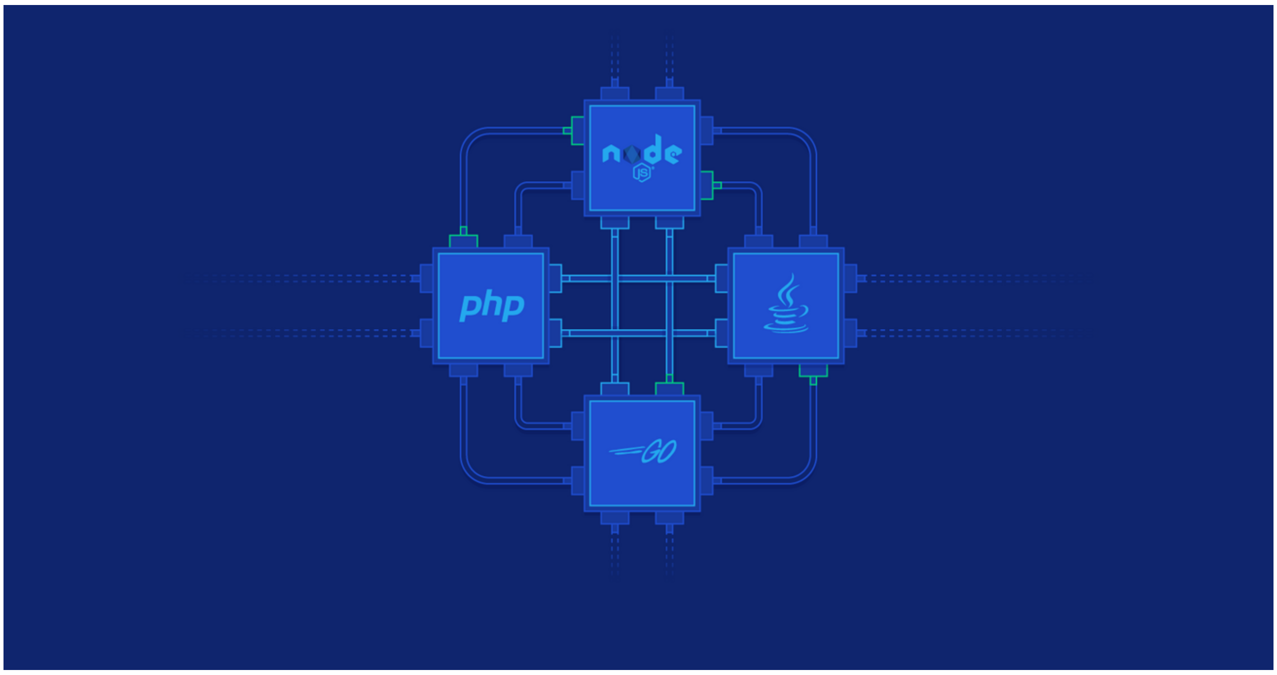
Server-side I/O: Node vs. PHP vs. Java vs. Go

toptal.com/back-end/server-side-io-performance-node-php-java-go

Brad Peabody



Understanding the Input/Output (I/O) model of your application can mean the difference between an application that deals with the load it is subjected to, and one that crumples in the face of real-world use cases. Perhaps while your application is small and does not serve high loads, it may matter far less. But as your application's traffic load increases, working with the wrong I/O model can get you into a world of hurt.

And like most any situation where multiple approaches are possible, it's not just a matter of which one is better, it's a matter of understanding the tradeoffs. Let's take a walk across the I/O landscape and see what we can spy.

In this article, we'll conduct a backend language performance comparison. We'll examine Node, Java, Go, and PHP with Apache (Go versus Java performance, Node.js versus Java performance, etc.), discussing how the different languages model their I/O, the advantages and disadvantages of each model, and conclude with some rudimentary performance benchmarks. If you're concerned about the I/O performance of your next web application, this article is for you.

I/O Basics: A Quick Refresher

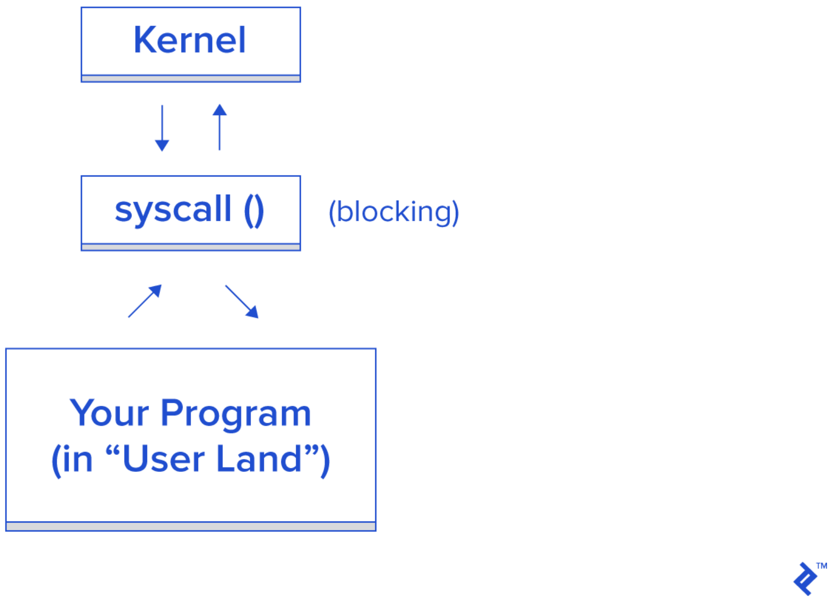
To understand the factors involved with I/O, we must first review the concepts down at the operating system level. While it is unlikely that will have to deal with many of these concepts directly, you deal with them indirectly through your application's runtime environment all the time. And the details matter.

System Calls

Firstly, we have system calls, which can be described as follows:

- Your program (in “user land,” as they say) must ask the operating system kernel to perform an I/O operation on its behalf.
- A “syscall” is the means by which your program asks the kernel do something. The specifics of how this is implemented vary between OSes but the basic concept is the same. There is going to be some specific instruction that transfers control from your program over to the kernel (like a function call but with some special sauce specifically for dealing with this situation). Generally speaking, syscalls are blocking, meaning your program waits for the kernel to return back to your code.
- The kernel performs the underlying I/O operation on the physical device in question (disk, network card, etc.) and replies to the syscall. In the real world, the kernel might have to do a number of things to fulfill your request including waiting for the device to be ready, updating its internal state, etc., but as an application developer, you don’t care about that. That’s the kernel’s job.

Syscalls



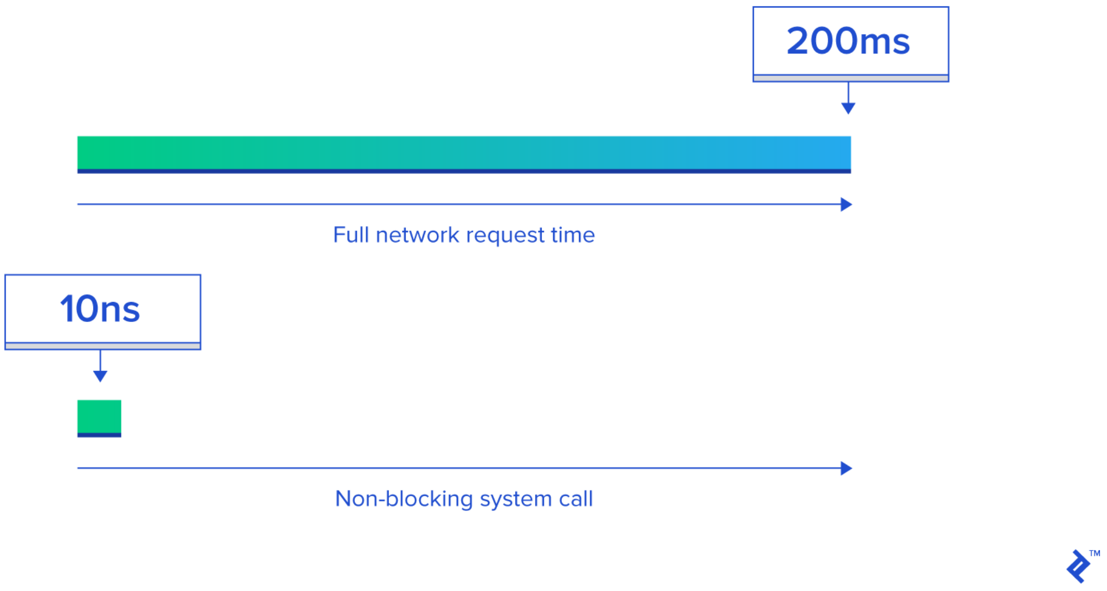
Blocking vs. Non-blocking Calls

Now, I just said above that syscalls are blocking, and that is true in a general sense. However, some calls are categorized as “non-blocking,” which means that the kernel takes your request, puts it in queue or buffer somewhere, and then immediately returns without waiting for the actual I/O to occur. So it “blocks” for only a very brief time period, just long enough to enqueue your request.

Some examples (of Linux syscalls) might help clarify: - `read()` is a blocking call - you pass it a handle saying which file and a buffer of where to deliver the data it reads, and the call returns when the data is there. Note that this has the advantage of being nice and simple. - `epoll_create()`, `epoll_ctl()` and `epoll_wait()` are calls that, respectively, let you create a group of handles to listen on, add/remove handlers from that group and then block until there is any activity. This allows you to efficiently control a large number of I/O operations with a single thread, but I'm getting ahead of myself. This is great if you need the functionality, but as you can see it's certainly more complex to use.

It's important to understand the order of magnitude of difference in timing here. If a CPU core is running at 3GHz, without getting into optimizations the CPU can do, it's performing 3 billion cycles per second (or 3 cycles per nanosecond). A non-blocking system call might take on the order of 10s of cycles to complete - or "a relatively few nanoseconds". A call that blocks for information being received over the network might take a much longer time - let's say for example 200 milliseconds (1/5 of a second). And let's say, for example, the non-blocking call took 20 nanoseconds, and the blocking call took 200,000,000 nanoseconds. Your process just waited 10 million times longer for the blocking call.

Time Difference Between a Non-Blocking System Call and a Network Request



The kernel provides the means to do both blocking I/O ("read from this network connection and give me the data") and non-blocking I/O ("tell me when any of these network connections have new data"). And which mechanism is used will block the calling process for dramatically different lengths of time.

Scheduling

The third thing that's critical to follow is what happens when you have a lot of threads or processes that start blocking.

For our purposes, there is not a huge difference between a thread and process. In real life, the most noticeable performance-related difference is that since threads share the same memory, and processes each have their own memory space, making separate processes tends to take up a lot more memory. But when we're talking about scheduling, what it really boils down to is a list of things (threads and processes alike) that each need to get a slice of execution time on the available CPU cores. If you have 300 threads running and 8 cores to run them on, you have to divide the time up so each one gets its share, with each core running for a short period of time and then moving onto the next thread. This is done through a “context switch,” making the CPU switch from running one thread/process to the next.

These context switches have a cost associated with them - they take some time. In some fast cases, it may be less than 100 nanoseconds, but it is not uncommon for it to take 1000 nanoseconds or longer depending on the implementation details, processor speed/architecture, CPU cache, etc.

And the more threads (or processes), the more context switching. When we’re talking about thousands of threads, and hundreds of nanoseconds for each, things can get very slow.

However, non-blocking calls in essence tell the kernel “only call me when you have some new data or event on one of any of these connections.” These non-blocking calls are designed to efficiently handle large I/O loads and reduce context switching.

With me so far? Because now comes the fun part: Let’s look at what some popular languages do with these tools and draw some conclusions about the tradeoffs between ease of use and performance… and other interesting tidbits.

As a note, while the examples shown in this article are trivial (and partial, with only the relevant bits shown); database access, external caching systems (memcache, et. all) and anything that requires I/O is going to end up performing some sort of I/O call under the hood which will have the same effect as the simple examples shown. Also, for the scenarios where the I/O is described as “blocking” (PHP, Java), the HTTP request and response reads and writes are themselves blocking calls: Again, more I/O hidden in the system with its attendant performance issues to take into account.

There are a lot of factors that go into choosing a programming language for a project. There are even a lot factors when you only consider performance. But, if you are concerned that your program will be constrained primarily by I/O, if I/O performance is make or break for your project, these are things you need to know.

The “Keep It Simple” Approach: PHP

Back in the 90’s, a lot of people were wearing Converse shoes and writing CGI scripts in Perl. Then PHP came along and, as much as some people like to rag on it, it made making dynamic web pages much easier.

The model PHP uses is fairly simple. There are some variations to it but your average PHP server looks like:

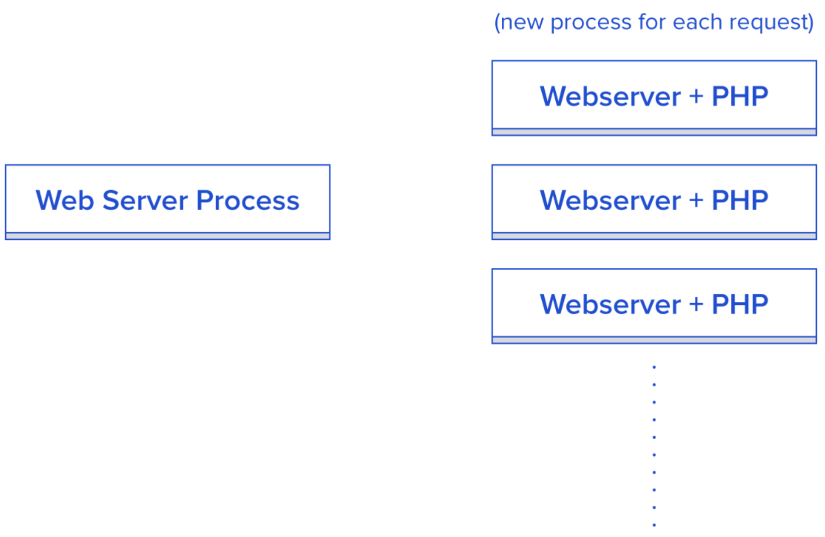
An HTTP request comes in from a user’s browser and hits your Apache web server. Apache creates a separate process for each request, with some optimizations to re-use them in order to minimize how many it has to do (creating processes is, relatively speaking, slow). Apache calls PHP and tells it to run the appropriate `.php` file on the disk. PHP code executes and does blocking I/O calls. You call `file_get_contents()` in PHP and under the hood it makes `read()` syscalls and waits for the results.

And of course the actual code is simply embedded right into your page, and operations are blocking:

```
<?php

// blocking file I/O
$file_data = file_get_contents('/path/to/file.dat');

// blocking network I/O
$curl = curl_init('http://example.com/example-microservice');
$result = curl_exec($curl);

// some more blocking network I/O
$result = $db->query('SELECT id, data FROM examples ORDER BY id DESC limit 100');

?>
```

In terms of how this integrates with system, it's like this:

I/O Model PHP



Pretty simple: one process per request. I/O calls just block. Advantage? It's simple and it works. Disadvantage? Hit it with 20,000 clients concurrently and your server will burst into flames. This approach does not scale well because the tools provided by the kernel for dealing with high volume I/O (epoll, etc.) are not being used. And to add insult to injury, running a separate process for each request tends to use a lot of system resources, especially memory, which is often the first thing you run out of in a scenario like this.

Note: The approach used for Ruby is very similar to that of PHP, and in a broad, general, hand-wavy way they can be considered the same for our purposes.

The Multithreaded Approach: Java

So Java comes along, right about the time you bought your first domain name and it was cool to just randomly say “dot com” after a sentence. And Java has multithreading built into the language, which (especially for when it was created) is pretty awesome.

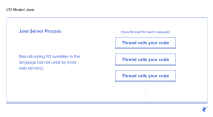
Most Java web servers work by starting a new thread of execution for each request that comes in and then in this thread eventually calling the function that you, as the application developer, wrote.

Doing I/O in a Java Servlet tends to look something like:

```
public void doGet(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException
{

        // blocking file I/O
        InputStream fileIs = new FileInputStream("/path/to/file");

        // blocking network I/O
        URLConnection urlConnection = (new URL("http://example.com/example-
microservice")).openConnection();
        InputStream netIs = urlConnection.getInputStream();

        // some more blocking network I/O
out.println("...");
}
```

Since our `doGet` method above corresponds to one request and is run in its own thread, instead of a separate process for each request which requires its own memory, we have a separate thread. This has some nice perks, like being able to share state, cached data, etc. between threads because they can access each other’s memory, but the impact on how it interacts with the schedule it still almost identical to what is being done in the PHP example previously. Each request gets a new thread and the various I/O operations block inside that thread until the request is fully handled. Threads are pooled to minimize the cost of creating and destroying them, but still, thousands of connections means thousands of threads which is bad for the scheduler.

An important milestone is that in version 1.4 Java (and a significant upgrade again in 1.7) gained the ability to do non-blocking I/O calls. Most applications, web and otherwise, don’t use it, but at least it’s available. Some Java web servers try to take advantage of this in various ways; however, the vast majority of deployed Java applications still work as described above.


I/O Model Java
Java Server Process
(new thread for each request)
Thread calls your code
(Non-blocking I/O available in the language but not used by most web servers.)
Thread calls your code
Thread calls your code

Java gets us closer and certainly has some good out-of-the-box functionality for I/O, but it still doesn’t really solve the problem of what happens when you have a heavily I/O bound application that is getting pounded into the ground with many thousands of blocking threads.

Non-blocking I/O as a First Class Citizen: Node

The popular kid on the block when it comes to better I/O is Node.js. Anyone who has had even the briefest introduction to Node has been told that it's "non-blocking" and that it handles I/O efficiently. And this is true in a general sense. But the devil is in the details and the means by which this witchcraft was achieved matter when it comes to performance.

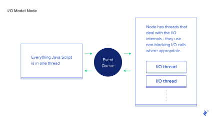
Essentially the paradigm shift that Node implements is that instead of essentially saying "write your code here to handle the request", they instead say "write code here to start handling the request." Each time you need to do something that involves I/O, you make the request and give a callback function which Node will call when it's done.

Typical Node code for doing an I/O operation in a request goes like this:

```
http.createServer(function(request, response) {
        fs.readFile('/path/to/file', 'utf8', function(err, data) {
                response.end(data);
        });
});
```

As you can see, there are two callback functions here. The first gets called when a request starts, and the second gets called when the file data is available.

What this does is basically give Node an opportunity to efficiently handle the I/O in between these callbacks. A scenario where it would be even more relevant is where you are doing a database call in Node, but I won't bother with the example because it's the exact same principle: You start the database call, and give Node a callback function, it performs the I/O operations separately using non-blocking calls and then invokes your callback function when the data you asked for is available. This mechanism of queuing up I/O calls and letting Node handle it and then getting a callback is called the "Event Loop." And it works pretty well.


I/O Model Node
Everything Java Script is in one thread
Event Queue
Node has threads that deal with the I/O internals - they use non-blocking I/O calls where appropriate.
I/O thread
I/O thread

There is however a catch to this model. Under the hood, the reason for it has a lot more to do with how the V8 JavaScript engine (Chrome's JS engine that is used by Node) is implemented [1] than anything else. The JS code that you write all runs in a single thread. Think about that for a moment. It means that while I/O is performed using efficient non-blocking techniques, your JS can that is doing CPU-bound operations runs in a single thread, each chunk of code blocking the next. A common example of where this might come up is looping over database records to process them in some way before outputting them to the client. Here's an example that shows how that works:

```
var handler = function(request, response) {

        connection.query('SELECT ...', function (err, rows) {

                if (err) { throw err };

                for (var i = 0; i < rows.length; i++) {
                        // do processing on each row
                }

                response.end(...); // write out the results

        })

};
```

While Node does handle the I/O efficiently, that `for` loop in the example above is using CPU cycles inside your one and only main thread. This means that if you have 10,000 connections, that loop could bring your entire application to a crawl, depending on how long it takes. Each request must share a slice of time, one at a time, in your main thread.

The premise this whole concept is based on is that the I/O operations are the slowest part, thus it is most important to handle those efficiently, even if it means doing other processing serially. This is true in some cases, but not in all.

The other point is that, and while this is only an opinion, it can be quite tiresome writing a bunch of nested callbacks and some argue that it makes the code significantly harder to follow. It's not uncommon to see callbacks nested four, five, or even more levels deep inside Node code.

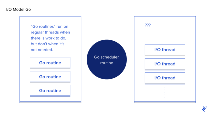
We're back again to the trade-offs. The Node model works well if your main performance problem is I/O. However, its achilles heel is that you can go into a function that is handling an HTTP request and put in CPU-intensive code and bring every connection to a crawl if you're not careful.

Naturally Non-blocking: Go

Before I get into the section for Go, it's appropriate for me to disclose that I am a Go fanboy. I've used it for many projects and I'm openly a proponent of its productivity advantages, and I see them in my work when I use it.

That said, let's look at how it deals with I/O. One key feature of the Go language is that it contains its own scheduler. Instead of each thread of execution corresponding to a single OS thread, it works with the concept of "goroutines." And the Go runtime can assign a goroutine to an OS thread and have it execute, or suspend it and have it not be associated with an OS thread, based on what that goroutine is doing. Each request that comes in from Go's HTTP server is handled in a separate Goroutine.

The diagram of how the scheduler works looks like this:



Under the hood, this is implemented by various points in the Go runtime that implement the I/O call by making the request to write/read/connect/etc., put the current goroutine to sleep, with the information to wake the goroutine back up when further action can be taken.

In effect, the Go runtime is doing something not terribly dissimilar to what Node is doing, except that the callback mechanism is built into the implementation of the I/O call and interacts with the scheduler automatically. It also does not suffer from the restriction of having to have all of your handler code run in the same thread, Go will automatically map your Goroutines to as many OS threads it deems appropriate based on the logic in its scheduler. The result is code like this:

```
func ServeHTTP(w http.ResponseWriter, r *http.Request) {

        // the underlying network call here is non-blocking
        rows, err := db.Query("SELECT ...")

        for _, row := range rows {
                // do something with the rows,
// each request in its own goroutine
        }

        w.Write(...) // write the response, also non-blocking

}
```

As you can see above, the basic code structure of what we are doing resembles that of the more simplistic approaches, and yet achieves non-blocking I/O under the hood.

In most cases, this ends up being "the best of both worlds." Non-blocking I/O is used for all of the important things, but your code looks like it is blocking and thus tends to be simpler to understand and maintain. The interaction between the Go scheduler and the OS scheduler handles the rest. It's not complete magic, and if you build a large system, it's worth putting in the time to understand more detail about how it works; but at the same time, the environment you get "out-of-the-box" works and scales quite well.

Go may have its faults, but generally speaking, the way it handles I/O is not among them.

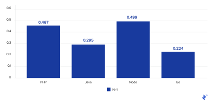
Lies, Damned Lies and Benchmarks

It is difficult to give exact timings on the context switching involved with these various models. I could also argue that it's less useful to you. So instead, I'll give you some basic benchmarks that compare overall HTTP server performance of these server environments. Bear in mind that a lot of factors are involved in the performance of the entire end-to-end HTTP request/response path, and the numbers presented here are just some samples I put together to give a basic comparison.

For each of these environments, I wrote the appropriate code to read in a 64k file with random bytes, ran a SHA-256 hash on it N number of times (N being specified in the URL's query string, e.g., `.../test.php?n=100`) and print the resulting hash in hex. I chose this because it's a very simple way to run the same benchmarks with some consistent I/O and a controlled way to increase CPU usage.

See these benchmark notes for a bit more detail on the environments used.

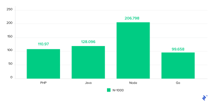
First, let's look at some low concurrency examples. Running 2000 iterations with 300 concurrent requests and only one hash per request (N=1) gives us this:



Times are the mean number of milliseconds to complete a request across all concurrent requests. Lower is better.

It's hard to draw a conclusion from just this one graph, but this to me seems that, at this volume of connection and computation, we're seeing times that more to do with the general execution of the languages themselves, much more so that the I/O. Note that the languages which are considered "scripting languages" (loose typing, dynamic interpretation) perform the slowest.

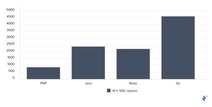
But what happens if we increase N to 1000, still with 300 concurrent requests - the same load but 100x more hash iterations (significantly more CPU load):



Times are the mean number of milliseconds to complete a request across all concurrent requests. Lower is better.

All of a sudden, Node performance drops significantly, because the CPU-intensive operations in each request are blocking each other. And interestingly enough, comparing PHP versus Java performance, PHP's performance gets much better (relative to the others) and beats Java in this test. (It's worth noting that in PHP the SHA-256 implementation is written in C and the execution path is spending a lot more time in that loop, since we're doing 1000 hash iterations now).

Now let's try 5000 concurrent connections (with N=1) - or as close to that as I could come. Unfortunately, for most of these environments, the failure rate was not insignificant. For this chart, we'll look at the total number of requests per second. *The higher the better*:



Total number of requests per second. Higher is better.

And the picture looks quite different. It’s a guess, but it looks like at high connection volume the per-connection overhead involved with spawning new processes and the additional memory associated with it in PHP+Apache seems to become a dominant factor and tanks PHP’s performance. **Clearly, Go is the winner here, followed by Java, Node and finally PHP.**

While the factors involved with your overall throughput are many and also vary widely from application to application, the more you understand about the guts of what is going on under the hood and the tradeoffs involved, the better off you’ll be.

In Summary

With all of the above, it’s pretty clear that as languages have evolved, the solutions to dealing with large-scale applications that do lots of I/O have evolved with it.

To be fair, both PHP and Java, despite the descriptions in this article, do have implementations of non-blocking I/O available for use in web applications. But these are not as common as the approaches described above, and the attendant operational overhead of maintaining servers using such approaches would need to be taken into account. Not to mention that your code must be structured in a way that works with such environments; your “normal” PHP or Java web application usually will not run without significant modifications in such an environment.

As a comparison, if we consider a few significant factors that affect performance as well as ease of use, we get this:

Language	Threads vs. Processes	Non-blocking I/O	Ease of Use
PHP	Processes	No	
Java	Threads	Available	Requires Callbacks
Node.js	Threads	Yes	Requires Callbacks
Go	Threads (Goroutines)	Yes	No Callbacks Needed

Threads are generally going to be much more memory efficient than processes, since they share the same memory space whereas processes don’t. Combining that with the factors related to non-blocking I/O, we can see that at least with the factors considered above, as we move down the list the general setup as it related to I/O improves. So if I had to pick a winner in the above contest, it would certainly be Go.

Even so, in practice, choosing an environment in which to build your application is closely connected to the familiarity your team has with said environment, and the overall productivity you can achieve with it. So it may not make sense for every team to just dive in and start developing web applications and services in Node or Go. Indeed, finding developers or the familiarity of your in-house team is often cited as the main reason to not use a different language and/or environment. That said, times have changed over the past fifteen years or so, a lot.

Hopefully the above helps paint a clearer picture of what is happening under the hood and gives you some ideas of how to deal with real-world scalability for your application. Happy inputting and outputting!

Tags

- PHP
- Java
- I/O
- Go
- Node.js

Brad Peabody

Verified Expert in Engineering

Located in Los Angeles, CA, United States

Member since February 9, 2017

About the author

With almost two decades of business software development, Brad’s led web teams, been a Linux sysadmin, and developed a storefront in Go.

Expertise

Back-end Java Go

Years of Experience

25