Introduction to SpeedyCGI

Sam Horrocks

Presented at YAPC North America
14 June 2001

Overview

This paper will introduce SpeedyCGI, a persistent perl interpreter, and will cover the following topics:

- What is SpeedyCGI?
- Why was SpeedyCGI created?
- How do I use SpeedyCGI?
- Why would I want to use SpeedyCGI instead of normal perl?
- How does SpeedyCGI compare to other persistent perl interpreters like mod_perl and fastcgi?
- How does SpeedyCGI work?
- Future directions for SpeedyCGI

What is SpeedyCGI?

SpeedyCGI is a persistent perl interpreter. In traditional perl when you run a perl script, a new process is created, your perl script is compiled and executed, and then the perl process exits. If the same script is run again, all these steps are repeated.

SpeedyCGI behaves a little differently. Just as in regular perl, the first time a perl script is run, a new process is created and the script is compiled and executed. However, with SpeedyCGI, at this point instead of exiting, the perl process is retained. If the same script is run again, then the perl process can execute it right away, without re-reading and re-compiling the script.

Why was SpeedyCGI created?

SpeedyCGI was created because a solution was needed that would:

- make perl CGI apps run much faster
- work completely outside the web server
- not require a lot of administration and tuning
- also speed up regular perl code, not just web-based apps
- be freely available
- allow code to be written that looked like a normal CGI app, and could be run if only regular perl was available

There wasn’t anything at the time that would meet all these requirements, so SpeedyCGI was created.

How do I use SpeedyCGI?

The simplest way is to change the `#!/usr/bin/perl` line at the top of your script to `#!/usr/bin/speedy`. For higher performance, you can also run SpeedyCGI via an Apache module (mod_speedycgi). Unfortunately not all code works correctly when run persistently. So in addition you may have to clean up your code to make it work. Using `strict` and the `-w` switch will solve a lot of these problems.

Why would I use SpeedyCGI instead of normal perl?

Performance. Under SpeedyCGI, if you run the same perl code over and over the script doesn't have to be compiled each time it is run. This means less CPU time is required for each run.

In addition, once your code is running persistently, it's possible to speed up execution even further by caching data or objects in global variables. In SpeedyCGI, global variables retain their values between runs of your script. You can take advantage of this fact to cache things like database handles or other resources instead of having to re-initialize them each time the script is run.

For example, if you have a subroutine "get_db_handle" that returns a database handle, the following code will cache this handle in a persistent global variable:

```
use vars '$dbh';
$dbh ||= &get_db_handle;
```

The first time this code is run, `$dbh` will be undefined so the `||` operator will cause `get_db_handle` to be called. On subsequent runs of the script, `$dbh` will already be defined (it will already hold the database handle) and `get_db_handle` will not be called.

Comparison to other persistent perl interpreters

There are other persistent perl environments around. Below is a comparison between SpeedyCGI and two popular persistent perl interpreters - mod_perl and FastCGI.

Comparison to both mod_perl and FastCGI

- **SpeedyCGI Advantages**
  - SpeedyCGI can be used for general purpose perl programming, not just web-based scripts.
  - SpeedyCGI provides real files for STDIN, STDOUT and STDERR, including real unix file descriptors, increasing compatibility with existing code.
  - Speedy always tries to use the fewest number of perl interpreters possible for the given load by reusing the same interpreters over and over. This results in fewer interpreters being used under a heavy load.
- **SpeedyCGI Disadvantages**
  - No Win32 version yet - Unix only.

mod_perl Comparison

- **SpeedyCGI Advantages**
  - The SpeedyCGI perl interpreter runs outside the web server, so bad perl code can’t affect the web server.
  - Each interpreter can be assigned to run only a single script, or only a certain group of scripts. This means you can keep one group of scripts from affecting another group, or set different policies for different groups of scripts. In mod_perl there is no control over this - each interpreter runs all of the scripts.
  - SpeedyCGI buffers the output from the script. If the buffer is made large enough, then this means that as soon as the perl interpreter is done producing results it can be reused for another request, regardless of how long it takes to send the buffered output to the client.
  - SpeedyCGI can work with any web server, not just Apache
- **SpeedyCGI Disadvantages**
  - mod_perl provides access to web server internals and allows for writing request handlers which are faster than CGI.
  - mod_perl doesn’t copy the output data twice - it goes directly from the program to the client.
  - The perl interpreters in mod_perl can share pre-compiled code because they are forked from a common base interpreter. When this feature is used it can mean a smaller amount of private memory used for each perl interpreter.

FastCGI Comparison

- **SpeedyCGI Advantages**
  - SpeedyCGI doesn’t require that you add an "accept" loop to your code. There are no architectural changes required by SpeedyCGI to the code.
  - SpeedyCGI can run more than one script in each interpreter.
- **SpeedyCGI Disadvantages**
  - FastCGI can use multiple systems to run the interpreters. SpeedyCGI only runs on the local system.
  - FastCGI supports languages other than perl.

How does SpeedyCGI work?

When you run `/usr/bin/speedy`, you are not directly running a perl interpreter. The speedy executable is only a "frontend". The actual perl interpreter is contained in a different executable named "speedy_backend".

When executed, the speedy frontend does the following:

1. Looks for an available backend to run this script.
2. If no backends are available, starts a new one.
3. As soon as a backend is located, connects to it and starts to send over %ENV, @ARGV and the STDIN data.
4. Brings back the STDOUT and STDERR data from the backend and sends it to its output.

The speedy backend does the following:

1. Initializes the perl interpreter and compiles the perl script
2. Waits for a frontend to contact it
3. Once a frontend contacts it, reads in and initializes %ENV and @ARGV
4. Sets up STDIN, STDOUT and STDERR so that they are connected to the frontend via Unix sockets.
5. Executes the perl code.
6. Goes back and waits for another frontend to contact it.

A data file in /tmp is used to keep track of the frontends and backends. Once the two processes find each other, Unix sockets are used for communication.

Future Directions

- Win32 Port
- Better buffering so that we don’t start a perl interpreter until we’ve received most of the STDIN data. This should reduce the number of interpreters needed during http post operations.
- Fork from a common perl interpreter so we can shared compiled perl code like mod_perl does.

More Information

For more information about SpeedyCGI see the SpeedyCGI home page at http://daemoninc.com/SpeedyCGI/