
Passenger architectural overview

Table of Contents

About the involved technologies	1
Typical web applications	1
Ruby on Rails	3
Apache	3
Passenger architecture	3
Overview	3
Spawning and caching of code and applications	4
The spawn server	5
Handling of concurrent requests	7
A. About this document	7

This document describes Passenger's architure in a global way. The purpose of this document is to lower the barrier to entry for new contributors, as well as to explain (some of the) design choices that we have made.

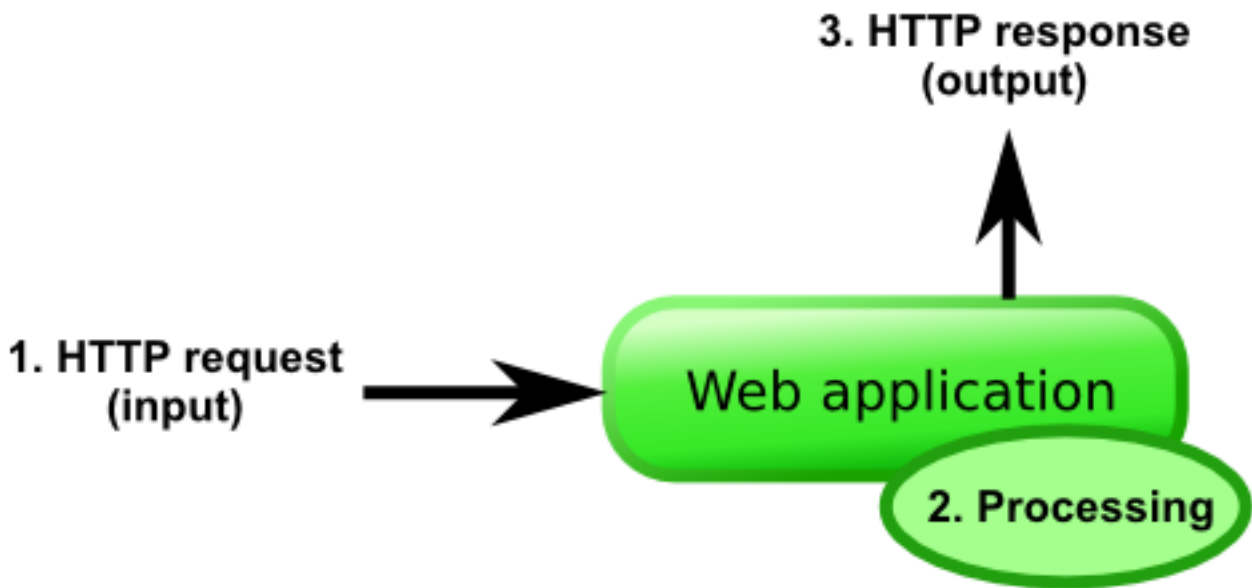
Or it can be a fun read for people who just want to know how Passenger works.

About the involved technologies

Typical web applications

Before we describe Passenger, it is important to understand how typical web applications work, from the point of view of someone who wants to connect the application to a web server.

A typical, isolated, web application accepts an HTTP request from some I/O channel, processes it internally, and outputs an HTTP response, which is sent back to the client. This is done in a loop, until the application is commanded to exit. This does not necessarily mean that the web application speaks HTTP directly: it just means that the web application accepts some kind of representation of an HTTP request.



Few web applications are accessible directly by HTTP clients. Common setups are:

1. The web application is contained in an application server. This application server may or may not be able to contain multiple web applications. The application server is then connected to the web server. The web server dispatches requests to the application server, which in turn dispatches requests to the correct web application, in a format that the web application understands. Conversely, HTTP responses outputted by the web application are sent to the application server, which in turn sends them to the web server, and eventually to the HTTP client.

A typical example of such a setup is a J2EE application, contained in the Tomcat web server, behind the Apache web server.

2. The web application is contained in a web server. In this case, the web server acts like an application server. This is the case for PHP applications, on Apache servers with *mod_php*. Note that this does not necessarily mean that the web application is run inside the same process as the web server: it just means that the web server manages applications.
3. The web application is a web server, and can accept HTTP requests directly. This is the case for the Trac bug tracking system, running in its standalone server. In many setups, such web applications sit behind a different web server, instead of accepting HTTP requests directly. The frontend web server acts like a reverse HTTP proxy.
4. The web application does not speak HTTP directly, but is connected directly to the web server through some communication adapter. CGI, FastCGI and SCGI are good examples of this.

These descriptions are true for virtually all web applications, whether they're based on PHP, Django, J2EE, ASP.NET, Ruby on Rails, or whatever. Note that all of these setups provide the same functionality, i.e. no setup can do something that a different setup can't. The critical reader will notice that all of these setups are identical to the one described in the first diagram, if the combination of web servers, application servers, web applications etc. are considered to be a single entity; a black box if you will.

It should also be noted that these setups do not enforce any particular I/O processing implementation. The web servers, application servers, web applications, etc. could process I/O serially (i.e. one request at a

time), could multiplex I/O with a single thread (e.g. by using `select(2)` or `poll(2)`) or it could process I/O with multiple threads and/or multiple processes.

Of course, there are many variations possible. For example, load balancers could be used. But that is outside the scope of this document.

Ruby on Rails

Every Ruby on Rails application has a *dispatcher*. This dispatcher is responsible for processing HTTP requests. It does not speak HTTP directly. Instead, it accepts data structures that contain the information of an HTTP request. Thus, the dispatcher is particularly interesting to developers who wish to develop software which connects Ruby on Rails to an HTTP processing layer (e.g. a web server).

The Ruby on Rails dispatcher can only process requests serially, i.e. one at a time. It is not possible to process two requests at the same time with threads, because parts of Ruby on Rails are not thread-safe. (In practice, this isn't as big of a problem as some people imagine. This will be elaborated further in Handling of concurrent requests.)

A particularly interesting thing to note, is that a lot of the memory occupied by Ruby on Rails applications is spent on storing the program code (i.e. the abstract syntax tree (AST) [http://en.wikipedia.org/wiki/Abstract_syntax_tree]) in memory. This is observed through the use of the memory statistics function in Ruby Enterprise Edition [<http://www.rubyenterpriseedition.com/>]. Also, a lot of the startup time of a Ruby on Rails application is spent on bootstrapping the Rails framework.

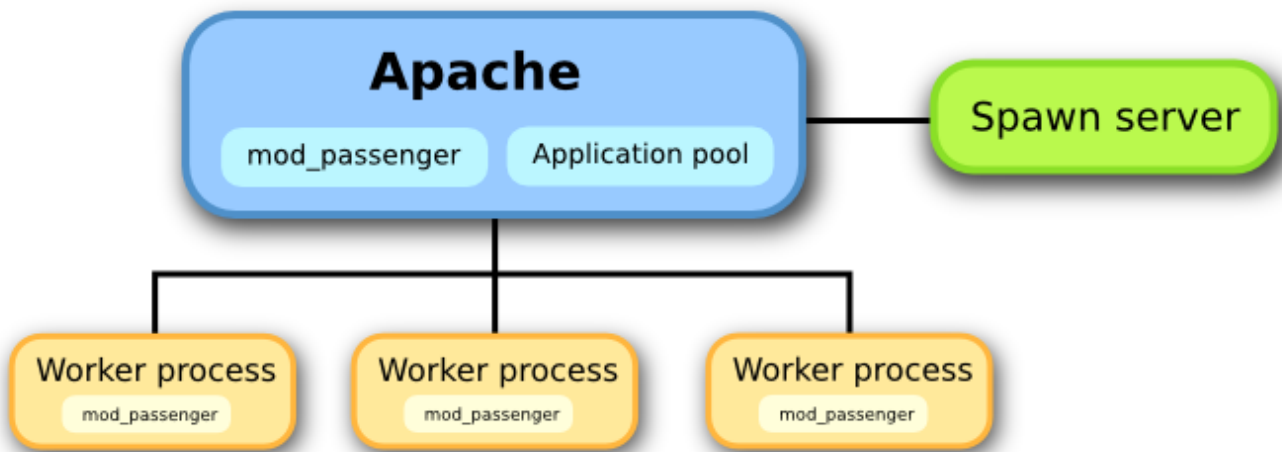
Apache

The Apache web server has a pluggable I/O multiprocessing (the ability to handle more than 1 concurrent HTTP client at the same time) architecture. An Apache module which implements a particular multiprocessing strategy, is called a Multi-Processing Module (MPM). The prefork MPM [<http://httpd.apache.org/docs/2.0/mod/prefork.html>] — which also happens to be the default — appears to be the most popular one. This MPM spawns multiple worker child processes. HTTP requests are first accepted by a so-called control process, and then forwarded to one of the worker processes. The next section contains a diagram which shows the prefork MPM's architecture.

Passenger architecture

Overview

Passenger's architecture is a lot like setup #2 described in Typical web applications. In other words, Passenger extends Apache and allows it to act like an application server. Passenger's architecture — assuming Apache 2 with the prefork MPM is used — is shown in the following diagram:



Passenger consists of an Apache module, *mod_passenger*. This is written in C++, and can be found in the directory *ext/apache2*. The module is active in the Apache control process and in all the Apache worker processes. When an HTTP request comes in, *mod_passenger* will check whether the request should be handled by a Ruby on Rails application. If so, then *mod_passenger* will spawn the corresponding Rails application (if necessary) and forward the request to that application.

It should be noted that the Ruby on Rails application does **not** run in the same address space as Apache. This differentiates Passenger from other application-server-inside-web-server software such as *mod_php*, *mod_perl* and *mod_ruby*. If the Rails application crashes or leak memory, it will have no effect on Apache. In fact, stability is one of our highest goals. Passenger is carefully designed and implemented so that Apache shouldn't crash because of Passenger.

Spawning and caching of code and applications

A very naive implementation of Passenger would spawn a Ruby on Rails application every time an HTTP request is received, just like CGI would. However, spawning Ruby on Rails applications is expensive. It can take 1 or 2 seconds on a modern PC, and possibly much longer on a heavily loaded server. This overhead is particularly unacceptable on shared hosts. A less naive implementation would keep spawned Ruby on Rails application instances alive, similar to how Lighttpd's FastCGI implementation works. However, this still has several problems:

1. The first request to a Rails website will be slow, and subsequent requests will be fast. But the first request to a different Rails website - on the same web server - will still be slow.
2. As we've explained earlier in this article, a lot of memory in a Rails application is spent on storing the AST of the Ruby on Rails framework and the application. Especially on shared hosts and on memory-constrained Virtual Private Servers (VPS), this can be a problem.

Both of these problems are very much solvable, and we've chosen to do just that.

The first problem can be solved by preloading Rails applications, i.e. by running the Rails application before a request is ever made to that website. This is the approach taken by most Rails hosts, for example in the form of a Mongrel cluster which is running all the time. However, this is unacceptable for a shared host: such an application would just sit there and waste memory even if it's not doing anything. Instead, we've chosen to take a different approach, which solves both of the aforementioned problems.

We spawn Rails applications via a *spawn server*. The spawn server caches Ruby on Rails framework code and application code in memory. Spawning a Rails application for the first time will still be slow, but subsequent spawn attempts will be very fast. Furthermore, because the framework code is cached independently from the application code, spawning a different Rails application will also be very fast, as long as that application is using a Rails framework version that has already been cached.

Another implication of the spawn server is that different Ruby on Rails will share memory with each other, thus solving problem #2. This is described in detail in the next section.

But despite the caching of framework code and application code, spawning is still expensive compared to an HTTP request. We want to avoid spawning whenever possible. This is why we've introduced the **application pool**. Spawned application instances are kept alive, and their handles are stored into this pool, allowing each application instance to be reused later. Thus, Passenger has very good average case performance.

The application pool is shared between different worker processes. Because the worker processes cannot share memory with each other, either shared memory must be used to implement the application pool, or a client/server architecture must be implemented. We've chosen the latter because it is easier to implement. The Apache control process acts like a server for the application pool. However, this does not mean that all HTTP request/response data go through the control process. A worker process queries the pool for a connection session with a Rails application. Once this session has been obtained, the worker process will communicate directly with the Rails application.

The application pool is implemented inside *mod_passenger*. One can find detailed documentation about it in the C++ API documentation [cxxapi/index.html], in particular the documentation about the `ApplicationPool`, `StandardApplicationPool` and `ApplicationPoolServer` classes.

Note

At the moment, Passenger does not support Apache with the worker MPM (which uses threads instead of processes). But because the application pool is implemented in a modular way, supporting the worker MPM shouldn't take more than 10 lines of code.

The application pool is responsible for spawning applications, caching spawned applications' handles, and cleaning up applications which have been idle for an extended period of time.

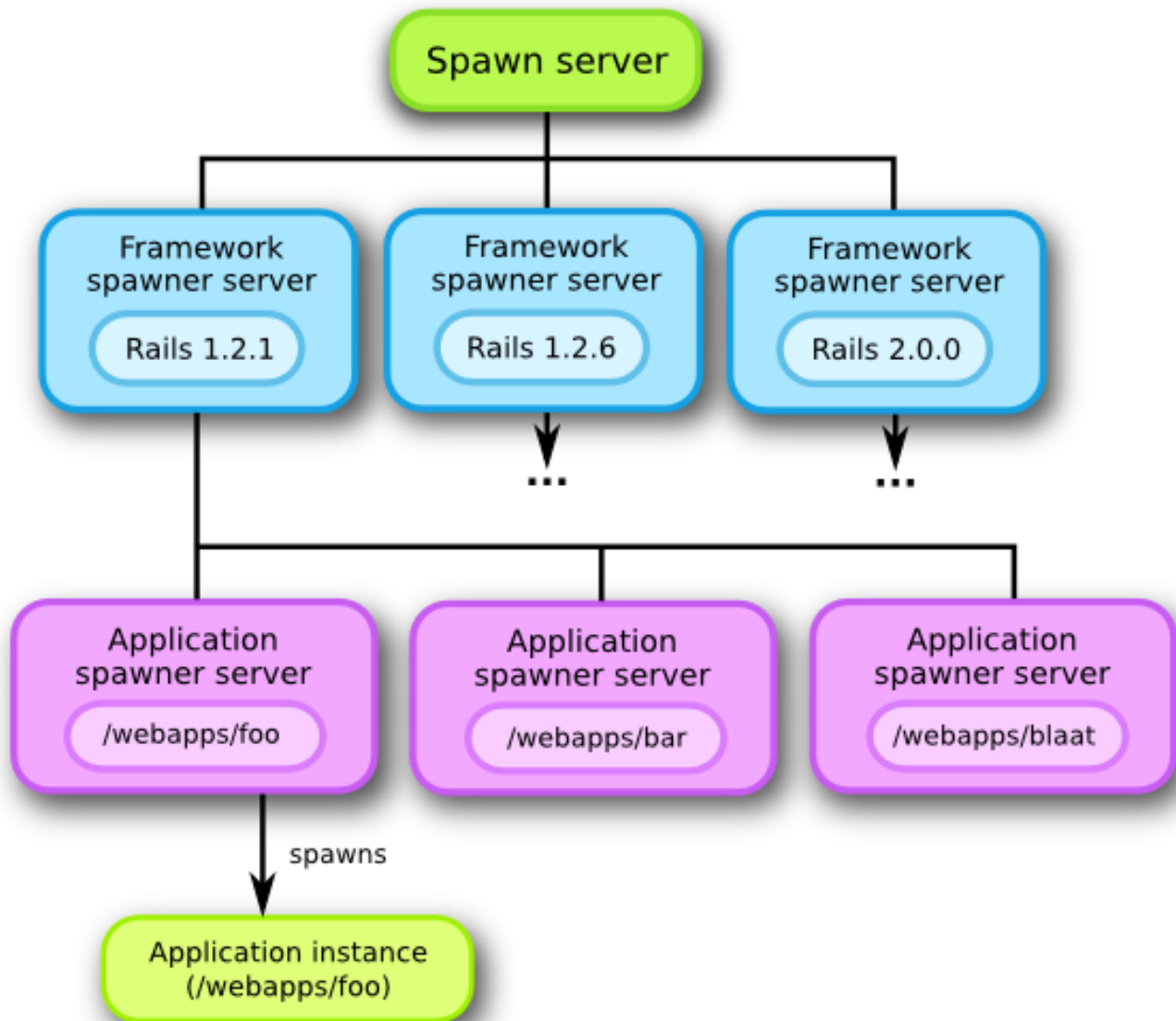
The spawn server

The spawn server is written in Ruby, and its code can be found in the directory *lib/passenger*. Its main executable is *bin/passenger-spawn-server*. The spawn server's RDoc documentation [rdoc/index.html] documents the implementation in detail.

The spawn server consists of 3 logical layers:

1. **The spawn manager.** This is the topmost layer, and acts like a facade for all the underlying layers. Clients who use the spawn server only communicate with this layer.
2. **The framework spawner server.** The spawn manager spawns a framework spawner server for each unique Ruby on Rails framework version. Each framework spawner server caches the code for exactly one Ruby on Rails framework version. A spawn request for an application is forwarded to the framework spawner server that contains the correct Ruby on Rails version for the application.

3. **The application spawner server.** This is to the framework spawner server what the framework spawner server is to the spawn manager. The framework spawner server spawns an application spawner server for each unique Ruby on Rails application (here “application” does not mean a running process, but a set of (source code) files). An application spawner server caches the code for exactly one application.



As you can see, we have two layers of code caching: when the spawn server receives a request to spawn a new application instance, it will forward the request to the correct framework spawner server (and will spawn that framework spawner server if it doesn't already exist), which — in turn — will forward it to the correct application spawner server (which will, again, be created if it doesn't already exist).

Each layer is only responsible for the layer directly below. The spawn manager only knows about framework spawner servers, and a framework spawner server only knows about its application spawner servers. The application spawner server is, however, not responsible for managing spawned application instances. If an application instance is spawned by `mod_passenger`, its information will be sent back to `mod_passenger`, which will be fully responsible for managing the application instance's life time (through the application pool).

Also note that each layer is a separate process. This is required because a single Ruby process can only load a single Ruby on Rails framework and a single application.

Memory sharing

On most modern Unix operating systems, when a child process is created, it will share most of its memory with the parent process. Processes are not supposed to be able to access each others' memory, so the operating system makes a copy of a piece of memory when it is written to by the parent process or the child process. This is called copy-on-write (COW). Detailed background information can be found on Ruby Enterprise Edition's website [<http://www.rubyenterpriseedition.com/>].

The spawn server makes use of this useful fact. Each layer shares its Ruby AST memory with all of its lower layers, as long as the AST nodes in question haven't been written to. This means that all spawned Rails applications will — if possible — share the Ruby on Rails framework's code, as well as its own application code, with each other. This results in a dramatic reduction in memory usage.

Note

Sharing memory only works if Ruby Enterprise Edition [<http://www.rubyenterpriseedition.com/>] is used. This is because the standard Ruby interpreter's garbage collector isn't copy-on-write friendly. Please visit the Ruby Enterprise Edition website for technical details.

Passenger works fine with standard Ruby. You still get to enjoy reduced Rails startup times. You just won't be able to benefit from memory sharing.

Note that Rubinius [<http://rubini.us/>]'s garbage collector is already copy-on-write friendly.

Handling of concurrent requests

As explained earlier, a single Rails application instance can only handle a single request at the same time. This is obviously undesirable. But before we dive into the solution, let us take a look how the “competition” solves this problem. PHP has similar problems: a single PHP script can also process only one HTTP request at a time.

- `mod_php` “solves” this problem by using Apache's MPM. In other words, `mod_php` doesn't do anything by itself at all. A single Apache worker process/thread can only handle 1 PHP request at a time, but Apache spawns multiple worker processes/threads.
- PHP-FastCGI solves the problem by spawning multiple persistent PHP servers. The number of PHP servers is independent from the number of Apache worker processes/threads. This approach is a lot like existing Rails setups, in which a frontend web server proxies requests to a persistent Mongrel cluster.

Passenger cannot use the `mod_php` way because it would force us to spawn a new Rails application for each request, which is — as explained earlier — unacceptably slow. Instead, Passenger uses the PHP-FastCGI approach. We maintain a pool of application instances, and whenever a request is received, we forward the request to one of the application instances in the pool. The size of the pool is configurable, which is useful for administrators of servers that are either heavily loaded or have little memory.

The reader might also be interested in studying the application pool's algorithm, which is non-trivial. The algorithm is documented in detail in `ApplicationPool algorithm.txt` [`ApplicationPool%20algorithm.txt`].

A. About this document

The text of this document is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported License [<http://creativecommons.org/licenses/by-sa/3.0/>].

