after creating the handle object – since AnyEvent::Handle provides a queue we can push as many requests as we want, and AnyEvent::Handle will handle them in order.

There is, however, no read type for "the remaining data". For that, we install our own `on_read` callback:

```
# and finally handle any remaining data as body
$handle->on_read (sub {
   $body .= $_[0]->rbuf;
   $_[0]->rbuf = "";
});
```

This callback is invoked every time data arrives and the read queue is empty – which in this example will only be the case when both response and header have been read. The `on_read` callback could actually have been specified when constructing the object, but doing it this way preserves logical ordering.

The read callback adds the current read buffer to its `$body` variable and, most importantly, *empties* the buffer by assigning the empty string to it.

Given these instructions, AnyEvent::Handle will handle incoming data – if all goes well, the callback will be invoked with the response data; if not, it will get an error.

In general, you can implement pipelining (a semi-advanced feature of many protocols) very easily with AnyEvent::Handle: If you have a protocol with a request/response structure, your request methods/functions will all look like this (simplified):

```
sub request {

   # send the request to the server
   $handle->push_write (...);

   # push some response handlers
   $handle->push_read (...);
}
```

This means you can queue as many requests as you want, and while AnyEvent::Handle goes through its read queue to handle the response data, the other side can work on the next request – queueing the request just appends some data to the write queue and installs a handler to be called later.

You might ask yourself how to handle decisions you can only make *after* you have received some data (such as handling a short error response or a long and differently-formatted response). The answer to this problem is `unshift_read`, which we will introduce together with an example in the coming sections.

Using `http_get`

Finally, here is how you would use `http_get`:

```
http_get "www.google.com", "/", sub {
   my ($response, $header, $body) = @_;

   print
      $response, "\n",
      $body;
};
```

And of course, you can run as many of these requests in parallel as you want (and your memory supports).

HTTPS

Now, as promised, let's implement the same thing for HTTPS, or more correctly, let's change our `http_get` function into a function that speaks HTTPS instead.

HTTPS is a standard TLS connection (**T**ransport **L**ayer **S**ecurity is the official name for what most people refer to as `SSL`) that contains standard HTTP protocol exchanges. The only other difference to HTTP is that by default it uses port `443` instead of port `80`.

To implement these two differences we need two tiny changes, first, in the `connect` parameter, we replace `http` by `https` to connect to the https port:

```
      connect  => [$host => 'https'],
```

The other change deals with TLS, which is something AnyEvent::Handle does for us if the Net::SSLeay module is available. To enable TLS with AnyEvent::Handle, we pass an additional `tls` parameter to the call to `AnyEvent::Handle::new`:

```
      tls      => "connect",
```

Specifying `tls` enables TLS, and the argument specifies whether AnyEvent::Handle is the server side (“accept”) or the client side (“connect”) for the TLS connection, as unlike TCP, there is a clear server/client relationship in TLS.

That’s all.

Of course, all this should be handled transparently by `http_get` after parsing the URL. If you need this, see the part about exercising your inspiration earlier in this document. You could also use the AnyEvent::HTTP module from CPAN, which implements all this and works around a lot of quirks for you too.

The read queue – revisited

HTTP always uses the same structure in its responses, but many protocols require parsing responses differently depending on the response itself.

For example, in SMTP, you normally get a single response line:

```
220 mail.example.net Neverusesendmail 8.8.8 <mailme@example.net>
```

But SMTP also supports multi-line responses:

```
220-mail.example.net Neverusesendmail 8.8.8 <mailme@example.net>
220-hey guys
220 my response is longer than yours
```

To handle this, we need `unshift_read`. As the name (we hope) implies, `unshift_read` will not append your read request to the end of the read queue, but will prepend it to the queue instead.

This is useful in the situation above: Just push your response-line read request when sending the SMTP command, and when handling it, you look at the line to see if more is to come, and `unshift_read` another reader callback if required, like this:

```
my $response; # response lines end up in here

my $read_response; $read_response = sub {
   my ($handle, $line) = @_;

   $response .= "$line\n";

   # check for continuation lines ("-" as 4th character")
   if ($line =~ /^...-/) {
      # if yes, then unshift another line read
      $handle->unshift_read (line => $read_response);

   } else {
      # otherwise we are done

      # free callback
      undef $read_response;

      print "we are don reading: $response\n";
   }
};
```

```
$handle->push_read (line => $read_response);
```

This recipe can be used for all similar parsing problems, for example in NNTP, the response code to some commands indicates that more data will be sent:

```
$handle->push_write ("article 42");

# read response line
$handle->push_read (line => sub {
   my ($handle, $status) = @_;

   # article data following?
   if ($status =~ /^2/) {
      # yes, read article body

      $handle->unshift_read (line => "\012.\015\012", sub {
         my ($handle, $body) = @_;

         $finish->($status, $body);
      });

   } else {
      # some error occurred, no article data

      $finish->($status);
   }
}
```

Your own read queue handler

Sometimes your protocol doesn't play nice, and uses lines or chunks of data not formatted in a way handled out of the box by AnyEvent::Handle. In this case you have to implement your own read parser.

To make up a contorted example, imagine you are looking for an even number of characters followed by a colon (":"). Also imagine that AnyEvent::Handle has no `regex` read type which could be used, so you'd have to do it manually.

To implement a read handler for this, you would `push_read` (or `unshift_read`) a single code reference.

This code reference will then be called each time there is (new) data available in the read buffer, and is expected to either successfully eat/consume some of that data (and return true) or to return false to indicate that it wants to be called again.

If the code reference returns true, then it will be removed from the read queue (because it has parsed/consumed whatever it was supposed to consume), otherwise it stays in the front of it.

The example above could be coded like this:

```
$handle->push_read (sub {
   my ($handle) = @_;

   # check for even number of characters + ":"
   # and remove the data if a match is found.
   # if not, return false (actually nothing)

   $handle->{rbuf} =~ s/^ ( (?:..)* ) ://x
      or return;

   # we got some data in $1, pass it to whoever wants it
```

```
      $finish->($1);

      # and return true to indicate we are done
      1
   });
```

Debugging aids

Now that you have seen how to use AnyEvent, here's what to use when you don't use it correctly, or simply hit a bug somewhere and want to debug it:

Enable strict argument checking during development

AnyEvent does not, by default, do any argument checking. This can lead to strange and unexpected results especially if you are just trying to find your way with AnyEvent.

AnyEvent supports a special "strict" mode – off by default – which does very strict argument checking, at the expense of slowing down your program. During development, however, this mode is very useful because it quickly catches the msot common errors.

You can enable this strict mode either by having an environment variable `AE_STRICT` with a true value in your environment:

```
AE_STRICT=1 perl myprog
```

Or you can write `use AnyEvent::Strict` in your program, which has the same effect (do not do this in production, however).

Increase verbosity, configure logging

AnyEvent, by default, only logs critical messages. If something doesn't work, maybe there was a warning about it that you didn't see because it was suppressed.

So during development it is recommended to push up the logging level to at least warn level (`5`):

```
AE_VERBOSE=5 perl myprog
```

Other levels that might be helpful are debug (`8`) or even trace (`9`).

AnyEvent's logging is quite versatile – the AnyEvent::Log manpage has all the details.

Watcher wrapping, tracing, the shell

For even more debugging, you can enable watcher wrapping:

```
AE_DEBUG_WRAP=2 perl myprog
```

This will have the effect of wrapping every watcher into a special object that stores a backtrace of when it was created, stores a backtrace when an exception occurs during watcher execution, and stores a lot of other information. If that slows down your program too much, then `AE_DEBUG_WRAP=1` avoids the costly backtraces.

Here is an example of what of information is stored:

```
59148536 DC::DB:472(Server::run)>io>DC::DB::Server::fh_read
type:    io watcher
args:    poll r fh GLOB(0x35283f0)
created: 2011-09-01 23:13:46.597336 +0200 (1314911626.59734)
file:    ./blib/lib/Deliantra/Client/private/DC/DB.pm
line:    472
subname: DC::DB::Server::run
context:
tracing: enabled
cb:      CODE(0x2d1fb98) (DC::DB::Server::fh_read)
invoked: 0 times
created
(eval 25) line 6        AnyEvent::Debug::Wrap::__ANON__('AnyEvent','fh',GL
```

```
DC::DB line 472         AE::io(GLOB(0x35283f0),'0',CODE(0x2d1fb98)=DC::DB::
bin/deliantra line 2776 DC::DB::Server::run()
bin/deliantra line 2941 main::main()
```

There are many ways to get at this data – see the AnyEvent::Debug and AnyEvent::Log manpages for more details.

The most interesting and interactive way is to create a debug shell, for example by setting `AE_DEBUG_SHELL`:

```
AE_DEBUG_WRAP=2 AE_DEBUG_SHELL=$HOME/myshell ./myprog

# while myprog is running:
socat readline $HOME/myshell
```

Note that anybody who can access *$HOME/myshell* can make your program do anything he or she wants, so if you are not the only user on your machine, better put it into a secure location (*$HOME* might not be secure enough).

If you don't have `socat` (a shame!) and care even less about security, you can also use TCP and `telnet`:

```
AE_DEBUG_WRAP=2 AE_DEBUG_SHELL=127.0.0.1:1234 ./myprog

telnet 127.0.0.1 1234
```

The debug shell can enable and disable tracing of watcher invocations, can display the trace output, give you a list of watchers and lets you investigate watchers in detail.

This concludes our little tutorial.

Where to go from here?

This introduction should have explained the key concepts of AnyEvent – event watchers and condition variables, AnyEvent::Socket – basic networking utilities, and AnyEvent::Handle – a nice wrapper around sockets.

You could either start coding stuff right away, look at those manual pages for the gory details, or roam CPAN for other AnyEvent modules (such as AnyEvent::IRC or AnyEvent::HTTP) to see more code examples (or simply to use them).

If you need a protocol that doesn't have an implementation using AnyEvent, remember that you can mix AnyEvent with one other event framework, such as POE, so you can always use AnyEvent for your own tasks plus modules of one other event framework to fill any gaps.

And last not least, you could also look at Coro, especially Coro::AnyEvent, to see how you can turn event-based programming from callback style back to the usual imperative style (also called ''inversion of control'' – AnyEvent calls *you*, but Coro lets *you* call AnyEvent).

Authors

Robin Redeker <elmex at ta-sa.org>, Marc Lehmann <schmorp@schmorp.de>.