An Analysis of the Performance of Websockets in Various Programming Languages and Libraries

Matt Tomasetti

Computer Science
Rampo College of New Jersey
Mahwah, United States of America
mtomaset@ramapo.edu

***Abstract* — As the demand for real-time data increases, so does the use of websockets. It is crucial to consider the speed along with the reliability of a language and websocket library before implementing it in an application. This study aims to benchmark various websocket servers in order to determine which one offers the fastest round trip time of a request, as well as the reliability of the websocket server under load.**

Keywords— websocket, server, client, benchmark, echo, c, libwebsocket, c++, cpp uwebsockets, c#, cs, fleck, go, gorilla, java, java-websocket, node, nodejs, php, ratchet, python, websockets, rust, rust-websocket, connection, request, round trip, time, docker, container, virtual machine, KVM

I. Purpose

The purpose of this benchmarking exercise is to determine which programming language/library will offer the best performance in a websocket application. The websocket servers that are being tested in this benchmark are purposely simple, without extensive configuration or multithreading. This is in order to showcase which server performs the best, right out of the box. For this reason, each server is kept as close to a generic "echo server" as possible. The data being collected will provide insights on the performance of the websocket servers. The specific metrics that are of interest in this assessment are the round trip time (the time from when a client sends a request to the server, to the time the client receives a response), and the success rate of requests (the percentage of requests which successfully get a response).

It is important to note that a websocket which is critical to an application should be set-up in a more optimal environment than the one being demonstrated in this testing. Nevertheless, accounting for the worst case scenario is always imperative. A websocket that needs to be dependable may use multithreading in order to achieve better performance, and/or a load balancer to distribute connected clients. That being said, perhaps one has a websocket with a defect, whether it is due to an undetected bug or a flaw in the initial design. Contemplate what may happen to the flawed websocket application in a moment of high demand. Even if load balanced, assess the events that would follow if one of the instances of the websocket goes down. Will the remaining instances be able to shoulder the influx of incoming connections, as the abandoned clients try to reconnect? Or, will another instance go down under the increased demand, causing a domino effect to occur? Which programming language/library offers the most reliability in order to handle the awaiting clients, until one's infrastructure can spool up another instance of the downed websocket server?

Originally, the intent of this project was to determine which websocket server is best suited for some of my own personal projects, based on speed, reliability, and furthermore, code complexity. However, it has been considered that this information would be useful to other developers as well. Therefore, I have decided to publish my findings.

II. Method

As mentioned, the websocket servers are kept as close to generic echo servers as possible. That is, with the exception of the servers having a small difference. In a traditional echo server, the client sends a string to the server, and the server responds back with that exact same string. The benchmark servers, on the other hand, wait for the client to send a JSON string containing the message count signified as "c" i.e. { "c": 1 }. The server then responds back with a JSON string containing "c" along with the additional property "ts" which is the Unix timestamp of when the server receives the message i.e. { "c": 1, "ts": 780283800 }.

The original intention of the "ts" property is for the client to be able to calculate the time it takes the server to receive the client's message, as well as the time it takes the client to receive the server's response. However, this calculation turns out to be more complex due to the system times of the client machine and the server machine not being perfectly synced. Because of this, only the total round trip time will be calculated for each request.

That being said, the "ts" property remains in the response in order to give the websocket a task to complete before responding. This small operation gives the websocket an intermediate step, rather than behave as a basic echo server. As this is not a performance test of the languages themselves, it felt improper to replace this step with some arbitrary pattern matching or matrix multiplication. Because of this, it has been decided to keep the process of decoding an incoming JSON string, adding a new property, and encoding a JSON string,

despite the fact that the "ts" property is not used. This felt like a more natural task for the application as manipulating JSON objects is extremely common for websockets.

In order to measure changes to the websocket servers' response and reliability, the benchmarking client puts each server under an increasing amount of stress. This is done by performing multiple rounds of testing. Each round consists of a collection of websocket clients sending requests to the server, with the amount of clients increasing by a fixed amount each round. The number of requests per client per round is also fixed. Therefore, as the number of total clients increases each round, so does the total number of requests to the server. More specifically, 100 new clients are added each round, with each client sending 100 requests. This increases the total number of requests sent each round by 10,000. There will be a total of 100 rounds, with a cumulative 5.5 million requests. This will allow us to see how each server behaves under an increasingly demanding load.

Determining the appropriate number of clients and requests took some trial and error. Any more than 14,000 clients, and the operating system throws an "Open File Limit" error. This limit can be increased, but doing so is unnecessary for our testing. The most important factor turns out to be time, which has dictated the final configuration the most. This configuration ended up being 100 rounds of testing, adding 100 clients each round, for a total of 10,000 clients, and each client sending 100 requests per round. A total of 9 websocket servers are to be tested for this experiment. The languages and libraries are as follows:

- C / Libwebsockets
- C++ / uWebSockets
- C# / Fleck
- Go / Gorilla
- Java / Java-WebSocket
- NodeJS / uWebsSocket
- PHP / Ratchet
- Python / websockets
- Rust / rust-websocket

For the sake of readability, going forward the websocket servers will be referred to solely by the language in which they are written. It should be noted that the results for a given language is not a representation of the language as a whole, and alternate libraries for the same language may yield different results.

The benchmarking client is written in NodeJS, which is specifically chosen because of its non-blocking nature. This is a desirable trait, as all of the websocket clients should make their requests to the server at the same time, rather than one connection having to finish sending all their requests before the next connection can start.

III. Prediction

Prior to running the benchmarks, my predictions are as follows: the websocket written in C will have the fastest round trip times. Furthermore, the websocket written in Python will offer the best performance when code complexity is taken into account. Finally, PHP will perform with the slowest round trip times, in addition to being the least reliable.

IV. Set-Up

A. Server Machine

The machine running the websocket servers is a Dell PowerEdge R410. It is outfitted with dual 8-core Xeon L5630's clocked at 2.13 GHz, and has 16 GB of DDR3 RAM. For the operating system, the machine is running Ubuntu 20. In order to isolate the benchmarking process from any other programs that may be running in the background, Docker has been installed on a virtual machine maintained by KVM. The actual benchmarking VM has 2 cores allocated to it with 4 GB of RAM, again running Ubuntu. The websocket servers will run one at a time inside their own Docker container, which the images for each are available on my Docker Hub.

B. Client Machine

The client machine is running an Intel i5-8600, with 6 cores, clocked at 3.10 GHz, but can boost up to 4.30 GHz. It has 8 GB of DDR4 RAM in a dual channel configuration, and utilizes Windows 10 as its operating system.

C. Connection

There is a direct cat 5e ethernet link running from the client machine to the server machine. This is specifically done as to not DDOS my home router. However, the direct link also has the additional benefit of reducing latency between the two machines, which further increases the accuracy of the results.

D. Reasoning

This set-up is deliberately in a configuration where the client machine is more powerful than the server machine. This is to ensure that any bottleneck in the benchmarking process is due to the benchmarking servers rather than a lack of requests coming from the client.

Furthermore, a CPU with a lower clock speed is utilized in order to exacerbate any performance differences in the websocket servers. Two Raspberry Pi 4B's were originally considered for the role of both the client and server machines. However, that idea was scrapped, as an early test showed that these devices are actually too slow for the job. The clock speed on the PowerEdge R410 turns out to be perfect to adequately highlight any performance difference in the websocket servers. Yet, it is fast enough as to not be unbearable to work on, and to perform the benchmark test in an adequate amount of time.

V. Results

Each test was run 3 times per websocket server, with the results of the 3 tests being averaged together. During the benchmarking process, the client machine sends upwards of 22.5 Mb/s to the server, with the more performant websocket

servers responding with up to 29.0 Mb/s. Although, this varies based on the performance of the websocket. The CPU of the client machine does have to boost from base clock, sustaining 4.09 GHz, and maxing at 4.13 GHz. The CPU never fully hits 100% utilization, despite staying in the high 90s for the later rounds of tests. TheVM server maxes out CPU utilization on all tests, and uses roughly 400 MB of RAM, give or take 100 MB.

A. Overall

It is unexpected to see that C and Python are unable to complete the benchmark test. Python consistently makes it to round 32, and then drops all the websocket connections. The benchmarking client eventually throws a "heap out of memory" error from trying to recursively connect back to the server. It is also extremely concerning that Python's elapsed time for each round seems to increase exponentially as the number of connections/requests increases linearly.

C, on the other hand, is a bit more unpredictable, making it to between round 70-80. Again, it drops the connections until the client runs out of heap space. Multithreading these websockets, or running them on a more powerful machine would potentially improve their results. However, that point is moot, since doing so will improve the results for all websocket servers.

Another surprise is that out of all the websockets that are able to complete the benchmark, Go performs the worst. Go's performance does not just lag behind by a little, rather it takes over twice as long to complete the benchmark when compared to the next slowest websocket, which is Rust.

Something that does not come as too much of a surprise is the fastest websocket. Although, my initial prediction was that C would perform the best, it is not unexpected that NodeJS takes the crown. Node's asynchronous nature allows for greater throughput of requests coming into the server.

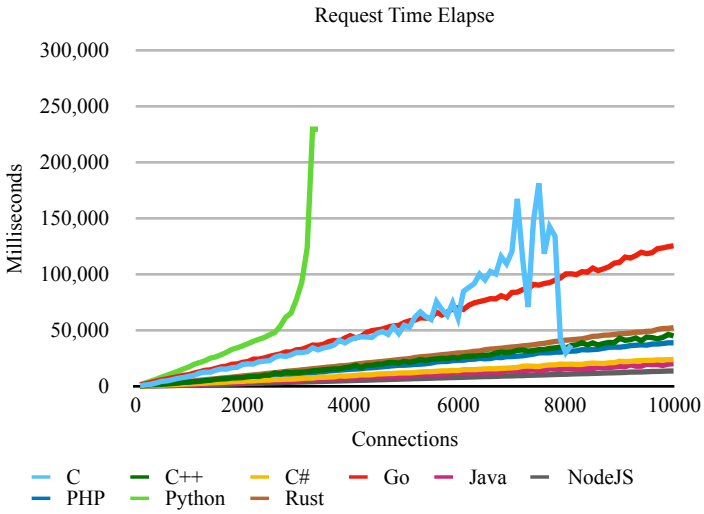

Fig. 1 Illustrates the time it takes each websocket server to respond to all requests from the benchmarking client for a given number of connections.

Java and C# follow closely behind NodeJS. With PHP, C++, and Rust performing the benchmark a little slower.

B. Success Rate

All the websockets remain 100% reliable throughout the benchmark. That is, with the exception of Python and C, which are the only two websockets that drop messages. Due to their unreliability, and inability to complete the benchmark, the results for C and Python are excluded from the rest of this section.

C. Connection Time

Despite being the slowest when it came to responding to requests, Go displays the best connection times. Connecting to the cumulative 10,000 clients in 8.8 seconds. Java, which is one of the best when it came to request times, performs the connections the slowest, connecting to the 10,000 clients in 205 seconds. C++ takes 60 seconds, with the other servers performing the connections in 10-20 seconds.

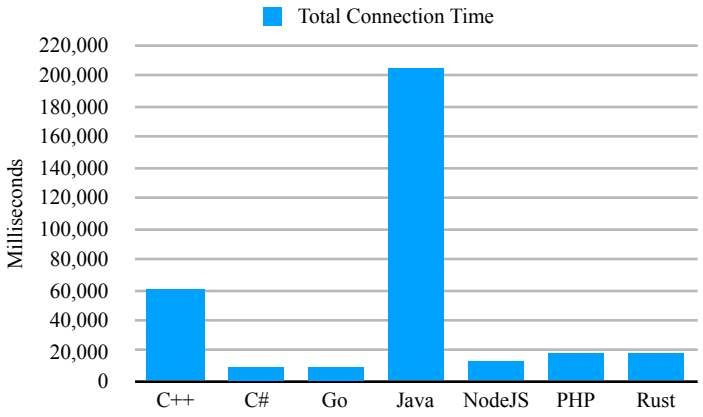

Fig. 2 Visually displays the time it takes each server to connect to the cumulative 10,000 websocket clients.

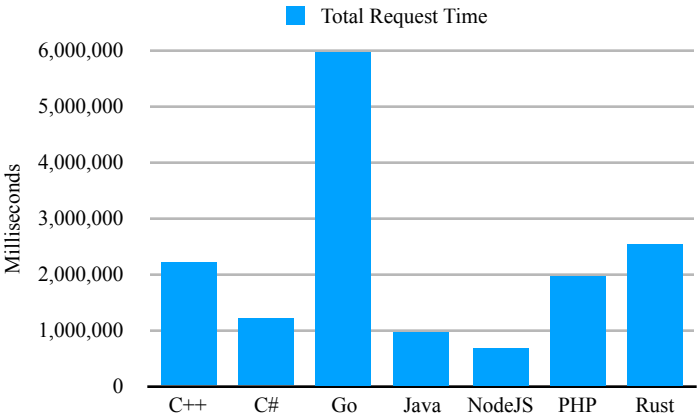

Fig. 3 Visually displays the time it takes each server to respond to the cumulative 5.5 million requests.

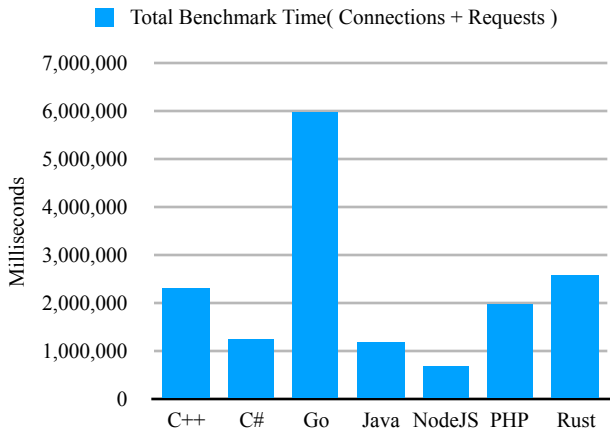

Fig. 4 Visually displays the total time each server takes to complete the entire benchmark test.

D. Request Time

The amount of time it takes for the websocket servers to respond to requests increases linearly as the total number of requests also increases linearly. Go is the biggest loser in this category, taking 100 minutes to complete the cumulative 5.5 million requests. NodeJS performs the best, taking under 12 minutes to complete all requests. Rust takes 42 minutes, C++ takes 37 minutes, PHP takes 32 minutes, C# 20 minutes, and Java 16 minutes.

E. Longest Round Trip

Another metric to be considered is the longest round trip time, this being the longest amount of time for a single round trip to complete. This is useful to see how long a websocket server could potentially leave a client waiting for a response. Obviously, Go has the longest round trip times. PHP and C++ tie for third. Java and C# tie for second, and NodeJS arises victorious having the fastest longest round trips.

VI. Analysis

While the results of the benchmark tests are not as expected, they do make sense once further understood. Let us start by looking at C, or more specifically LWS (Libwebsockets). As it turns out, LWS is in fact an extremely inefficient websocket library when it comes to performance. The entire websocket server runs in a blocking event loop on a single thread. To put it simply the server operates something like this:

1. Accept incoming message
2. Read incoming message
3. Generate response
4. Send response
5. Repeat

It does this for each incoming request, one request at a time, and all in a single thread. In fact, it was proposed earlier that multithreading the application could improve performance, only

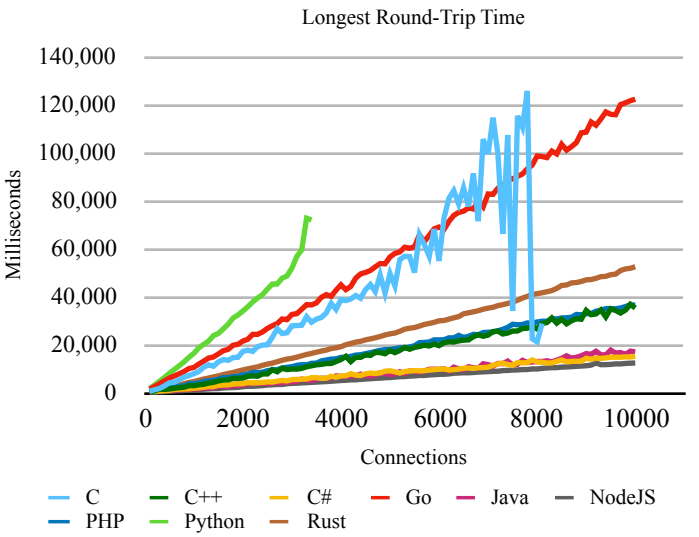

Fig. 5 Illustrates the longest amount of time it takes each websocket server to respond to a single request for a given number of connections.

to find out that this is highly discouraged. Libwebsocket's own website [2] states "Directly performing websocket actions from other threads is not allowed. Aside from the internal data being inconsistent in forked() processes, the scope of a wsi (struct websocket) can end at any time during service with the socket closing and the wsi freed." The wsi variable referenced in that quote refers to the pointer to the client connection. Plainly speaking, if one tries to multithread LWS, one could end up with incorrect data in the forked thread, or lose the connection to the client all together. Therefore, the poor performance experienced is a design decision of the LWS library. This suggests that while our tests shows poor performance for the C websocket server, there may be other C libraries which would offer better speed and reliability.

The idea of this single-threaded design also explains Go's poor performance. Go does not meet the performance expected either. However, the explanation is quite simple. Go is designed to take advantage of concurrent processing. In other words, Go achieves its renowned performance by completing tasks in parallel. Therefore, running everything from a single goroutine (a single thread) substantially hampers the performance of the websocket, as it was never designed to be utilized in such a bare-bones set-up. The good news is, unlike the C websocket, the Go server is able to use multiple goroutines, and therefore be multithreaded for better performance.

So how did C++, PHP, and Rust achieve better performance than their C and Go counterparts? To put it simply, while C and Go servers are subject to blocking code, the C++, PHP, and Rust servers are not. In other words, C and Go complete each task one at a time, in order, one after another. Meanwhile C++, PHP, and Rust can complete their tasks asynchronously, out of order, in whatever sequence will get the

job done the fastest. The advantage here is that these asynchronous servers can work on other tasks while waiting for an entirely different tasks to complete. This leads to huge performance improvements, as seen in the results of this benchmark.

So what tricks do Java, C#, and NodeJS use to improve performance even further? Not only are these servers performing their tasks asynchronously, but they also automatically spawn dozens of threads to perform them in parallel. This gave Java, C#, and NodeJS an even greater advantage than the aforementioned servers which are limited to one single thread. It should be noted that, if one desires to do so, C++, PHP, and Rust can also be multithreaded to achieve similar performance improvements. This just has to be implemented manually by the developer.

Lastly, that brings us to Python. Why does Python perform so poorly? Is it also a victim of blocking code? Actually, no. While the Python server does run on a single thread, the code is written to be asynchronous. In part, the reason Python performs so terribly is because the websocket library being used is horribly unoptimized. While reputable websocket libraries are being used for the other websocket servers, like the established Ratchet websocket for PHP and uWebsockets for NodeJS, Python is different. For the Python websocket, a generic module is used which is simply named "websockets". The documentation for this module is limited, and custom configuration is non-existent. This is most likely a module that offers the simplest of websocket functionality. Now, it was mentioned that this only partly explains the poor performance. While writing this report, it seemed unjust not to give Python a fighting chance. So, the websocket server has been rebuilt with the more trusted Autobahn library and the benchmark test has been rerun. This new server does lead to better results, though I use the word "better" loosely. With the Autobahn server, the time to complete each round increases linearly rather than exponentially, which is a promising sign. Even so, the performance is still worse than Go's websocket server. Additionally, it is still unable to finish the benchmark test even with this more optimized websocket library. It gets to round 98, and then the server drops the websocket connections, with many dropped messages throughout the benchmarking process. Nevertheless, the Python server is rebuilt one more time, this time with a library by the name of "aiohttp." At last, all 100 rounds of the benchmark are able to be completed, though not very well. Aiohttp still takes longer than Go, and becomes substantially unreliable after round 50, dropping anywhere from 30-50% of the messages. It can only be concluded that the reason for this dreadful performance is Python itself. Python, which is interpreted at run time rather than compiled, suffers from slow execution time. Even when compared to other interpreted languages, like PHP, Python's performance still lags behind [1].

VII. How To Replicate

The source code for each websocket server as well as the client can be found on GitHub at: https://github.com/matttomasetti. In addition, ready-made docker images can be found on Docker Hub at: https://hub.docker.com/u/mtomasetti. The images will automatically start the websocket server listening on port 8080. The image for the websocket client will automatically start the benchmarking client, looking for a server on localhost. However the default settings can be altered through environment variables. Further details are listed in the README files in each repository. The results of each benchmark test can be viewed in greater detail at https://matttomasetti.com.

VIII. What Can Be Improved

If one chooses to revisit this experiment, it would be beneficial to perform it with a greater number of websocket variations. These variations would include websockets written in more languages as well as multiple libraries per language. This will not only highlight which library is the best, but also potentially which language offers the best performance for websocket applications. It would also be a good idea to increase the number of clients and rounds of the benchmark test in order to see if there is a breaking point for additional websocket servers other than just the C and Python implementations. Lastly, it could be interesting to multithread the websockets servers (that support it) as to give a level playing field between the servers that do so automatically and the ones that do not.

If one wants to go above and beyond they may not only multithread the servers, but also load up multiple instances of the websockets with a load balancer in front of them. This will be useful to see just how much performance could be achieved out of each websocket when it is given a more optimal environment, rather than thrown in a worse case scenario.

IX. Conclusion

As we can see, all my predictions turned out to be entirely incorrect, but this is not a bad thing. Ending up with data that goes against one's initial expectations proves that there is knowledge to be gained. From the information that has been uncovered in this report, I propose the following 4 guidelines when selecting a websocket library:

1. Ensure the websocket library is asynchronous. This may also be expressed as "non-blocking".
2. Ensure the library allows the websocket to be multithreaded, either done automatically, or with additional configuration by the developer.
3. Greater performance may be achieved by using a compiled language over an interpreted one.
4. Do not use Python.

All in all, the winner here is clearly NodeJS. From its amazing performance to its code complexity (or rather lack of complexity), NodeJS is an optimal choice for a websocket

project. I thought a similar complement would be applicable to Python. Unfortunately, after this test, it is clear that Python based websockets should be avoided at all costs. For a more business oriented application, one cannot go wrong with the enterprise favorites of Java or C#. That being said, if one is to take a single piece of knowledge away from this study, it would be to always use an asynchronous websocket.

REFERENCE

[1] A. Burets, "Python vs PHP: Main Differences and Comparison: SCAND Blog," *SCAND*, 28-Jun-2019. [Online]. Available: https://scand.com/company/blog/python-vs-php-for-web-development/. [Accessed: 26-Jan-2021].

[2] "Notes about coding with lws ," *libwebsockets*. [Online]. Available: https://libwebsockets.org/lws-api-doc-master/html/md_README_8coding.html. [Accessed: 26-Jan-2021].

Additional Graphs

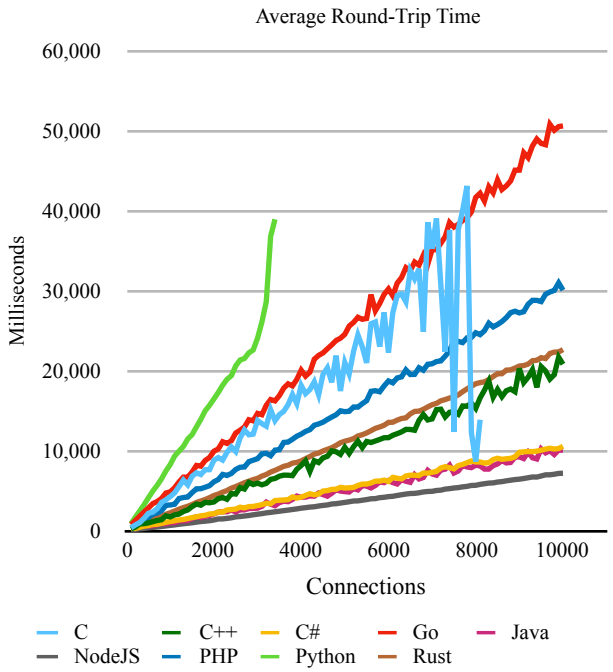

Fig. 6 Illustrates the average amount of time it takes each websocket server to respond to a single request for a given number of connections.

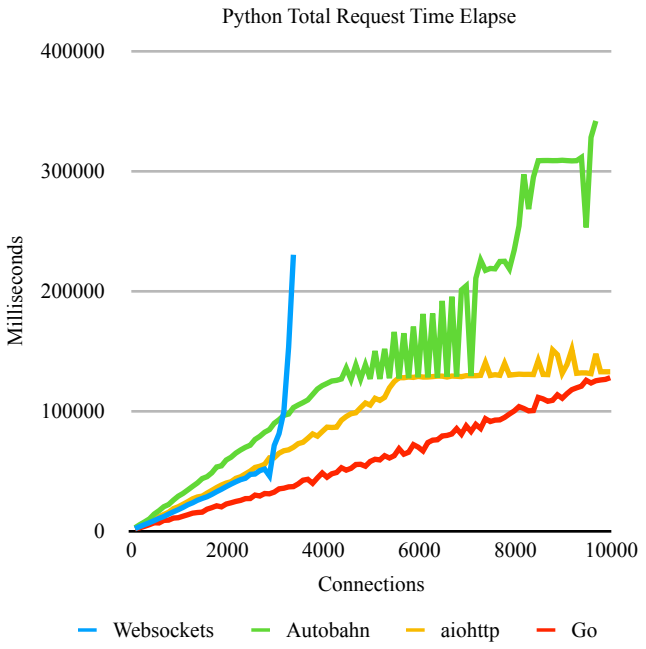

Fig. 5 Illustrates the total amount of time it takes each websocket server to complete all requests for a given number of connections

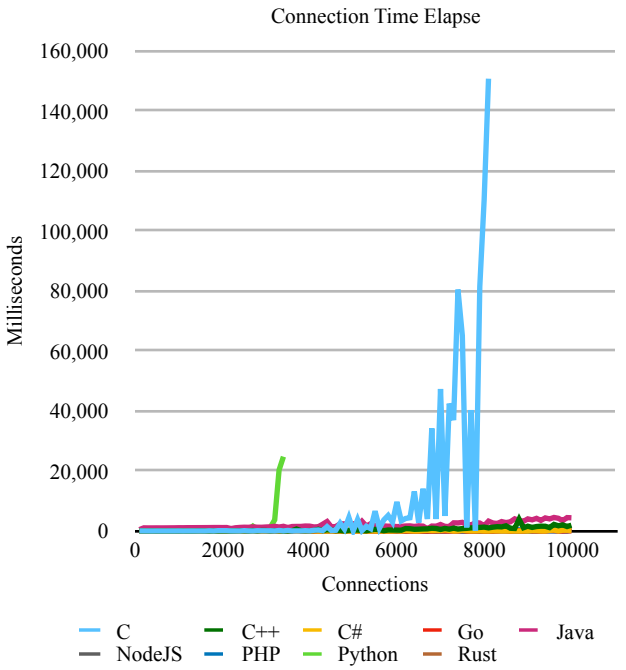

Fig. 7 Illustrates the amount of time it takes for each websocket server to add 100 new connections with a given number of already existing connections

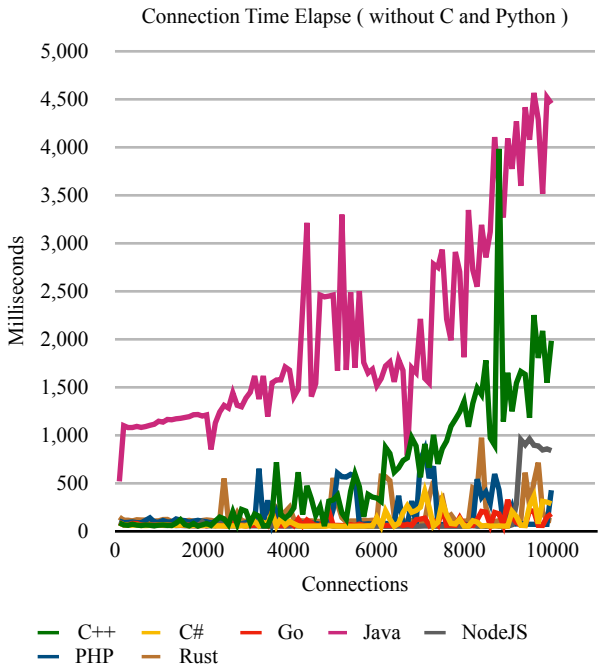

Fig. 8 Illustrates the amount of time it takes for each websocket server to add 100 new connections with a given number of already existing connections (same a Figure 7, just without C and Python).