Practical Perl Tools

Parallel Asynchronicity, Part 2

DAVID N. BLANK-EDELMAN

David Blank-Edelman is the Technical Evangelist at Apcera (the comments/views here are David's alone and do not represent Apcera/Ericsson) . He has spent close to thirty years in the systems administration/DevOps/SRE field in large multiplatform environments including Brandeis University, Cambridge Technology Group, MIT Media Laboratory, and Northeastern University. He is the author of the O'Reilly Otter book *Automating System Administration with Perl* and is a frequent invited speaker/organizer for conferences in the field. David is honored to serve on the USENIX Board of Directors. He prefers to pronounce Evangelist with a hard 'g'.
dnblankedelman@gmail.com

Welcome back for Part 2 of this mini-series on modern methods for doing multiple things at once in Perl. In the first part (which I highly recommend that you read first before reading this sequel), we talked about some of the simpler methods for multi-tasking like the use of fork() and Parallel::ForkManager. We had just begun to explore using the Coro module as the basis for using threads in Perl (since the native stuff is so ucky) when we ran out of time. Let's pick up the story roughly where we left it.

Coro

As a quick reminder, Coro offers an implementation of coroutines that we are going to use as just a pleasant implementation of cooperative threading (see the previous column for more picayune details around the definition of a coroutine). By cooperative, we mean that each thread gets queued up to run, but can only do so after another thread has explicitly signaled that it is ready to cede its time in the interpreter. Thus, you get code that looks a little like this:

```
use Coro;

async { print "1\n"; cede;};
async { print "2\n"; cede;};
async { print "3\n"; cede;};
cede;
```

The script runs the main thread, which queues up three different threads and then cedes control of the interpreter to the first queued thread. It cedes control, so the second thread runs and so on. In this example, we don't technically need to write "cede;" at the end of each definition (since each queued thread will cede control simply by exiting), but it is a good habit to get into. The one place we definitely do need to explicitly write "cede;" is at the end of the script. If we didn't cede control at the end of the script, nothing would be printed because the main thread would have exited without realizing it should cede control to anything else.

We can do some more interesting things with this model, but before we do, it would probably be useful to understand how one goes about debugging a Coro-based program. When debugging a program like this, it would be a supremely handy thing to have information about the current state of the program that could tell us just what thread is running and what threads are queued up to run.

Coro ships with a debugger module that does all of this and more. There are two ways to make use of it: a non-interactive way and an interactive way. The interactive way works when used with an event loop-based Coro program like those you might be able to write after reading the last section of the column. But since we are not there yet, let's look at how to use the non-interactive method. We add Coro::Debug to the module loads and then insert a line that runs a debugger command. Let's modify the dead simple code example from above like so:

```
use Coro;
use Coro::Debug;

async { $Coro::current->{desc} = 'numero uno';
      print "1\n";
      cede;
};
async { $Coro::current->{desc} = 'numero dos';
      print "2\n";
      cede;
};
async { $Coro::current->{desc} = 'numero tres';
      print "3\n";
      Coro::Debug::command 'ps';
      cede;
};
cede;
```

We've made two changes. First, we've added lines to each async definition to give each one a description. You'll see how this comes in handy in just a moment. Second, in the third definition we've inserted a debugger command. When we run this script, it now prints something like:

```
1
2

PID              SC  RSS   USES  Description      Where
140252207746400  RC   21k    0   [main::]         [t:20]
140252207836704  N-   216    0   [coro manager]   -
140252207836680  N-   216    0   [unblock_sub     -
                                 scheduler]
140252207540736  R-  2060    1   numero uno       [t:9]
140252208378256  N-   216    0   [AnyEvent idle   -
                                 process]
140252208229104  RC  2600    1   numero dos       [t:13]
140533218598712  UC  2600    1   numero tres      [t:17]  3
```

This output shows us the status of all of the threads. Let me cherry-pick the key parts of this output to describe.

The first line is the main thread. It shows that it is [R]eady to run (the first letter of the SC column), has been scheduled 0 times (USES column) because the main thread doesn't need to be scheduled explicitly, and that it is currently running line 20 of the script (the file name is "t"). If we skip the threads that are part of Coro, we come to the first one we defined ("numero uno"—now you see why setting a description is useful). It too is Ready to run (currently at line 9 in the program). "numero dos" is in a similar state. The final thread we defined is shown as r[U]nning ("R" was taken by Ready). All of our defined threads are shown with a 1 in the USES column because they all have been queued to run once.

More Advanced Coro

In the puny code samples we've seen so far, each of the threads we've scheduled has been totally independent. Each printed a number, a process that didn't require any coordination (beyond making sure to be good neighbors by ceding to each other). But this isn't the most common of situations. Many (most?) times threads in a multi-threaded program are all trying to work towards the same goal by taking on a portion of the work. In those cases, threads have to work together collectively to make sure they aren't stepping on each other's toes. To do so they need a way to signal each other and maybe even pass on data in the process.

Anyone who has done other multi-threaded programming knows I'm headed towards talking about semaphores because that's the classic mechanism for intra-thread signaling. A semaphore is a shared resource (feel free to think of it as a magic variable) that the threads can read or attempt to change before they want to take an action. If a thread's attempt doesn't succeed (because another thread got there first), it can block and wait for the semaphore to become ready. This seems a little abstract, so let me show you some code from the Coro doc [1].

```
use Coro;

 my $sem = new Coro::Semaphore 0; # a locked semaphore

 async {
   print "unlocking semaphore\n";
   $sem->up;
 };

 print "trying to lock semaphore\n";
 $sem->down;
 print "we got it!\n";
```

In this case we are seeing a "counting" semaphore (where the semaphore has a value that can be incremented and decremented) being used as a binary semaphore (is it "locked" or "unlocked").

To follow the flow of the program, the main thread defines a semaphore with a value of 0, queues a separate thread (async{}), prints a message, and then attempts to decrement the semaphore with a call to down(). Since the semaphore is already at 0, the down() call blocks. In Coro, that blocking action cedes, and so the first queued thread gets a chance to run. When it runs, it increments the value of the semaphore and exits. Now that the semaphore is no longer 0, the down() call succeeds and the main thread continues to its end. This is a very basic semaphore mechanism—Coro offers a number of different variations on it so I recommend you look at the documentation.

Semaphores are a simple and effective way to keep threads from getting in each other's way, but what if they actively want to collaborate? That would entail being able to share information.

There are lots of ways threads could pass information around between them, but one built-in way Coro offers is through "channels." Channels (in Coro) are described as message queues. Any thread can add data to the queue, and any (other) thread can consume that data.

The syntax and method for using channels is as straightforward as you might hope. You create a new channel:

```
my $channel = new Coro::Channel;
```

write to it (from any thread):

```
$channel->put ('somepieceofdata');
```

and read from it (presumably from a different thread):

```
my $data = $channel->get;
```

If there is nothing in the channel, that thread will block and cede its time (just like a semaphore attempting to down() if the semaphore is already 0) until data does become available. Easy peasy.

Event-Based Programming

Let's move on to the final paradigm of this series. Event-based programming is yet another way to construct a system where a program can behave as if it is doing several things at once. There are a number of flavors of event-based systems, so let me give a broad generalization of a description that covers what we're about to do.

With the event-based programming style we're about to encounter, the basic idea is to specify events in the program's life that we care about and the code that should run when those events take place. These events could be external to the program (someone clicked on a button in a GUI) or events internal to it (when a piece of the program finishes). It is this latter case that interests us most at the moment because it means we can launch a whole bunch of actions—for example, a ton of DNS requests—and have them run at the same time.

Unlike your usual program that states "do this, then do this, then do this" (which means that thing #3 doesn't happen until #1 and #2 have completed), event-based programming lets you write code that says "do all the things, let me know when any of them finish, and I'll handle them at that point." Most of the time this is described in terms of registering interest in certain events and then starting an event loop that continuously checks if any of the events have come to pass. If it finds this has happened, the code associated with that event (a callback) is executed and then the loop continues.

There are a whole slew of Perl modules for writing event-based programs. Some of them are pure Perl; the more performant ones wrap external event libraries like libevent and libev. For this final section of the column, let's use all of them. Well, maybe most of them. But let's use them at the same time.

More precisely, let's use a module that calls itself "the DBI of event-loop programming." DBI, for those new to Perl, is a standard way to program database-related tasks in Perl that lets the programmer write database code that isn't tied to a specific database. AnyEvent aims to do this for event loops. It provides a uniform way to write code that is event-loop independent. The module will attempt to probe your system for the presence of a relatively long list of other event-based modules (including the performant ones). If it finds one, it will use it (without your having to know the specifics for the one it finds). If it doesn't find one, it will use a Perl-based "backend" that will function fine even without any of those modules being present. AnyEvent has proven quite popular in the community and so now a whole bunch of AnyEvent::Something modules are available for lots of tasks you might commonly want to do in an event-based/high-performance fashion.

Because event-based programming can get hairy quickly, we're only going to skim the top of AnyEvent to discuss the major ideas and then show one example of one of the task-specific AnyEvent::* modules. One other quick note before we move forward: AnyEvent comes with two different interfaces, a method-based one (AnyEvent) and a function-based one (AE). For example, you can write:

```
AnyEvent->timer (after    => $seconds,
                 interval => $intseconds,
                 cb       => ...);
```

or

```
AE::timer $seconds, $intseconds, sub { ... };
```

The function-based one is more terse but is actually 5–6x faster with some backends. For this column, I'm going to use the method interface because I think it is easier for people not familiar with AnyEvent to read. When you write your own code and become comfortable with the arguments being passed to the methods, I encourage you to consider using AE instead so you can gain the performance increases.

The first concept central to any AnyEvent code is the "watcher." AnyEvent provides a set of different kinds of watchers including:

- I/O—when a file handle is ready to be read/written
- time—when a certain amount of time has elapsed
- signal—when we have received a certain signal
- child—when a child process changed state (completed)
- idle—when nothing else is happening

Let's look at a trivial AnyEvent code sample. It uses a time watcher because people can intuitively understand the idea of time events taking place (e.g., "Tell me when ten seconds have elapsed" or "Every two seconds, do the following…"). Here's a sample that uses two time watchers:

```
use AnyEvent;

my $enough   = 15;
my $yammer   = 0;

my $c = AnyEvent->condvar;

my $w;
$w = AnyEvent->timer(
    after      => 2,
    interval   => 2,
    cb         => sub {
        print "Every 2 (" . localtime( AnyEvent->now ) . ")\n";
        $yammer++;
        $c->send if ( $yammer == $enough );
    }
);

my $w2;
$w2 = AnyEvent->timer(
     after     => 5,
     interval  => 5,
     cb        => sub {
        print "Every 5 (" . localtime( AnyEvent->now ) . ")\n";
        $yammer++;
        $c->send if ( $yammer == $enough );
   }
);

print localtime( AnyEvent->now ) . "\n";
$c->recv;
print localtime( AnyEvent->now ) . "\n";
```

After loading AnyEvent, we specify that we only want 15 lines of output and define a variable that will be used to track the number of lines printed. Then we define a condition variable (more on this in a moment because it is fairly important).

Following this are the actual watchers. For each watcher, we say when we want AnyEvent to notice the time. For the first one, we want to notice when two seconds have gone by and then every time two seconds goes by after that. The second watcher is the same except it is paying attention to events every five seconds. When either of the events takes place, they run a tiny callback subroutine that prints the time, increments the output counter, and then decides whether to signal that it is okay to end the event loop (using that mysterious condition variable).

One other small Perl note. You might notice that we did something a little more verbosely than necessary, namely, defining a variable and using it as two different lines (which we almost always do on the same line):

```
my $w;
$w = AnyEvent->timer(
```

The reason we do this is a little subtle and not apparent in this sample itself. Each watcher can have a callback subroutine that gets defined as part of defining the watcher (we do this above). If a watcher wants to disable itself during the program's run, let's say it decides it has done its duty and wants to shut itself off, it does so from within the callback. The way it does so is to "undef" itself. So if the first watcher above wanted to disable itself at any point, in the callback subroutine it would state "undef $w;".

The tricky thing here is that Perl doesn't let you reference a variable in the same statement as the one where it gets defined. We can't do the equivalent of this:

```
my $var = sub { undef $var };
```

hence we have to define the variable that is going to represent the watcher and then create the watcher in two separate steps. You'll see this multiple-statement definition being used all over AnyEvent-based code.

The output of our sample code looks like this:

```
Mon Jun  1 10:37:34 2015
Every 2 (Mon Jun  1 10:37:36 2015)
Every 2 (Mon Jun  1 10:37:38 2015)
Every 5 (Mon Jun  1 10:37:39 2015)
Every 2 (Mon Jun  1 10:37:40 2015)
Every 2 (Mon Jun  1 10:37:42 2015)
Every 5 (Mon Jun  1 10:37:44 2015)
Every 2 (Mon Jun  1 10:37:44 2015)
Every 2 (Mon Jun  1 10:37:46 2015)
Every 2 (Mon Jun  1 10:37:48 2015)
Every 5 (Mon Jun  1 10:37:49 2015)
Every 2 (Mon Jun  1 10:37:50 2015)
Every 2 (Mon Jun  1 10:37:52 2015)
Every 5 (Mon Jun  1 10:37:54 2015)
Every 2 (Mon Jun  1 10:37:54 2015)
Every 2 (Mon Jun  1 10:37:56 2015)
Mon Jun  1 10:37:56 2015
```

So let's talk about condition variables (condvar) because they are one of the most important and the most confounding of AnyEvent concepts. One way to wrap your head around condvar is to harken back to the semaphores and channels we dealt with earlier in the column. Condvars are a way for different parts of the program to communicate with each other through a magic variable. This variable starts off as "false" and only becomes true when another part of the program sends a signal for it to change. In the interim, anything waiting for that signal will block (and here's an important part) while the rest of the event loop continues on around it. In the code we just saw, after defining the condvar ($c) and the watchers we say:

```
$c->recv;
```

which says, "Wait around for the condvar to become true during the event loop before continuing." This very act of waiting for something to happen in the event loop actually instructs AnyEvent to run the event loop.

Both of the watchers we defined check during the event loop if we've produced the right number of output lines in their callback subroutine. If either one determines this condition has been reached, they will send() on the condvar, and the program will stop waiting at the recv(). Since we are no longer waiting for event loop actions to take place, the loop shuts down and the program proceeds to its final print statement.

As you can probably guess, there's a bunch more functionality available from AnyEvent. For example, condvars can be used in a transactional way using begin() and end() calls so that the program can say, "Run an unspecified number of things at once, but only continue once all of them have completed." Rather than dive into more of these features, I want to show one small code example that makes use of one of the other AnyEvent-based modules in the ecosystem. This module we're about to see actually ships with AnyEvent itself.

Inspired by an example in Josh Barratt's excellent presentation on AnyEvent [2], here's some code that uses AnyEvent::DNS to check whether a domain exists in each of the current top-level domains. This version is a little spiffier than Barratt's because it pulls down the current list of all possible TLDs from IANA and checks against that. We'll talk about some of the pieces of the code after you've had a chance to see it:

```
use AnyEvent;
use AnyEvent::DNS;
use HTTP::Tiny;

# receive name to check from command line
my $name = shift;

my $domainslist =
  'http://data.iana.org/TLD/tlds-alpha-by-domain.txt';

my $domainlist = HTTP::Tiny->new->get($domainslist)->{content};

# ignore the comment and the test TLDs
my @domains = grep ( !/^(\#|XN--)/,
                     split( "\n", $domainlist ) );

my $c = AnyEvent->condvar;

my %domainresults;
for my $domain (@domains) {
    $c->begin;
    AnyEvent::DNS::a "$name.$domain", sub {
        $domainresults{$domain} = shift || "did not resolve";
        $c->end;
        }
}
my $start = AnyEvent->now;
$c->wait;
print "$#domains domains looked up in " .
   (scalar AnyEvent->now - $start) . " seconds.\n";
```

The first part of the code pulls down the IANA list. We then begin to iterate over each top-level domain, creating events that perform the lookups for us. When we do, we bracket each event with a condvar-based begin()/end() pair. This is the "transaction-like" use we mentioned earlier. The initial begin() records that we've started something, the end() indicates that we've finished something. We set the event loop in motion with a wait() call that basically says, "Run the event loop until all of the begin()s have had end()s."

Now, you may be as curious as I was to see just how much faster an AnyEvent version would be than one which worked its way through all of the TLDs, one TLD at a time. To test this, I gutted the AnyEvent watcher part in the middle and instead wrote the following:

```
use Net::DNS;
       ...
    my $reply = $res->search("$name.$domain");

    $domainresults{$domain} = "did not resolve";
    if ($reply) {
        foreach my $rr ( $reply->answer ) {
            next unless $rr->type eq "A";
            $domainresults{$domain} = $rr->address;
        }
    }
```

The version above yielded the following:

```
861 domains looked up in 717 seconds.
```

The AnyEvent version I showed first?

```
861 domains looked up in 54 seconds.
```

So, yes, quite a substantial speedup. I leave it as an exercise to the reader to write Parallel::ForkManager and Coro versions of the same program to see how they stack up.

We've come to the end of this column, but before I leave let me just mention that Coro has special support for AnyEvent that lets you use threads and an event-loop seamlessly. See the doc for Coro::AnyEvent for more information. And with that, take care and I'll see you next time.

References
[1] Coro documentation: http://search.cpan.org/perldoc?Coro/Intro.pod.

[2] Josh Barratt's AnyEvent presentation: https://vimeo.com/17163462.

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

The USENIX Store Is Open for Business!

Want to buy a subscription to *;login:*, the latest short topics book, a USENIX or conference shirt, or the box set from last year's workshop? Now you can, via the **USENIX Store!**

Head over to www.usenix.org/store and check out the collection of t-shirts, video box sets, *;login:* magazines, short topics books, and other USENIX and LISA gear. USENIX and LISA SIG members save, so make sure your membership is up to date.

www.usenix.org/store