Ruby 1.9.x Web Servers Booklet

A SURVEY OF ARCHITECTURAL CHOICES
& ANALYSIS OF RELATIVE PERFORMANCE

MUHAMMAD A. ALI
<oldmoe>
http://oldmoe.blogspot.com
http://github.com/oldmoe
oldmoe@gmail.com

Abstract

This survey looks at different Ruby web servers and attempts to rationalize their architectural choices. It tries to build a background of best practices in building web servers and compares those to what is found in the Ruby servers being surveyed. Finally benchmarking results are presented and compared to initial expectations.

About the Author

Muhammad A. Ali, A.K.A oldmoe, is a software developer from Alexandria, Egypt. He has been writing software professionally for more than nine years now. He authored the NeverBlock, Reactor and MySQLPlus Ruby libraries. He loves to code in Ruby and JavaScript. While not coding he enjoys looking at code written by the likes of _Why the Lucky Stiff and Marc-André Cournoyer.

Disclaimer

This survey is rather limited in its scope as it focuses solely on the Ruby 1.9.x series. It does not attempt to cover Ruby 1.8.x, Ruby EE or JRuby. The version used for the tests presented in this document is 1.9.1.

This survey does not cover all web servers written in Ruby, due to time and resource constraints I limited the tests to the servers I use myself (Thin & Passenger) plus two of the defacto Ruby servers (Webrick & Mongrel). No other servers were tested.

This survey only looks at the performance and scalability (as in handling concurrent load) aspects of the presented web servers. It doest not attempt to cover topics like security, configurability and administrability. Thus any conclusion must be looked upon as limited in scope.

Table of Contents

Abstract....2
About the Author....3
Disclaimer....4
1. Introduction....6
2. Web Server Primer....7
3. Concurrency Models....8
3.1. Introduction....8
3.2. Multi Processing....8
3.3. Multi Threading....10
3.4. Asynchronous Event Loop....14
3.5. Combinations....16
4. Performance Bottlenecks....17
4.1. Introduction....17
4.2. Data Copying....17
4.3. Context Switching....17
4.4. Lock Contention....18
4.5. Memory Management....18
4.6. Blocking Operations....18
4.7. HTTP Parsing....18
4.8. The TCP Stack....19
5. Ruby Web Servers....20
5.1. Introduction....20
5.2. WEBrick....21
5.3. Mongrel....24
5.4. Thin....27
5.5. Passenger....30
6. Benchmarks....33
6.1. Introduction....33
6.2. Benchmarking Methodology....34
6.3. Test Setup....36
6.4. Results....37
6.4.1. Serving Static Files....37
6.4.2. Serving Dynamic Requests (Single Rails Process)....44
6.4.3. Serving Dynamic Requests From A Proxied Cluster (2 Processes)....49
7. Benchmark Analysis....54
7.1. Static File Serving Performance....54
7.1.1. Small File Sizes....54
7.1.2.Large File Sizes....54
7.2. Dynamic Request Serving Performance....58
7.2.1. Quick and Small Responses....58
7.2.1. Heavy Processing but Small Responses....58
7.2.1. Little Processing but Large Responses....58
8. Conclusion....59
References....60

1. Introduction

Ruby was never famous for its capability as a language for writing high performance web servers. The only major implementation for quite some time was Webrick. A sweet little server written entirely in Ruby and not known to break any speed records. In 2006 though a new contender arrived to the scene in the form of Mongrel. A server written by the now famous Zed A. Shaw. Mongrel totally changed the web serving scene in Ruby and raised the bar tremendously. It brought several innovations, most notably the Ragel based HTTP parser which was small, quick and very robust. Mongrel inspired others and more servers were written for Ruby (and in Ruby).

In this survey we will analyze the various offerings in the Ruby web server arena. We will attempt to understand the different architectures and in then we will see how the servers compare when faced with a set of various workloads. We will do the analysis and the benchmarking against both Ruby 1.8.x series and Ruby 1.9.x series (whenever possible).

Although this survey focuses on Ruby web servers it should be of value to anyone researching web server architectures in general. The survey also discusses alternative choices that are not implemented in any of the servers being surveyed.

It should be noted (again) that this is not a comprehensive comparison between the surveyed servers. It focuses on performance and scalability but there are many other considerations like security, configurability and many others. It should be clear that the presented results are of limited scope and hence any conclusion derived from them are limited to that scope as well.

2. Web Server Primer

At its core any web server is simply a never ending loop that attempts to accept connections on a listening socket. Here is a very simple TCP server

```
require 'socket'
server = TCPServer.new("0.0.0.0", 8080)
loop do
  connection = server.accept
  inputline = connection.gets
  ...
  connection.puts outputline
  connection.close
end
```

Code Listing 1

The servers differ in how they construct this loop and how they process incoming connections. The above sample is for a blocking server. Which means that it can only process one request at a time and that other requests will be waiting for the current one to finish. A long running request might make the server unreachable for a while. A group of those will quickly render the server unusable. There are several strategies to overcome this shortcoming. We will discuss those strategies and look at how they are utilized by the different servers

For a server to be called a web (HTTP) server it must speak the HTTP protocol. Hence it needs a way to parse the incoming HTTP requests. Each of the servers presented here attempts to solve this problem in its own way. But we will soon find that most of them rely on some clone of Mongrel's parser. If we modify our first server to include HTTP support it could like this:

```
require 'socket'
server = TCPServer.new("0.0.0.0", 8080)
loop do
  connection = server.accept
  request = HTTP.parse(connection.gets)# an imaginary HTTP parser
  ...
  connection.puts status
  connection.puts headers
  connection.puts body
  connection.close
end
```

Code Listing 2

3. Concurrency Models

3.1. Introduction

Since web servers need to be able to handle multiple clients at once then they must employ some form of concurrency. Different concurrency models bring with them different concerns and implications. We will look at various options available to Ruby web servers. The list is not inclusive as there are other models that are more prominent outside the Ruby sphere.

3.2. Multi Processing

In its simplest form you build a server that performs all its I/O in a blocking manner. Then you instantiate it several times. Each instance can handle one connection at a time. Hence you can handle as many connections as there are processes.

There are usually two approaches here. One is by starting independent processes (like a mongrel cluster, each process listens on its own port) or by dynamically forking processes from each other. The latter enjoys the benefit of being able to adapt the number of processes to load while the former is almost always fixed in size. In its extreme we use a process per connection (like in CGI mode) but the common use is a pool (referred to as cluster sometimes) of processes which are load balanced by some proxy server.

This model has its roots in the UNIX traditions of extensive use of multiprocessing for concurrency. Many UNIX web servers where built around fork a server per connection model. While easy, this model has scalability challenges as the cost of forking many processes prohibits it from being able to handle a relatively large number of concurrent connections.

Here is a sample forking server written in Ruby:

```
require 'socket'
server = TCPServer.new("0.0.0.0", 8080)
loop do
  connection = server.accept
  if fork
    request = HTTP.parse(connection.gets)
    ...
    connection.puts status
    connection.puts headers
    connection.puts body
    connection.close
  end
end
```

Code Listing 3

On the other hand using a set of pre-forked (or pre-instantiated) servers avoids the overhead of forking servers per request but it limits the ability of the server to handle concurrent connections to the count of the servers that were instantiated. The forked servers in that model either inherit the server socket from the master and share the same port or the master accepts all connections and load balances among the workers. In the first case the operating system will be doing the load balancing among the server. Code listings 4 and 5 implement those two models.

```
require 'socket'
server = TCPServer.new("0.0.0.0", 8080)
(server_count-1).times do
  break unless fork
end
# This code will run in all processes
# the kernel will synchronize calls for 'accept'
loop do
  connection = server.accept
  request = HTTP.parse(connection.gets)
  ...
  connection.puts status
  connection.puts headers
  connection.puts body
  connection.close
end
```

Code Listing 4

```
require 'socket'
server = TCPServer.new("0.0.0.0", 8080)
workers = []
worker_count.times do
  # a socket pair that the master uses to send connections to the worker
  master, worker = UnixSocket.pair
  workers << worker
  if !fork
    # this code runs in the worker
    loop do
      # the workers receives a connection from the master
      conn = master.recv_io
      request = HTTP.parse(conn.gets)
      ...
      conn.puts status
      conn.puts headers
      conn.puts body
      conn.close
    end
  end
end
# this is the master loop
loop do
  connection = server.accept
  w = workers.shift     # pick a worker
  w.send_io(connection) # send the connection to that worker
  connection.close      # the master closes the connection (worker has it now)
  workers << w          # worker is returned to the pool
end
```

Code Listing 5

Currently that is the only model that scales to multiple CPU cores in Ruby regardless of whether we are talking about 1.8.x or 1.9.x.

3.3. Multi Threading

Much like Multi Process but this time we spawn multiple threads in the same process. A single process can handle multiple concurrent requests that way. Each thread is handled in a blocking manner but other threads are able to run while it is blocked. Usually a thread is created per connection but pooling is also used to avoid the overhead of thread creation. Ruby 1.9 is totally different from 1.8 in this model. As it utilizes much more expensive to create kernel threads. But on the other hand scheduling those is much faster than Ruby 1.8 threads. Threads are not able to utilize more than one processor core because under 1.8 they are green

threads and under 1.9 there is a giant virtual machine lock that ensures that only one thread runs at a time. That is to enable the use of non thread safe c extensions.

There are several variations of this model. One is to run several threads that implement the full serving logic. Each of those listens on the socket, accepts connections, processes requests and send back responses. Another model is an acceptor thread (usually the main thread) that accepts connections and pass them over to worker threads that handle them. The models are roughly implemented in code listings 6 and 7.

```
require 'socket'
require 'thread'
server = TCPServer.new("0.0.0.0", 8080)
# function body implements the serving logic
def start
  loop do
    connection = server.accept
    request = HTTP.parse(connection.gets)
    ...
    connection.puts status
    connection.puts headers
    connection.puts body
    connection.close
  end
end
(worker_count-1).times do
  # create worker threads and each runs a server
  Thread.new do
    start
  end
end
start # even the main thread can run its own server
```

Code Listing 6

```
require 'socket'
require 'thread'
server = TCPServer.new("0.0.0.0", 8080)
loop do
  connection = server.accept
  # spawn a new thread to handle the connection
  Thread.new do
    request = HTTP.parse(connection.gets)
    ...
    connection.puts status
    connection.puts headers
    connection.puts body
    connection.close
  end
end
```

Code Listing 7

While threads are cheaper than processes they remain relatively expensive resources (specially for Ruby 1.9.x). A server that spawns too many threads will waste considerable time on activities not directly related to web serving like thread creation, locking and context switching. That is why thread pooling is very common in threaded web servers. Code listing 8 presents another strategy that uses a fixed number of threads and a queue.

```
require 'socket'
require 'thread'
@queue = Queue.new
server = TCPServer.new("0.0.0.0", 8080)
# spawn multiple worker threads
worker_count.times do
  Thread.new do
    loop do
      # attempt to retrieve a connection from the queue and handle it
      if conn = @queue.shift
        request = HTTP.parse(conn.gets)
        ...
        conn.puts status
        conn.puts headers
        conn.puts body
        conn.close
      end
    end
  end
end
loop do
  # accept a new connection and put it in the queue
  @queue << server.accept
end
```

Code Listing 8

It is important to mention that getting threads to work correctly is not a trivial problem. Since threads run in the same process they get to share its data. Accessing this data safely from multiple threads is a challenging problem that needs to be handled carefully. Deadlocks and race conditions are the most common pitfalls of threaded applications. And they usually mean that you are either locking more than you need or less than you need. It is a thin wire that you have to walk upon when it comes to writing correct, non-trivial threaded applications.

It should be noted that while threads cannot utilize multiple CPU cores they stand out as the easiest means of performing concurrent disk I/O along with multiple processes. This is because disk I/O does not fit easily with event based architectures as we will see next.

3.4. Asynchronous Event Loop

In this concurrency model all I/O notifications are handled by a central tight loop calling a method like select() or poll() on the set of file descriptors the program is dealing with. Whenever there is activity an object is notified and it operates on the active IO object. Then it delegates I/O notification back to the reactor core and so on.

An example would be registering interest in read activity for the server socket. Whenever select() is called it will tell you whether the socket has data or not. If it has data then the connection is accepted and inserted in the event loop to be processed whenever there is data to read from it. After a connection is done with it should be removed from the event loop to register disinterest in further events.

Such applications are single threaded and in that single thread many I/O operations can occur concurrently. All I/O must be done in a non blocking manner though or the whole server will block and not other operations will be done till this I/O completes.

Also it is worthy to note that this model provides I/O concurrency only. Processing concurrency is not achieved since it is done in a single process that can only use one CPU. If a request uses too much CPU time all other pending requests will have to wait till it finishes.

Also disk I/O will not be concurrent. This can only be done with multi processing or multi threading (only when using Ruby1.9 for the case of multi threading). The issue is that file system calls might block anyway even after you receive a ready notification on them. This makes the use of file descriptors for real files in a select() call almost useless.

While select() and poll() are the most common method calls to poll they are hindered by the fact that their performance is relative to the number of file descriptors they are watching. O(n) to better describe it. While this is OK for a small set of file descriptors it is expensive for servers with many concurrent clients. Modern event loops use platform specific methods like epoll on Linux and kqueue on FreeBSD which get rid of the linear performance profile and provide O(1) implementations that perform the same regardless of the file descriptors count.

An example asynchronous server is presented in code listing 9.

```
# note, this server can only handle data that can be read or written
# in a single shot. A correct implementation is more involved but details
# are removed here for clarity purposes
require 'socket'
server = TCPServer.new("0.0.0.0", 8080)
@write, @write_actions = [], {}
@read, @read_actions = [], {}
# add the server to the read sockets and an action to fire once it is ready
@read << server
@read_actions[server] = proc{|server|
  # accept a new connection and add it to the read list
  @read << conn = server.accept
  @read_actions[conn] = proc{|conn|
    data = conn.read
    ...
    # once a connection is readable then read the data
    # and remove it from the read list
    # then add it to the write list
    @read.delete(conn)
    @read_actions.delete(conn)
    @write << conn
    @write_actions[conn] = proc{|conn|
       # once the connection is writable then write the response
       # close the connection and remove it from the write list
       conn.write(response)
       conn.close
       @write.delete(conn)
       @write_actions.delete(conn)
    }
  }
}
loop do
  # call select to see which sockets are ready
  if res = IO.select(@read,@write,nil,0.01)
    # some sockets have data to be read
    res[0].each{|io|@read_actions[io].call(io)} if res[0]
    # we can write to some sockets
    res[1].each{|io|@write_actions[io].call(io)} if res[1]
  end
end
```

Code Listing 9

It is clear from code listing 9 that writing servers based on an event loop is a much more involved process that the previous models. Nevertheless, this model typically provides the best performance.

3.5. Combinations

A server might opt to use multiple processes with multiple threads each or may be an event loop will delegate connections to a pool of worker threads. The system on which the server will run and the requirements of the application can help decide which is the best strategy for handling I/O. Code listing 10 represents a hybrid multi processing and multi threading server.

```
require 'socket'
require 'thread'
server = TCPServer.new("0.0.0.0", 8080)
(server_count-1).times do
  break unless fork
end
# This code will run in all processes
# the kernel will synchronize calls for 'accept'
loop do
  connection = server.accept
  # spawn a new thread to handle the connection
  Thread.new do
    request = HTTP.parse(connection.gets)
    ...
    connection.puts status
    connection.puts headers
    connection.puts body
    connection.close
  end
end
```

Code Listing 10

4. Performance Bottlenecks

4.1. Introduction

Some metrics contribute heavily to web server performance. Let's go through a few of those and see how they can affect a Ruby web server.

4.2. Data Copying

Servers usually send data back to clients. This data might come from a data fetch and a rendering process but it also can come from a file on disk or data collected from a back end server. In the last two cases the usual approach results in data being copied from kernel space to the user program in the initial step and then from the user program to kernel space once more as it is being sent to the client. There are other ways to send data directly without the extra copying being involved. Such techniques are called zero-copy. One such technique is the sendfile() method which sends data between two file descriptors (files, sockets or pipes) directly via the kernel without passing it through the user program. Zero copy is particularly useful in Ruby since not only memory copying is expensive but also is the creation of Ruby objects and then garbage collecting them.

4.3. Context Switching

There are several forms of context switching. First any system call involves a context switch from the user program to the kernel. Multiple threads and processes also perform context switches to move the old one away and make way for the newly scheduled thread or process. Traditionally Ruby interpreters did not do a great task of handling thread context switches efficiently. This and the fact that excessive context switches are costly even for the most optimized code should point us towards a conclusion that too much threads (or processes, or fibers) is a bad idea. The general rule is to try to keep context switches at a minimum. We will see how the Ruby server fare in that regard.

4.4. Lock Contention

When the server uses multiple threads and processes which share data then this data must be properly locked to protect it from concurrent access. The problem is that locking is an extremely hard subject and you either end up with coarse grained locking which results in too many threads waiting for resources to be freed or even deadlocks if you are not careful or you end up with a very complicated fine grained locking scheme which has a substantial overhead and can easily introduce race conditions. The general rule is to avoid sharing data between your concurrency primitives as much as possible.

4.5. Memory Management

For web servers lots of memory is being allocated and freed frequently. This poses a challenge for the server's memory allocator/deallocator. Even worse, Ruby's garbage collector is famous of being generally slow. Which adds a considerable overhead to the memory management operation. Several techniques could be used to help with this problem. Like allocating the least amount of objects and reusing as much of them as possible. We will look at how different servers are attempting at this problem. The word on the street though is that Ruby is getting a new GC very soon. Peeking into the fresh sources confirms that we are going to see a generational garbage collector in Ruby's next release.

4.6. Blocking Operations

When servers perform any I/O operations they have to either wait for the operation to finish or use some sort of nonblocking I/O. Many servers opt for simple blocking I/O in combination with threads (and or processes) other use nonblocking I/O exclusively. We will look at how the servers make sure that I/O operations don't keep them busy from serving other clients.

4.7. HTTP Parsing

Implementing the HTTP protocol is tricky. Due to many performance and security considerations. We have mentioned earlier that most servers use some derivatives of Mongrel's Ragel based C parser. We will see how this differs from rolling a hand written

parser in Ruby as is the case with Webrick.

4.8. The TCP Stack

The TCP protocol is in the heart of most server operations and hence it has a big impact on the server's performance. The TCP protocol implementations provide a set of option for tuning the TCP operations. These options can have a great effect on performance as they contribute to saving bandwidth and time for certain operations. Example options for Linux are (similar options are available for FreeBSD as well):

- TCP_NO_DELAY, eliminates wait time between transmissions, for keep alive requests
- TCP_CORK, only transmits full packets, very useful for large transfers
- TCP_DEFER_ACCEPT, delays accepting connections until clients send data
- TCP_QUICKACK, whether or not to send ACK flags in their own packets.

We will see how the servers use those options to their benefit.

5. Ruby Web Servers

5.1. Introduction

Earlier in this document we have looked at different concurrency models and explained some of the bottlenecks facing server implementations. We are going to look at some Ruby web servers and which concurrency models they apply and how do they handle the bottlenecks.

The list was based mostly on web server popularity among the Ruby community. It was arranged in a chronological order to show how the Ruby community is adapting its web server implementations as the technology evolves.

Missing from this list are some notable web servers like Ebb, Rucy, Flow, Unicorn and Evented Mongrel. There must be other attempts at web servers that are not known to me as of this writing. Nevertheless, the selected group represent a wide scope of architectures and design strategies that make them a very representative sample.

5.2. WEBrick

WEBrick is a pioneering Ruby server that was created in the year 2000 by TAKAHASHI Masayoshi and GOTOU Yuuzou. It dominated the scene as the leading standalone Ruby server until Mongrel came out in 2006. WEBrick is a feature rich server that is written entirely in Ruby. It has support for stuff like HTTPS, HTTP authentication and listening on several ports. WEBrick has a very modular structure as well. Its core is concerned with network I/O, logging and configuration while it delegates request handling to handlers that have a common interface. WEBrick comes with several handlers out of the box that support static file serving, CGI and other goodies. Extending WEBrick boils down to creating a new handler and adding it to the set of existing ones.

WEBrick is implemented as a single process multi threaded server. Nothing prevents you from starting several WEBricks, each listening on its own port and load balancing between them via and external load balancer. But the server itself does not provide any multiprocessing features of its own.

WEBrick's server is implemented as a never ending loop that checks the ready state of the server socket. It does that via the select call. The main loop will spawn threads upon receiving new connections. Those threads will wait for client connections to have data ready (using another select call) and will them process the requests and invoke the appropriate handlers.

Since WEBrick is written entirely in Ruby, the HTTP parser it employs is hardly know for its performance. Actually it is one of the biggest performance drains for WEBrick. This is due to the fact that parsing requests involves a lot of string processing and generates a lot of garbage. Thus taxing the CPU and the garbage collector heavily.

WEBrick's use of a multiple threads ensures that It can handle requests concurrently but not in parallel since Ruby VMs cannot make use of multiple CPU cores yet. Due to the costs involved with spawning too many threads in Ruby, it is expected that WEBrick will suffer as the number of concurrent clients increases beyond a certain value. The benchmarking results will give us more insight as to how WEBrick scales in the face of high loads.

Diagram 1 shows a simplified explanation of how WEBrick works.

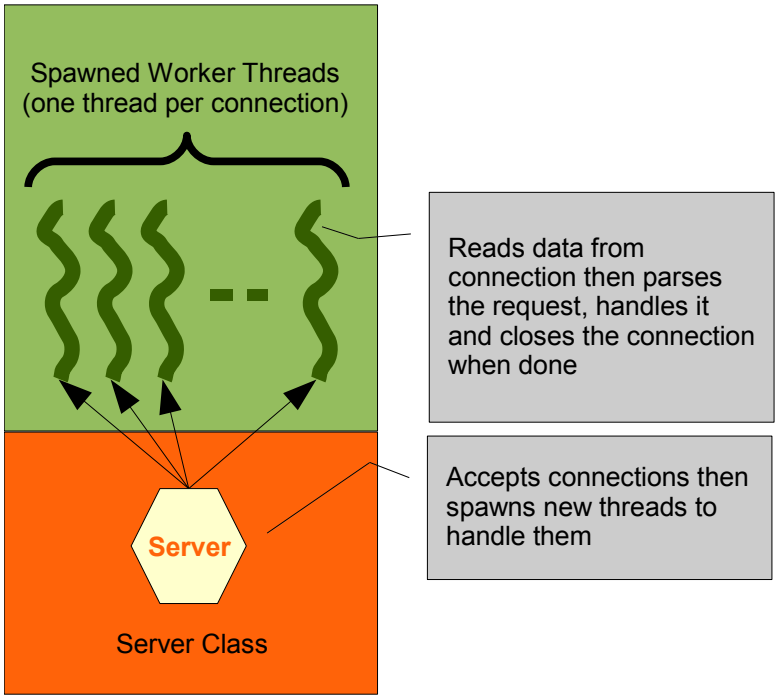


Diagram 1

In brief, WEBrick is a single process multi threaded application. Let's see how it attempts to handle the various bottlenecks we discussed earlier.

1. **Data Copying** WEBrick is seemingly neglecting this issue. No attempts can be seen in the code to minimize data copying.
2. **Context Switching** Since threading is used extensively, even when using select as there's usually a select per connection thread. We can guess that a WEBrick server will do a lot of context switches specially since all I/O operations are done independently and grouping system call (again, like select) are seldom used for that purpose.
3. **Lock Contention** There are hardly any shared data between threads, this makes it easy as too little locking (if any) is needed.
4. **Memory Management** WEBrick does not attempt to conserve memory aggressively. There are hardly signs of reusable objects. A request will generate a good amount of garbage objects that add a sizable overhead to the garbage collector.

5. **Blocking Operations** It appears that WEBrick is setting the connection sockets to nonblocking before passing them to handlers. I am not entirely sure why it is doing that as there are no traces of code that handles blocking exceptions (though highly unlikely as the sockets are not operated upon except after a select call).
6. **HTTP Parsing** As we mentioned earlier, WEBrick parses HTTP in pure Ruby, hence it is rather slow and heavy on the GC.
7. **TCP Stack** There are no traces of code attempting to optimize the TCP Stack.

Preferred Setup

Since Webrick does not utilize multiprocessing on its own and due to the fact that most Ruby web frameworks (except a few exceptions) do not fully utilize multi-threading there must be a way for Webrick to process multiple requests simultaneously. In this case, we start a handful of Webrick servers (each on its own port) and pass requests to them from some proxy server.

Conclusion

WEBrick is a fine little server with loads of features but it builds atop a not very scalable I/O model and doest not attempt to really optimize its internals and handle the known bottlenecks. It is expected to perform poorly under heavy loads.

5.3. Mongrel

Early 2006 saw the earliest announcements of Mongrel on Ruby mailing lists by Zed A. Shaw. Mongrel took the Ruby world by a storm. Since WEBrick was so slow users usually opted for something like FastCGI but this was known to not be a very stable solution. It worked for some but many had issues. Mongrel changed all that. Finally A Ruby web server that is fast enough that you can rely on it on production. After Mongrel reached a stable state it became the preferred deployment method for Ruby on Rails and other Ruby frameworks at the time.

Mongrel excelled by its unique implementation of the HTTP parser. It used a C parser that was auto generated by Ragel from the HTTP specification. This made it very stable and fast parser. And it was enough for Mongrel to surpass Webrick's performance by a wide margin.

Mongrel is composed of a single process that starts an acceptor thread. The acceptor thread runs forever and whenever a connection is received it spawns a new thread to handle the connection. Those spawned threads make use of the C based HTTP parser to parse requests . Much like with Webrick, Mongrel's use of Ruby threads means that it cannot utilize multiple CPU cores and that it will not be able to process requests in parallel, rather they will be processed in a concurrent manner. It also means that each request is penalized by the cost of creating a new thread which is not trivial specially for Ruby 1.9.x which uses native threads.

Aside from Mongrel's concurrency model, it also tries to optimize the TCP Stack. It enlarges the server's socket listening queue to 1024 rather than the 5 provided by the default Ruby implementation. It also sets TCP_CORK on the client connection sockets before serving the response. This better aligns the response packets for optimal bandwidth usage.

One downside is that Mongrel does not support HTTP KEEP ALIVE connections. This means that a new connection has to be established for each new request even if they originate from the same client. Clients usually attempt to reuse the connections that were opened for earlier requests. This prohibits their ability to do that.

Mongrel was usually used in a clustered configuration were several Mongrel processes were started each listening to a different port and a proxy load balancer was used to pass requests to those processes in a round robin fashion. This was the mostly adopted method of deploying

Mongrels since Rails was not able to utilize Mongrel's multiple threads. This approach was the only way for Rails requests to run in parallel. Several proxy servers are paired with Mongrel. The most popular are Apache and Nginx.

Diagram 2 shows a simplified explanation of how Mongrel works.

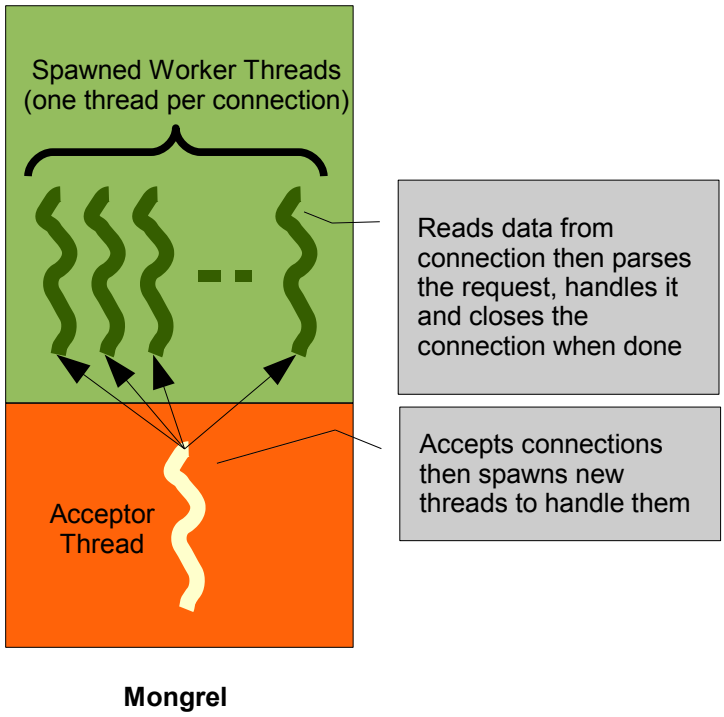


Diagram 2

To sum Mongrel, we can describe it as a single process multi threaded server. Here's how it attempts to handle our set of bottlenecks.

1. **Data Copying** Mongrel does not try to avoid data copying in any way. All I/O goes through the normal Kernel to Process to Kernel paths.
2. **Context Switching** Much like Webrick, Mongrel relies solely on threads to perform concurrent I/O. This means lots of context switches each coming with its cost.
3. **Lock Contention** With almost no data shared between the different threads, locking is largely a non issue for Mongrel. It can become a real issue if applications loaded within Mongrel decide to share data. In such a case synchronization must be introduced and this will most likely introduce some level of lock contention.

4. **Memory Management** Parser objects are reusable in Mongrel. Mongrel also tries to minimize the use of string internally and replaces them with constants in an attempt to reduce the amount of garbage generated during request handling. The efforts do yield an improvement over Webrick's memory usage.
5. **Blocking Operations** Mongrel performs all I/O operations in a blocking manner. This is basically due to its use of threads which will naturally block until I/O is finished
6. **HTTP Parsing** This is where Mongrel shines. Its HTTP parser is also its most touted feature. A correct, secure and fast implementation in the form of auto generated C code.
7. **TCP Stack** Mongrel optimizes the C stack using 2 methods. It increases the count of connections that can wait in the server socket's listen queue. It also uses TCP_CORK to optimize bandwidth usage.

Preferred Setup

Much like Webrick. Mongrel is usually setup as a cluster of processes (each with its own port) behind a proxy server like Apache and Nginx. Those will handle the static file serving and pass dynamic requests to the mongrel cluster.

Conclusion

Mongrel is the server that has the motto “faster is possible” and indeed it was a lot faster than WEBrick. Even though it builds on the same concurrency model that Webrick uses it tries hard to optimize the request path. The parser that it uses was widely adopted by other server writers as we will see next. We expect Mongrel to provide decent performance even under the face of moderate to high concurrent loads.

5.4. Thin

With the start of 2008 Marc-Andre Cournoyer released the first version of Thin. The new thing about it was its departure from the thread-per-request practice of the former Ruby servers. Rather it relied on the established EventMachine library for very fast, single threaded based I/O. Actually what was cool about Thin was its integration of three solid components. EventMachine as the I/O back-end, Mongrel's infamous parser and Rack as a Ruby web application interface.

With these three components Thin offered out of the box performance and scalability that exceeded any of the former offerings. Exceeding Webrick is understandable since it uses Mongrel's much faster C based parser. But how can Thin outperform Mongrel when it is sharing that bulk of code with it?.

As mentioned earlier, Thin departs from the thread-per-model and opts for an asynchronous event loop. As we discussed earlier, those have several advantages over threaded implementations since the require less memory and are free of context switching overheads.

Another point for Thin in the Thin vs. Mongrel issue is its support HTTP KEEP ALIVE connections. Which means much less overhead with clients that will request multiple resources from the server.

Internally Thin starts it event loop and waits for notifications on incoming requests. Once a request arrives a connection is established and is then inserted in the event loop for notifications on when it should be read from or written to. Aside from the network I/O, requests are passed to Rack applications for processing. Due to the way Rack works and the single threaded nature of Thin; only one request can be inside the Rack container at a time.

Recently several attempts have been made to avoid this limitation. Thin now offers an option which will cause requests to be dispatched in a background pool of 20 threads. Another very recent attempt is work done for asynchronous Rack processing.

Generally, Thin's request handling can be described as illustrated in Diagram 3:

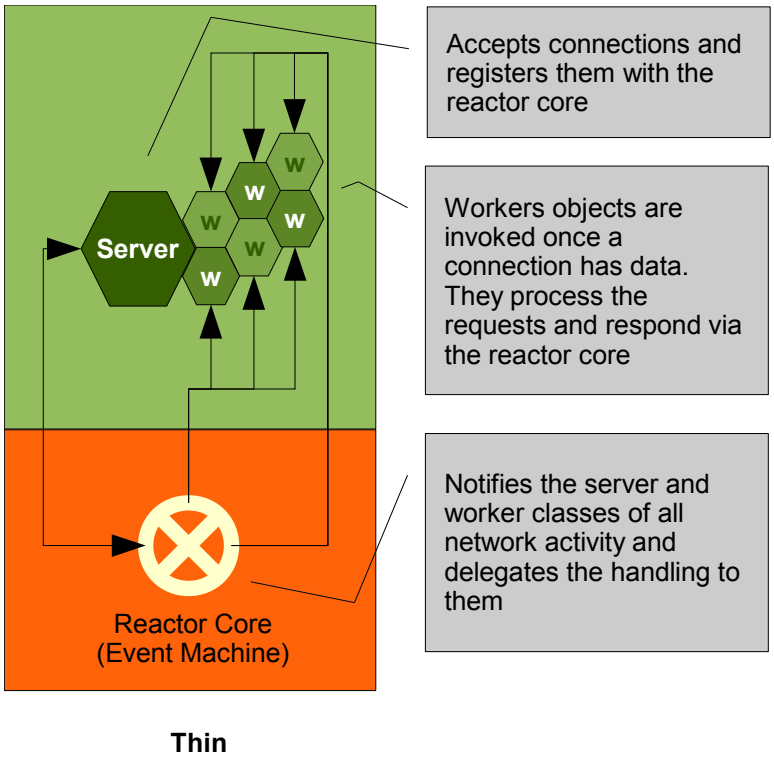


Diagram 3

So basically Thin is a single process single threaded server that uses an asynchronous event loop. It also has the option to dispatch requests to a thread pool. Looking at how it attempts to solve the different bottlenecks we described earlier we can see the following:

1. **Data Copying** Thin does not attempt to avoid data copying in any way it even delegates static file request handling to a very generic Rack module which fairs badly in that regard.
2. **Context Switching** Here is where Thin shines. It is a single stack implementation and hence does not context switch at all, except for the occasional error handling or if you use the thread pool option.
3. **Lock Contention** There is virtually no lock contention. Except for the case when the thread pool is used is which case threads must synchronize their use of the event loop.
4. **Memory Management** Thin tries hard to avoid generating many garbage objects during request processing. It even tries to optimize all hash lookups by using constants

instead of strings. Where it fails though is in how it retrieves responses from Rack. It does so in a way that guarantees that the data is buffered twice in memory before it is being sent. This can easily lead to memory fragmentation specially with large response bodies.

5. **Blocking Operations** Since Thin is relying in EventMachine most of the I/O operations are done in a non-blocking manner. This helps a lot in reducing server latencies.
6. **HTTP Parsing** Much like Mongrel. Thin uses the C based Ragel generated parser. It also adds a few options like keep alive and chunked encoding support.
7. **TCP Stack** Thin does not try to optimize the TCP stack. Apparently it is relying on EventMachine for this. But till this date, EventMachine does not perform any TCP tuning to its sockets.

Preferred Setup

Like Mongrel and Webrick before it. Thin is usually setup as a cluster of processes (each with its own port) behind a proxy server of some sort. It is recommended that the proxy handles static file serving to offload that duty from the Thin cluster. Apache and Nginx are the most popular options.

Conclusion

Thin deserves its name. It is a minimalistic server that acts as a thin glue for several robust libraries. The careful selection of libraries and the use of event loops meant a server was built that is more scalable and faster than the competition. Before we go into benchmarking we expect that Thin will easily outperform Mongrel and Webrick, even as we scale to high concurrent loads specially if we server quick small requests.

5.5. Passenger

The servers we described earlier are all self contained web server that can handle a request as soon as it hits the physical server ports. The guys at Phusion thought differently. There are many proven web servers out there and the part where Ruby is needed is running Ruby applications rather than building the whole web stack. The idea is rather than build a standalone server you augment an already existing one with Ruby processes that will handle requests specific to the Ruby application. Furthermore, instead of instantiating those processes for every request it will prefork a group of them and keep them around for faster response time. Sounds a lot like CGI and FastCGI? Well yes, the Phusion team created their own FastCGI look-a-like. Their implementation is much less general though and excels at what it is intended to do.

Given a web server like Apache or Nginx, Passenger is added to it as a module. This module is configured to handle certain types of requests. Basically the web server falls back to it when it cannot find a static file to serve in the configured location for the Ruby application. When the module starts it initiates a Ruby process that we can regard as a master process since it will be responsible for all the Ruby processes handling the application. The master then forks the worker processes. Each of those workers is connected to the master and it also initiates a UNIX socket connection with the module.

Once the server passes a request over to the module it sends it to one of the worker processes. The request is sent in a certain format agreed upon by the module and the workers. Along with the request information the module also sends the client connection itself using the UNIX socket send_io facility. At this point the web server can totally ignore this client and even close the connection from its side. Passenger workers will be handling it from this point forward.

The worker is basically a single threaded process that handles one request at a time. This limits the ability of Passenger servers to handle concurrent requests beyond the number of forked processes. Passengers ships with a special version of Ruby though that is optimized for the forking model and reportedly saves memory by a factor of one third over vanilla Ruby.

Thus enabling the use of more worker processes per application. We will be doing our testing using the stock Ruby 1.9.1 though which does not utilize these changes.

Diagram 4 describes the internals of Passenger:

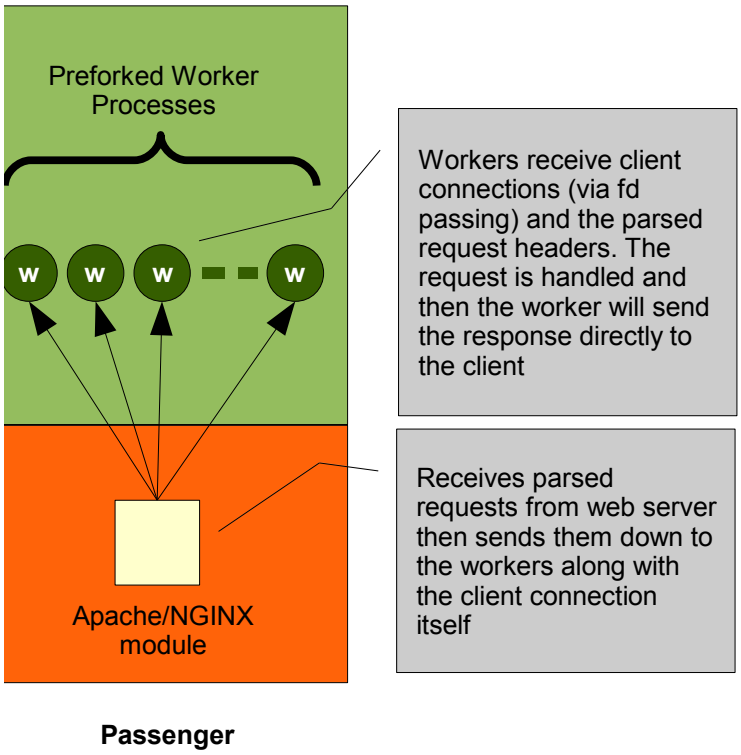


Diagram 4

Passenger is the first genuinely multi-process server that appears in this review. Even though it is common practice to use a cluster of Mongrel or Thin servers behind a proxy it requires manual configuration and each process occupies its own port. This does not equate the transparent multiprocessing capabilities provided by Passenger. Passenger also differs in how it attempts to handle the bottlenecks as follows:

1. **Data Copying** Passenger does not do much about this internally but it does a great job by passing the file descriptors to worker processes hence not needing to send data to the clients through the parent web server. All other implementations pass data to upstream proxies before it is sent to the clients.
2. **Context Switching** This is limited to process context switching. If not many worker processes are used then the application will not face a lot of context switches while

being able to utilize multiple CPU cores at the same time.

3. **Lock Contention** No shared data means no lock contentions. Passenger does not rely on sharing inter process data.
4. **Memory Management** This is the ares that most effort was put to make Passenger use less memory. The Fork friendly Ruby implementation plus the ability of the master process to recycle worker processes from time to time ensure that Passenger is the most memory efficient Ruby application server.
5. **Blocking Operations** Not much is done here as the processes are all single threaded. Multi threaded workers have appeared recently and might be used more as Rails and other Rack libraries get more robust thread safety.
6. **HTTP Parsing** Passenger provides no HTTP parser at all. It relies instead on the web server which parsed the request already. This makes lots of sense and prevents duplicating efforts.
7. **TCP Stack** On its own, Passenger does not optimize the TCP stack a lot. But the web server can set some TCP parameters on the connection before it is sent to Passenger.

Preferred Setup

Passenger is the easiest server to setup in this survey. You only need to configure Apache or Nginx and then forget about it. Passenger will control the process loading and reloading if needed. Using Ruby 1.9.x with Passenger does it a disservice though, as Passenger can be coupled with Ruby Enterprise Edition and use it in conjunction with UNIX's Copy-On-Write semantics to greatly reduce memory consumption of your servers.

Conclusion

Passenger can be described as a (finally) working FastCGI for Ruby. Along with several memory saving optimizations and a dead easy configuration it makes much sense that it suddenly became the preferable deployment method for Ruby web applications. The model does not scale very well though due to the lack of a scalable concurrency model. But it fares well for most Ruby web applications that are inherently single threaded, like most Ruby on Rails based applications.

6. Benchmarks

6.1. Introduction

After the discussion of the several architectures we wanted to verify that the expectations will actually map lab testing results. So a battery of benchmark tests were prepared and each of the servers took its turn in running those tests against.

6.2. Benchmarking Methodology

The servers where tested against various workloads. The workloads were constructed to stress various aspects of the servers. The following tests were performed:

1. Static File Serving
    1. Serving a very small 200B file
    2. Serving a small 1KB file
    3. Serving a small 30KB file
    4. Serving a medium 126KB file
    5. Serving a large 1MB file
    6. Serving a large 10MB file
    7. Serving a very large 170MB file
2. Dynamic content serving
    1. A Rails application that returns “Hello World!”
    2. A Rails application that fetches a record from DB and renders an ERB template
    3. A Rails application that fetches a record from DB and renders it as YAML
    4. A Rails application that does heavy processing (Marshals lots of objects)
    5. A Rails application that sends a large 1MB response
    6. A Rails application that sends a large 10MB response
3. Multi-process dynamic content serving (2 processes proxied by Apache)
    1. A Rails application that returns “Hello World!”
    2. A Rails application that fetches a record from DB and renders an ERB template
    3. A Rails application that fetches a record from DB and renders it as YAML
    4. A Rails application that does heavy processing (Marshals lots of objects)

5. A Rails application that sends a large 1MB response
6. A Rails application that sends a large 10MB response

Thin was tested twice, once with worker threads enabled and once when they were disabled. Passenger was not tested for static file serving. Apache Bench (ab) was used for benchmarking.

6.3. Test Setup

The tests were done on a DELL Inspiron 1520 laptop. It is equipped with Core2Duo T7000 dual core CPU that is clocked at 2.0GHZ and is equipped with 2MB of cache. The system also has 2GB of RAM and a 5400RPM hard drive.

The System had only one of the servers working on any given time to make sure all resources were dedicated to the active test.

Some tests were repeated several times and anomalies were excluded. Long running tests were not repeated as they ran long enough to produce stable results. Only when anomalies where detect the long running tests were repeated to ensure a constant behavior is being reported.

A warm up run was done to each server before an actual testing round to make sure all required files are loaded and that the caches are filled.

Since this is a dual core processor we only used one process for all the servers. This was due to the fact that the tests hardly had any blocking I/O operations and that the 2 cores will be split between the server under test and Apache Bench itself as we did all the tests on the back loop to eliminate network overhead.

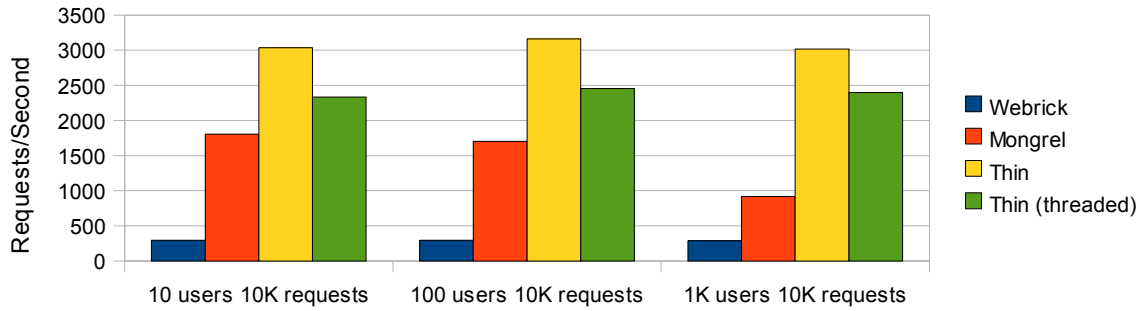
6.4. Results

6.4.1. Serving Static Files

Serving a very small 200B file

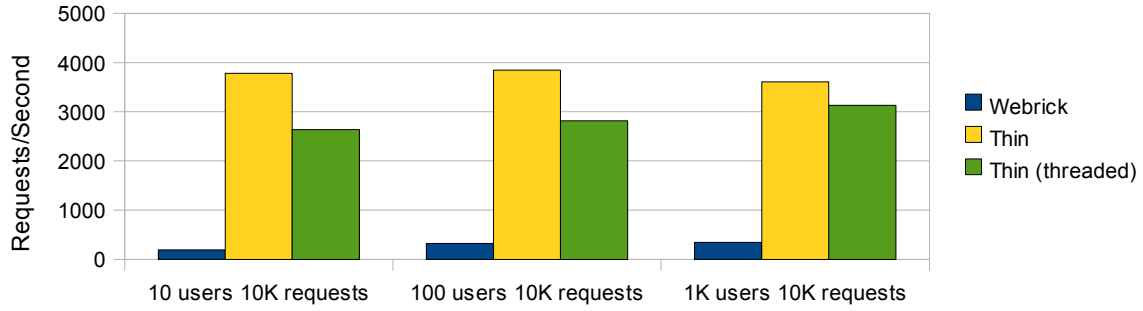
Normal

	Webrick	Mongrel	Thin	Thin (threaded)
10 users 10K requests	293[1]	1805	3034	2335
100 users 10K requests	295	1704	3163	2456
1K users 10K requests	288	915	3016	2398



Keep alive

	Webrick	Thin	Thin (threaded)
10 users 10K requests	194	3780	2636
100 users 10K requests	320	3847	2814
1K users 10K requests	343	3610	3129

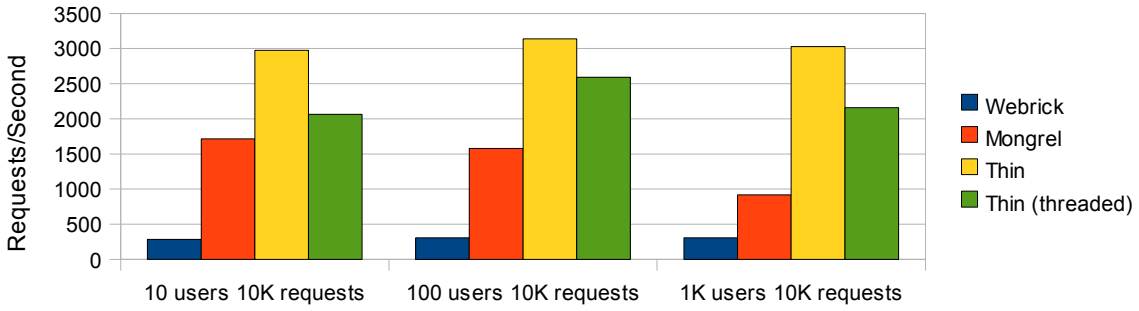


1 All results represent the number of requests per second

Serving a small 1KB file

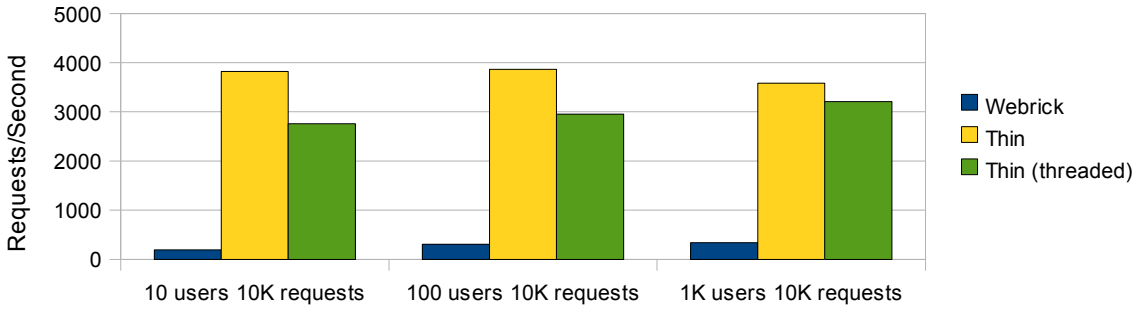
Normal

	Webrick	Mongrel	Thin	Thin (threaded)
10 users 10K requests	284	1715	2976	2065
100 users 10K requests	307	1579	3137	2590
1K users 10K requests	306	915	3027	2159



Keep alive

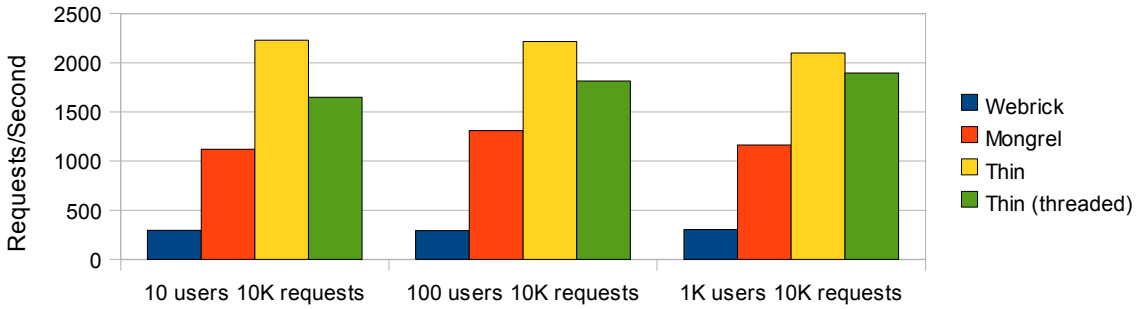
	Webrick	Thin	Thin (threaded)
10 users 10K requests	195	3818	2755
100 users 10K requests	307	3864	2952
1K users 10K requests	338	3581	3205



Serving a small 30KB file

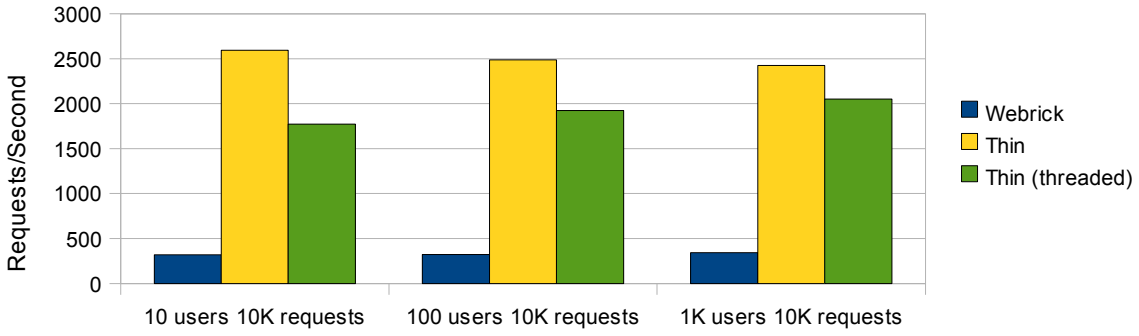
Normal

	Webrick	Mongrel	Thin	Thin (threaded)
10 users 10K requests	297	1120	2230	1650
100 users 10K requests	293	1310	2217	1816
1K users 10K requests	304	1163	2099	1895



Keep alive

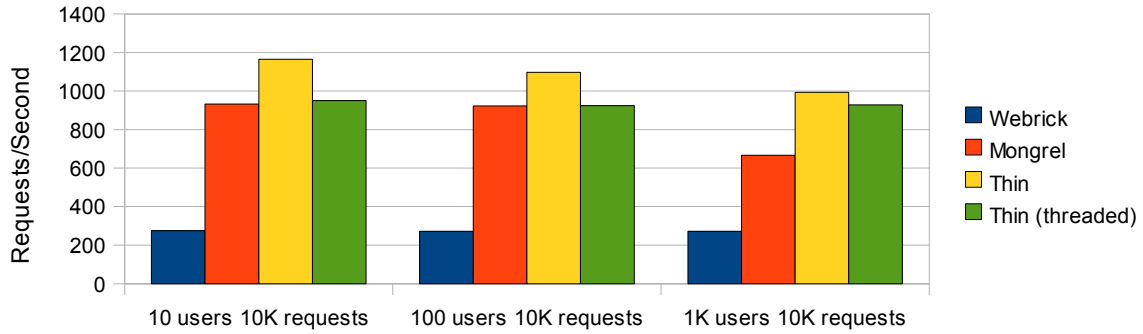
	Webrick	Thin	Thin (threaded)
10 users 10K requests	320	2595	1774
100 users 10K requests	323	2487	1925
1K users 10K requests	342	2425	2051



Serving a medium 126KB file

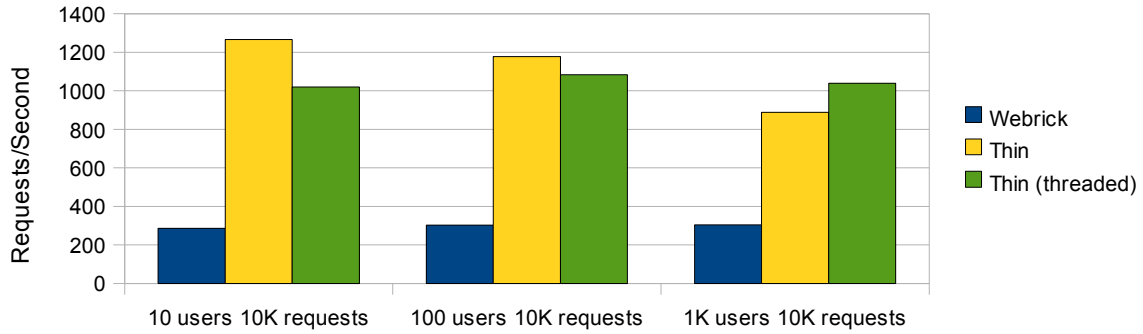
Normal

	Webrick	Mongrel	Thin	Thin (threaded)
10 users 10K requests	275	932	1165	950
100 users 10K requests	272	922	1097	924
1K users 10K requests	272	667	994	928



Keep alive

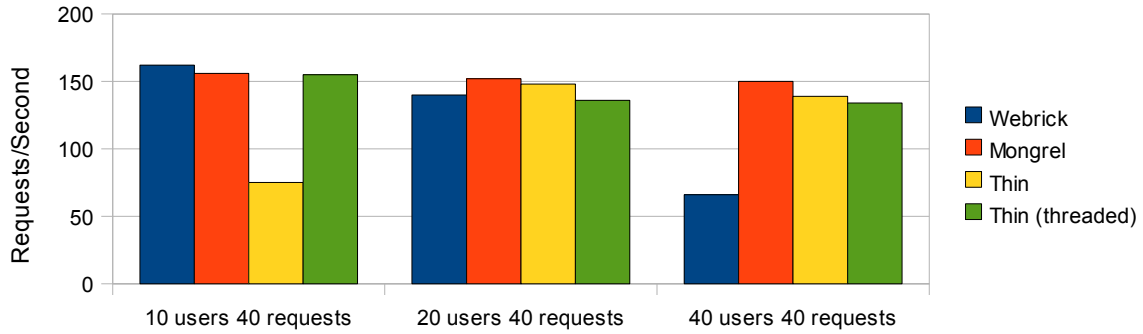
	Webrick	Thin	Thin (threaded)
10 users 10K requests	286	1266	1019
100 users 10K requests	303	1178	1084
1K users 10K requests	304	888	1039



Serving a large 1MB file

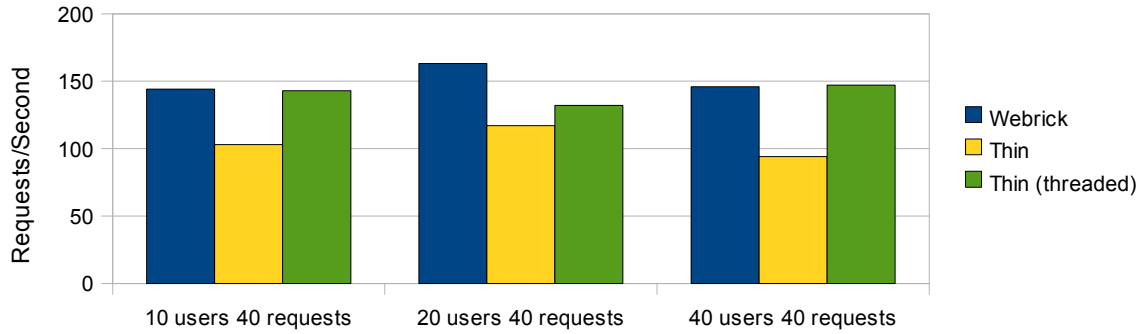
Normal

	Webrick	Mongrel	Thin	Thin (threaded)
10 users 40 requests	162	156	75	155
20 users 40 requests	140	152	148	136
40 users 40 requests	66	150	139	134



Keep alive

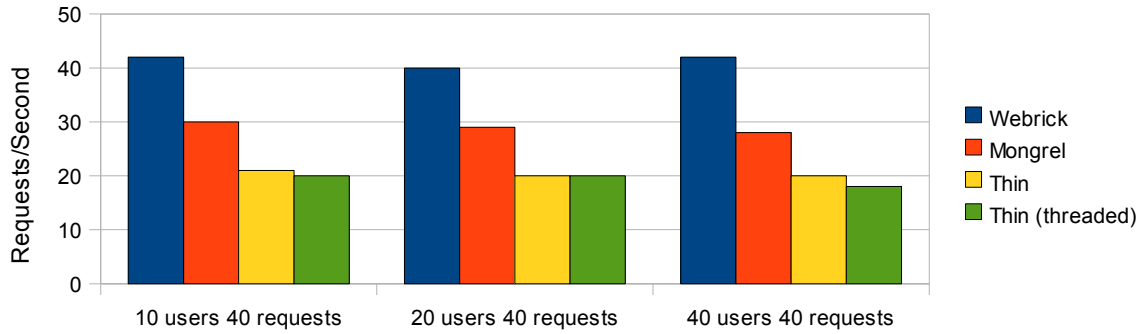
	Webrick	Thin	Thin (threaded)
10 users 40 requests	144	103	143
20 users 40 requests	163	117	132
40 users 40 requests	146	94	147



Serving a large 9MB file

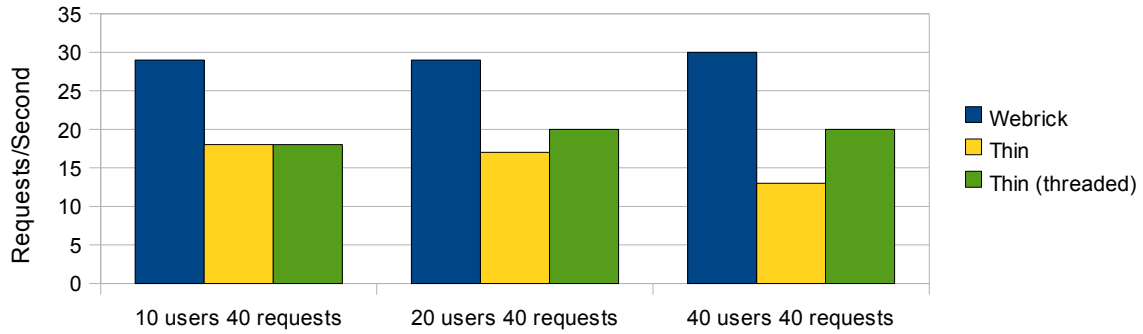
Normal

	Webrick	Mongrel	Thin	Thin (threaded)
10 users 40 requests	42	30	21	20
20 users 40 requests	40	29	20	20
40 users 40 requests	42	28	20	18



Keep alive

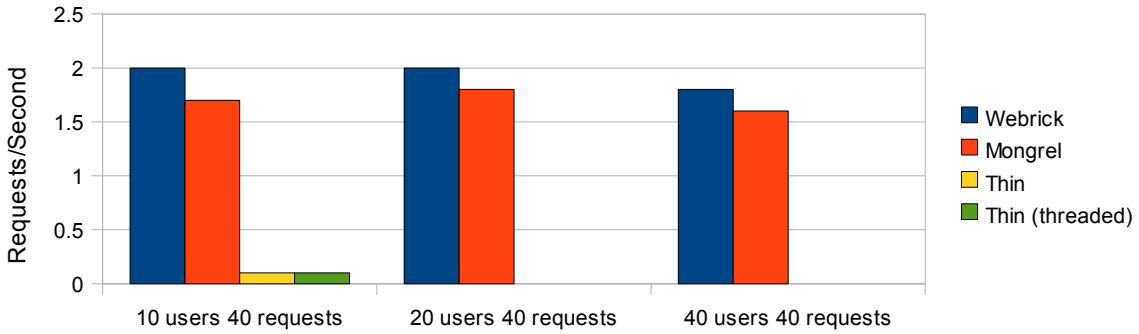
	Webrick	Thin	Thin (threaded)
10 users 40 requests	29	18	18
20 users 40 requests	29	17	20
40 users 40 requests	30	13	20



Serving a huge 170MB file

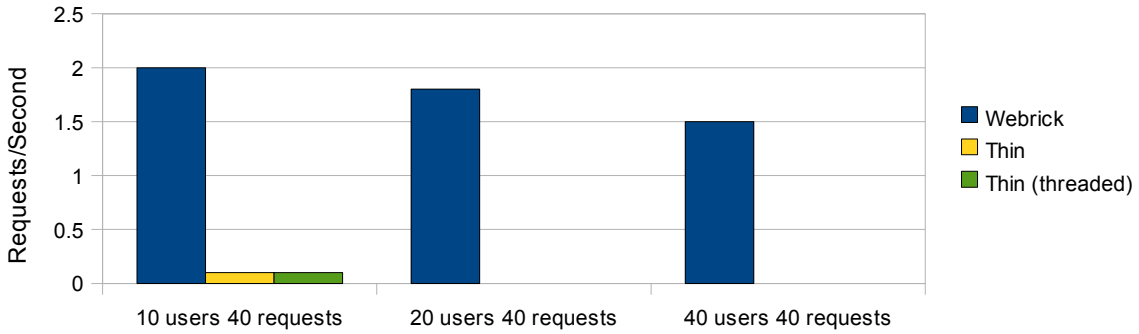
Normal

	Webrick	Mongrel	Thin	Thin (threaded)
10 users 40 requests	2	1.7	0.1	0.1
20 users 40 requests	2	1.8	FAILED	FAILED
40 users 40 requests	1.8	1.6	FAILED	FAILED



Keep alive

	Webrick	Thin	Thin (threaded)
10 users 40 requests	2	0.1	0.1
20 users 40 requests	1.8	FAILED	FAILED
40 users 40 requests	1.5	FAILED	FAILED

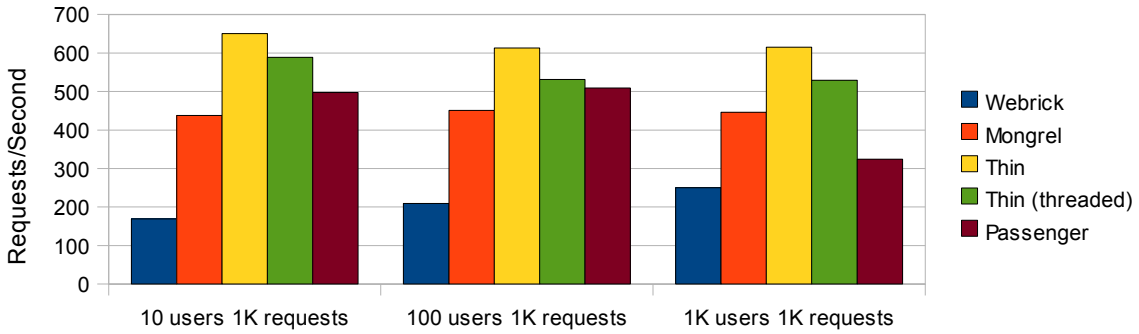


6.4.2. Serving Dynamic Requests (Single Rails Process)

Serving 'Hello World!'

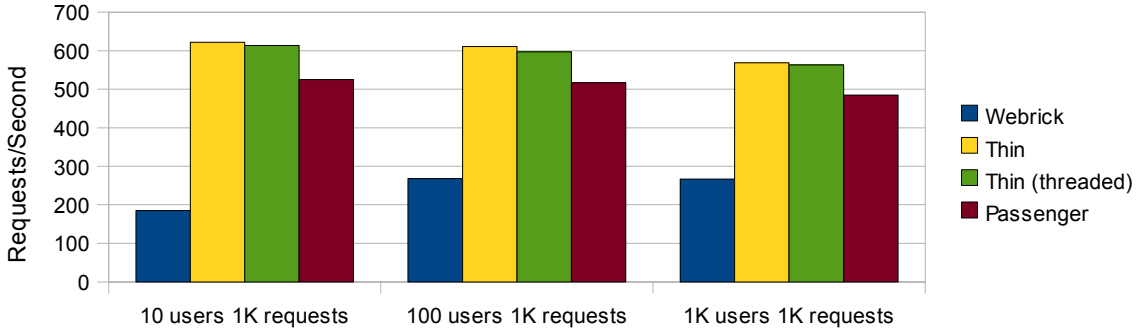
Normal

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 1K requests	169	438	650	589	497
100 users 1K requests	209	451	613	531	509
1K users 1K requests	250	446	615	529	324



Keep alive

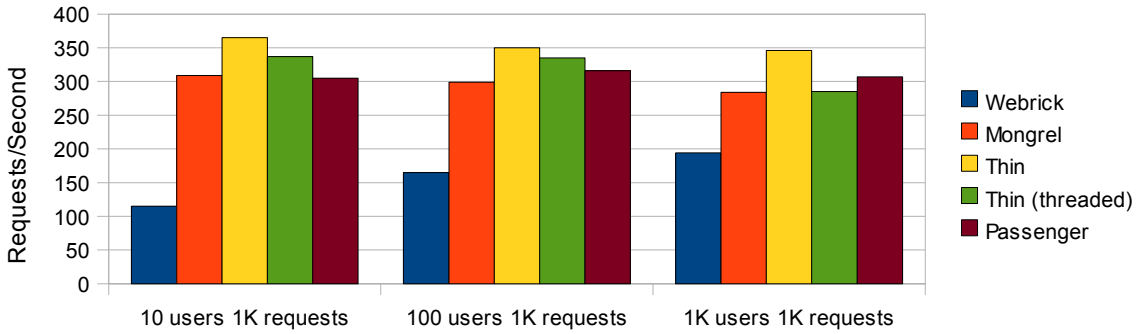
	Webrick	Thin	Thin (threaded)	Passenger
10 users 1K requests	185	622	614	525
100 users 1K requests	268	611	597	517
1K users 1K requests	267	569	563	485



Fetching a record and rendering an ERB template

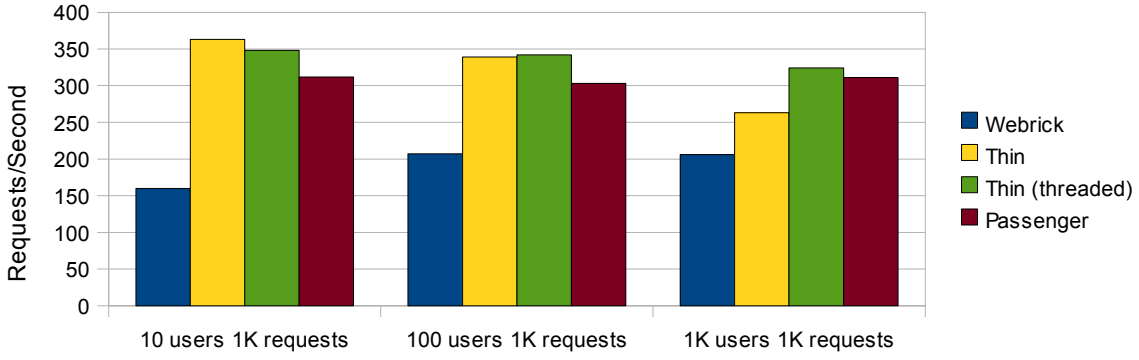
Normal

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 1K requests	115	309	365	337	305
100 users 1K requests	165	299	350	335	316
1K users 1K requests	194	284	346	285	307



Keep alive

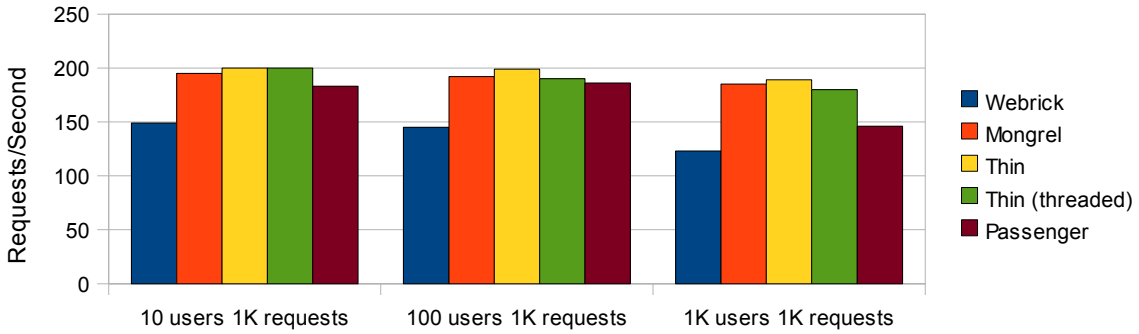
	Webrick	Thin	Thin (threaded)	Passenger
10 users 1K requests	160	363	348	312
100 users 1K requests	207	339	342	303
1K users 1K requests	206	263	324	311



Fetching a record and rendering it as YAML

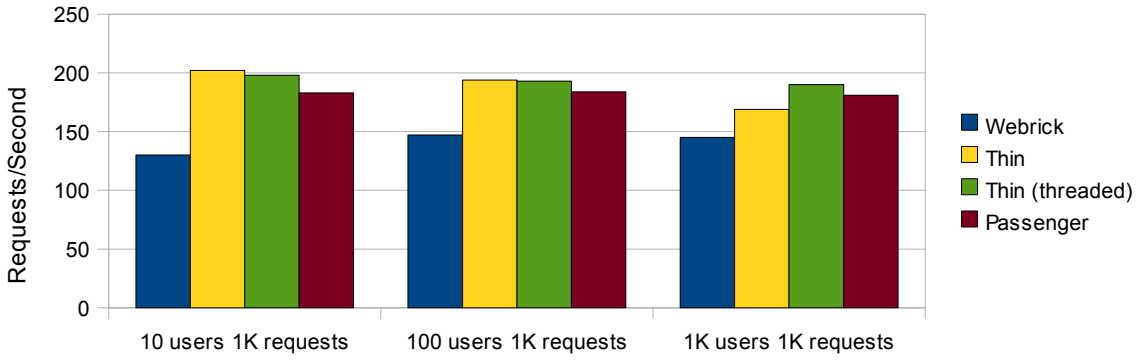
Normal

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 1K requests	149	195	200	200	183
100 users 1K requests	145	192	199	190	186
1K users 1K requests	123	185	189	180	146



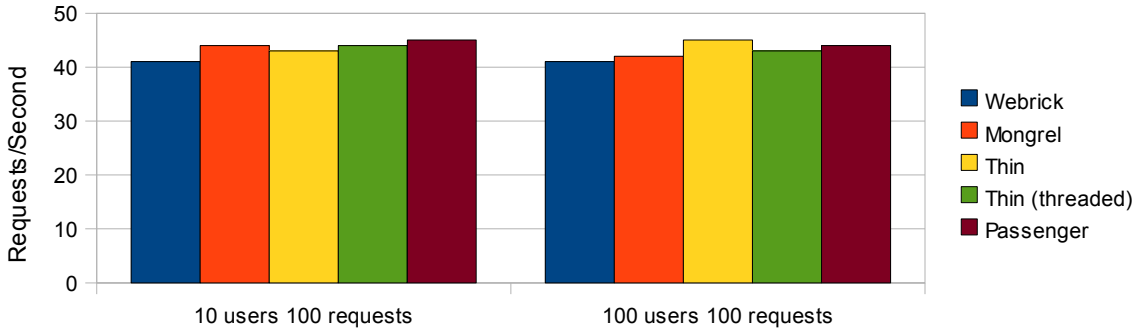
Keep alive

	Webrick	Thin	Thin (threaded)	Passenger
10 users 1K requests	130	202	198	183
100 users 1K requests	147	194	193	184
1K users 1K requests	145	169	190	181



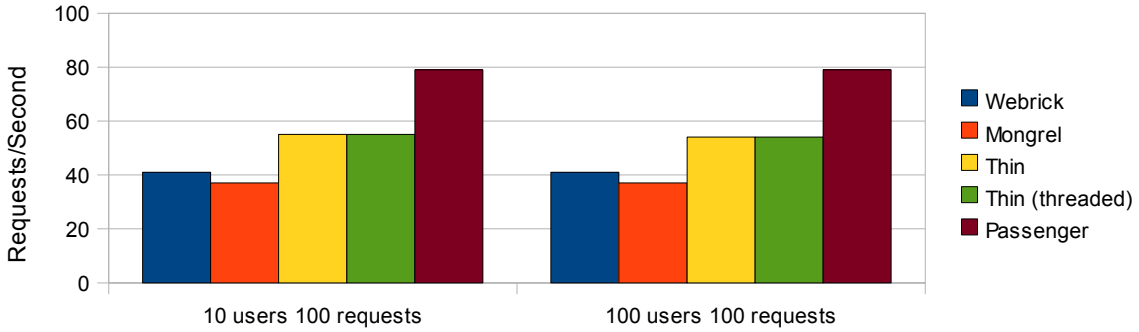
Marshalling lots of objects (500 complex objects per request)

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 100 requests	41	44	43	44	45
100 users 100 requests	41	42	45	43	44



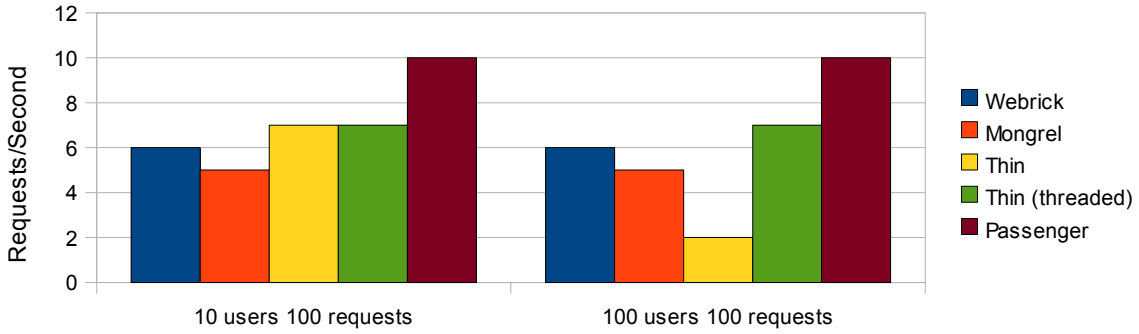
Sending a large 1MB response

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 100 requests	41	37	55	55	79
100 users 100 requests	41	37	54	54	79



Sending a very large 10MB response

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 100 requests	6	5	7	7	10
100 users 100 requests	6	5	2	7	10

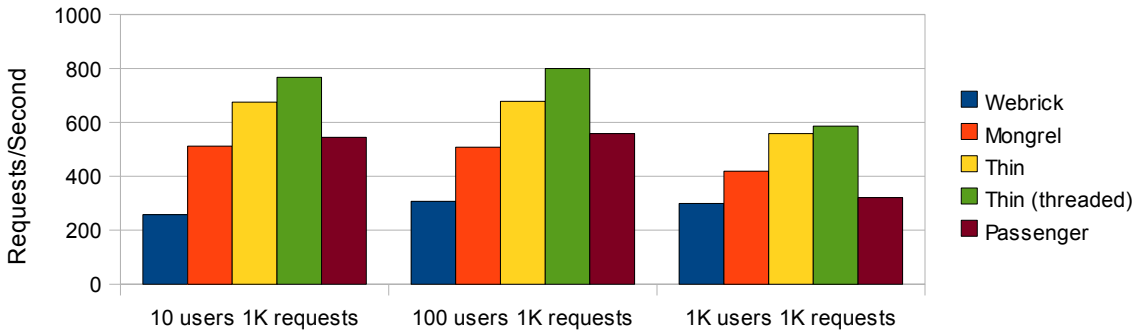


6.4.3. Serving Dynamic Requests From A Proxied Cluster (2 Processes)

Serving 'Hello World!'

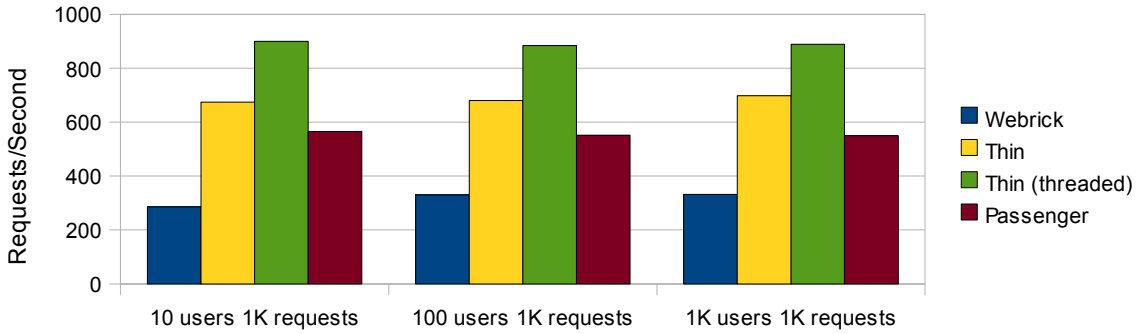
Normal

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 1K requests	258	512	675	767	544
100 users 1K requests	307	508	678	800	558
1K users 1K requests	299	419	558	586	321



Keep alive

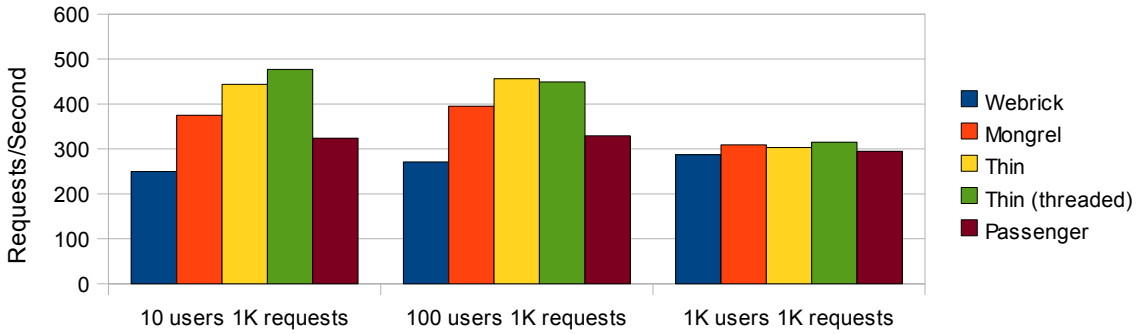
	Webrick	Thin	Thin (threaded)	Passenger
10 users 1K requests	285	674	900	565
100 users 1K requests	330	680	884	552
1K users 1K requests	332	698	889	549



Fetching a record and rendering an ERB template

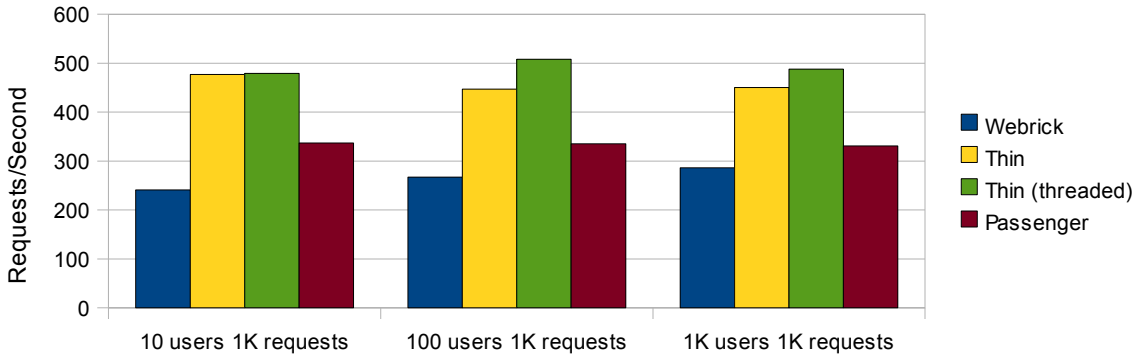
Normal

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 1K requests	250	375	444	477	324
100 users 1K requests	271	395	456	449	329
1K users 1K requests	287	309	303	315	295



Keep alive

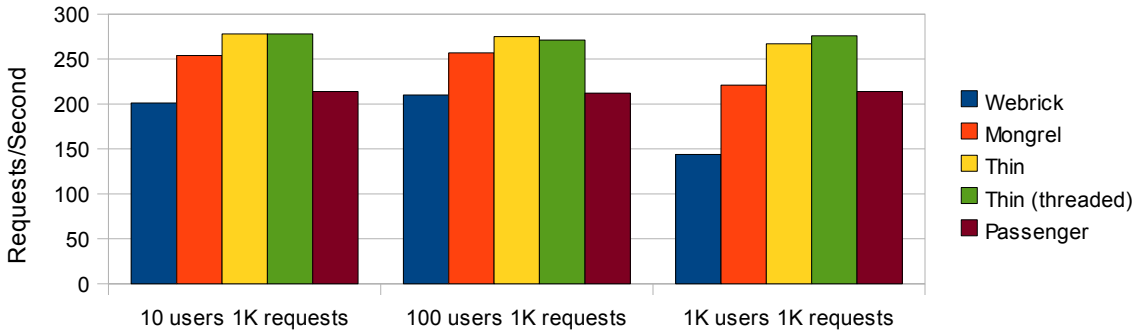
	Webrick	Thin	Thin (threaded)	Passenger
10 users 1K requests	241	477	479	337
100 users 1K requests	267	447	508	335
1K users 1K requests	286	450	488	331



Fetching a record and rendering it as YAML

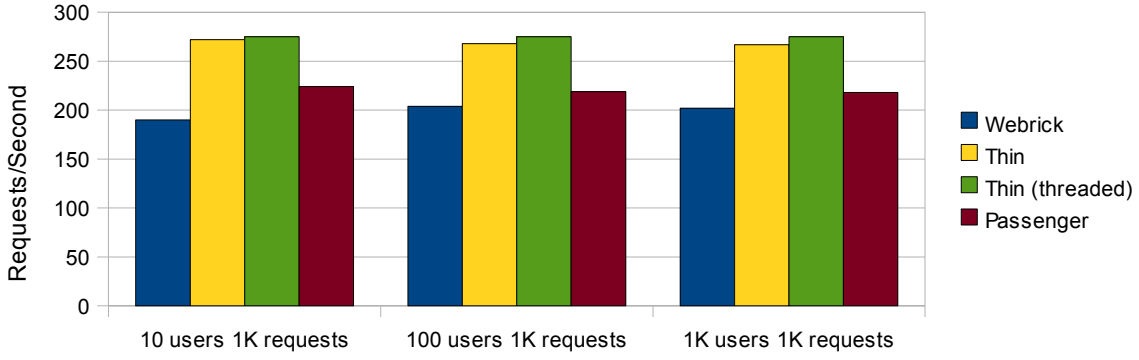
Normal

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 1K requests	201	254	278	278	214
100 users 1K requests	210	257	275	271	212
1K users 1K requests	144	221	267	276	214



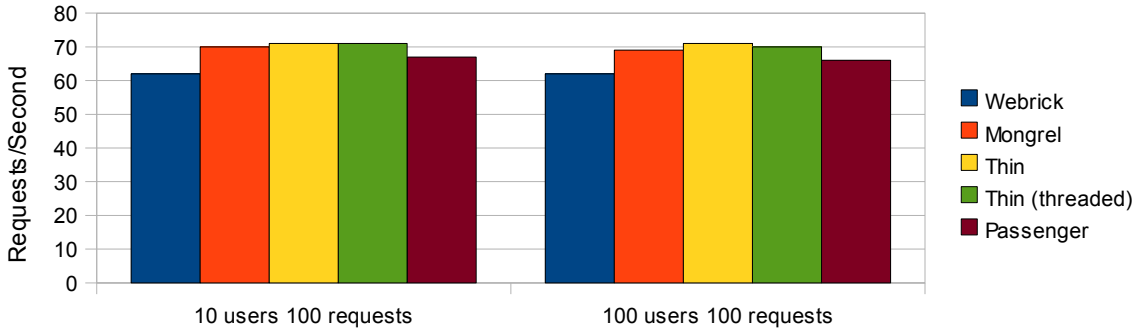
Keep alive

	Webrick	Thin	Thin (threaded)	Passenger
10 users 1K requests	190	272	275	224
100 users 1K requests	204	268	275	219
1K users 1K requests	202	267	275	218



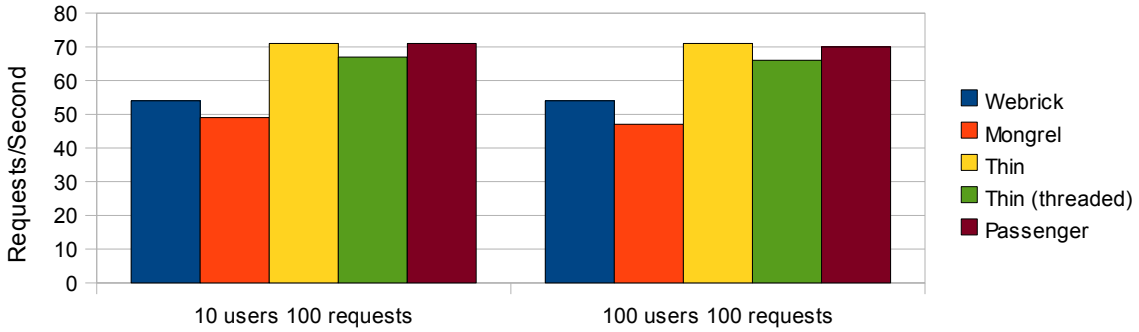
Marshalling lots of objects (500 complex objects per request)

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 100 requests	62	70	71	71	67
100 users 100 requests	62	69	71	70	66



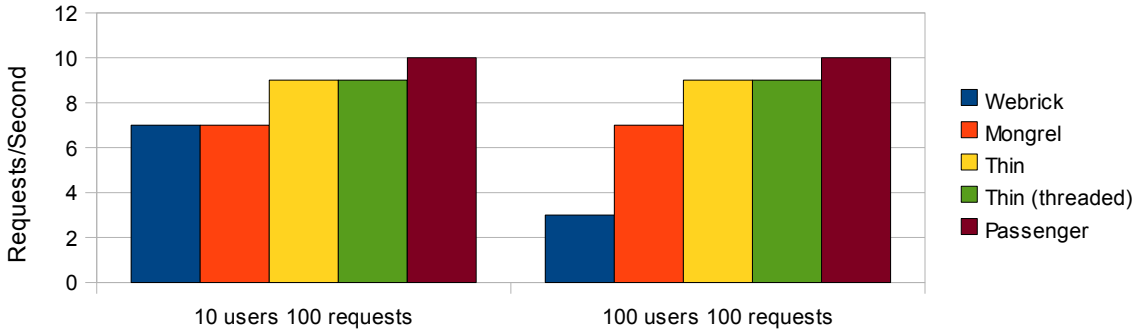
Sending a large 1MB response

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 100 requests	54	49	71	67	71
100 users 100 requests	54	47	71	66	70



Sending a very large 10MB response

	Webrick	Mongrel	Thin	Thin (threaded)	Passenger
10 users 100 requests	7	7	9	9	10
100 users 100 requests	3	7	9	9	10



7. Benchmark Analysis

7.1. Static File Serving Performance

Generally speaking it is not recommended to utilize your Ruby server in static file serving. It is better to offload this task to something like Apache or Nginx and free up the Ruby server to serve dynamic requests (in Rails or any other Ruby script).

Nevertheless, these tests are interesting since they simulate (to an extent) how the servers would fare when serving cached content. In a cached content situation there is no much data processing and rendering, you just fetch the cache then send it through the wire. This is very similar to static content serving.

The benchmarks can be split to two categories, small file sizes and large file sizes. Let's take a closer look at each.

7.1.1. Small File Sizes

It is apparent that Thin leads the pack with a healthy margin, approaching a very respectable 4000 request/sec in one of the cases. While still less than half what Nginx could deliver in the same situation it is still a very good show from an Ruby based server. Second comes Thin again but in the threaded flavor, apparently threads and EventMachine are not happy delivering too many quick responses. It is worth noting though that for all threaded Thin tests the epoll back-end was disabled. Third comes Mongrel, which delivers roughly half to two-thirds the performance of Thin. Then it the tail of theses tests Webrick establishes its presence. Too slow to be even considered, many times it was 10x as slow as the competition.

Passenger was not tested here since it relies on the host web server (either Apache or Nginx) for static file serving.

7.1.2.Large File Sizes

I was in for a good surprise when I performed the large file tests. I used a 1MB, a 9MB and a 170MB files. The results was so strange I kept repeating the tests but always getting the same

results. As the file size increases the performance profiles are completely reversed. Now Webrick is leading the pack, Mongrel is second and Thin is trailing, with the threaded a version a little faster than the pure evented one. This is the exact opposite of the small file results. In some of the largest file tests Thin would even completely fail to deliver the files. Let's attempt to understand what is going on here.

It is expected that Webrick and Mongrel will deliver similar performance. They are both built using a very similar I/O strategy, Mongrel's advantage, which is the much faster HTTP parser would not make much of a difference for long running requests which will be saturated by disk and subsequently network I/O. Still, one would expect that Mongrel will outperform Webrick by even a small margin due to its use to TCP-CORK which provides better TCP packet alignment. Nevertheless, the results are not a huge departure from the expectations even though they were tilted in the other direction. The real problem here are the results for the Thin server. It's very strange that the fastest of the group suddenly becomes the slowest, even failing at times.

Digging deeper into Thin's file serving code reveals the culprit. Thin is relying on the Rack::File module for file serving. Rack::File is handled much like any other Rack middle-ware. Thin will grab the response from the Rack::File module, buffer it and then give way to EventMachine to send it as it sees fit. What happens is that the file (multiple instances of it if we have concurrent clients) will be fully loaded in memory before the response is sent. Here's how it works inside Thin:

1. When a client sends a group of concurrent requests they will be buffered in the Thin socket's kernel buffers
2. EventMachine will notice that the socket is now readable and will loop 10 times each time reading one request from the socket and passing it to Thin (obviously the loop will be broken if there are less than 10 requests)
3. Thin parses the request and passes it to Rack::File. Thin then reads the response 4KB at a time from Rack::File and sends it via the EventMachine::Connection.
4. During the whole response reading process, Thin does not give way for the EventMachine to send any of the buffered data.

5. This process continues for (at max) 9 more requests (EventMachine attempts to read requests from the server socket 10 at a time), all accumulating the responses in the EventMachine buffers before a single byte is actually sent.
6. Once the batch of incoming requests are done, EventMachine starts to send responses from those requests and accept new ones if any.

As one can see from the above, sending 10 concurrent requests for 170MB files results in roughly 1.6GB memory usage before Thin starts sending back responses and freeing memory.

This explains why Thin fails in some benchmarks, it simply consumed all the system memory and hitting the disk results in very low performance. To the point that responses are delayed beyond the timeout set by Apache Bench.

So, why does Thin perform a little better with the support of threads? Let's see the processing flow in that case and try to understand.

1. When a client sends a group of concurrent requests they will be buffered in the Thin socket's kernel buffers
2. EventMachine will notice that the socket is now readable and will loop 10 times each time reading one request from the socket and passing it to Thin (obviously the loop will be broken if there are less than 10 requests)
3. Thin parses the request and passes it to Rack::File in a deferred thread. After the file is located and opened the thread returns the response. The main EventMachine thread will then run a call back which will start reading the response 4KB at a time from Rack::File and sends it via the EventMachine::Connection.
4. The main difference here is that before running the callback (and between callbacks) EventMachine is free to perform its own operations so it gets a chance (a slim one that is) to send accumulated data before processing callbacks of further requests.

This is why the threaded flavor of Thin outperforms the pure evented one when serving large files. There is also an interesting observation here. EventMachine attempts to accept at most 10 requests every time the server socket is readable. In practice this means that for long running requests you can get a considerable delay with high concurrency. Consider the

following scenario:

1. Server is idle, waiting for connections
2. 20 requests come at the same time (each takes 1 second to process)
3. Socket is flagged as readable, no writable sockets (yet)
4. First connection is extracted, request is processed (1 second)
5. Resulting data from step 4 now buffered waiting for a new reactor run
6. Steps 4 & 5 are repeated 9 more times (10 seconds passed now)
7. After the 10 requests are processed the reactor is finally free for another run
8. Now the reactor checks its sockets again, we have 10 writable sockets with data on them. But we also have a server socket with 10 pending connections.
9. The reactor will process with the server socket first and repeat the steps from 4 to 6
10. Now that 20 seconds has passed, it starts to process the writable sockets and send back responses.

Assuming a negligible response sending time we find that the requests were answered in the following manner, min : 20s, avg : 20s, max : 20s. On the other hand, if we were able to send the responses immediately it could've been, min : 1s, avg : 10s, max : 20s which is a much better distribution.

7.2. Dynamic Request Serving Performance

We have 3 categories of requests tested. The first are quick requests with little processing and small responses. The second are requests with average response sizes but heavy processing. The third are requests with little processing but large responses. I/O time was not a major factor in most of the tests. Since the dataset is really small and it was surely present in the database server's cache.

7.2.1. Quick and Small Responses

Thin leads the pack as expected. It is really optimized for those work loads. Passenger and Mongrel fight for the second position. Passenger exhibits a strange behavior though, while it passes Mongrel in the single process tests, it is a little bit slower in the dual process tests. Needless to say, Webrick is the lowest performer by an embarrassingly wide margin.

7.2.1. Heavy Processing but Small Responses

Once all servers start consuming a lot of CPU the picture changes a lot. The difference have all evaporated and we see all the servers performing almost the same. This is due to the fact that the differences in the request handling pipeline are negligible when compared to long runs of CPU crunching code.

7.2.1. Little Processing but Large Responses

Now Passenger is leading the pack. It is much better on the system memory even though it is also buffering everything in memory. But since it handle one request at a time per process it doesn't eat crazy memory amounts like Thin does. Thin follows as the second server here which was surprising since Webrick and Mongrel were faster in large static file tests. Apparently the file serving code of Thin is guilty. And while Thin is an overall very good server the file serving module drags it behind on large files. Hence it is able to exceed its rivals on large dynamic responses. The rest of the pack remain the same with Webrick exceeding Mongrel a little.

8. Conclusion

I should not be making conclusions given the abundance of the presented data which can help you decide on your own. But I know that some people may want to skip all the bloat and read directly what the author thinks he found out. So, without further ado, here are my conclusions for those deploying Ruby 1.9.x applications.

1. Offload your file serving to the likes of Nginx and Apache (in that order)
2. It won't matter much which server you go with in the case of Rails (unless it is called Webrick, which should be avoided). For ease of use I recommend Passenger even though Thin is a little faster for quick requests.
3. If you are building an API or a service that perform fast and quick operations then look no further than Thin. This one will excel in those applications.

References

1. The C10K Problem, http://www.kegel.com/c10k.html
2. Benchmarking BSD and Linux http://bulk.fefe.de/scalability/
3. High-Performance Server Architecture http://pl.atyp.us/content/tech/servers.html
4. Mongrel Web Server http://mongrel.rubyforge.org/
5. Thin Web Server http://code.macournoyer.com/thin/
6. Phusion Passenger (Mod Rails) http://www.modrails.com/
7. Ruby Enterprise Edition http://www.rubyenterpriseedition.com/
8. Event Machine http://rubyeventmachine.com/
9. Ruby Event Machine – The Speed Demon http://www.igvita.com/2008/05/27/ruby-eventmachine-the-speed-demon/
10. Concurrency is a Myth in Ruby http://www.igvita.com/2008/11/13/concurrency-is-a-myth-in-ruby/
11. I/O models: how you move your data matters http://timetobleed.com/io-models-how-you-move-your-data-matters/
12. Threading models: So many different ways to get stuff done http://timetobleed.com/threading-models-so-many-different-ways-to-get-stuff-done/
13. Why Events Are a Bad Idea (for High Concurrency Servers) http://www.usenix.org/events/hotos03/tech/vonbehren.html