BENCHMARKING PYTHON WEBSOCKET FRAMEWORKS

MELIH CIHAN KIZILTOPRAK

melihcihankiziltoprak@gmail.com
+90 (552) 789 12 01

ABSTRACT

This work critically examines the performance and suitability of Python WebSocket libraries and frameworks for a specific project (real-time data processing and low-latency communication), drawing insights from current literature and empirical comparisons. Initially, contrasting viewpoints from Matt Tomasetti and Ville Kärkkäinen present a dichotomy regarding Python's efficacy, particularly with uWebSockets-based libraries. The study explores alternative libraries like Socketify, revealing its potential but highlighting critical reliability issues. Subsequently, the focus shifts to evaluating two Python frameworks, aiohttp and fastapi, where concerns about reliability and community support prompt a comprehensive analysis. Empirical comparisons showcase aiohttp's consistent performance and scalability, positioning it as the preferred choice over fastapi. The research methodology involves benchmarking on various servers under different scenarios, providing valuable insights into speed, CPU utilization, reliability, scalability, and overall behavior.

CURRENT LITERATURE

Based on a research published by Matt Tomasetti (https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3778525), across several programming languages (C / Libwebsockets, C++ / uWebSockets, C# / Fleck, Go / Gorilla, Java / Java-WebSocket, NodeJS / uWebsSocket, PHP / Ratchet, Python / websockets, Rust / rust-websocket), Python (with websockets) underperforms drastically while C++, NodeJS and Go excells in majority of tests. Therefore, Tomasetti suggests not using Python at all!

In contrary to Tomasetti, Ville Kärkkäinen in his work (https://ville-karkkainen.medium.com/python-is-slow-wait-its-actually-fast-6d2e49621b1), demonstrates that when using more efficient libraries than "websockets", Python can compete and can actually beat other programming languages! In his research Kärkkäinen comes to the conclusion that uWebSockets library (written in C/C++) outperforms all, when used in Python.

In conclusion, based on the findings from the current literature on benchmarking websocket library/framework performance, supported both by the findings of Tomasetti and Kärkkäinen, one could infer that when used in Python, uWebSockets based libraries outperforms the rest. Therefore, in order to maximize performance, one should prefer uWebSockets based libraries in Python such as: **uws, uWebSockets, Socketify.**

uws & uWebSockets

After checking out the current literature, by common sense I have checked and tried to utilize both uws and uWebSockets. However, unfortunately I have realized that they are no longer supported and even uWebSockets' source codes have become private (by source code I mean the Python bindings of the uWebSockets C++ library: https://github.com/uNetworking/uWebSockets/discussions/1202). So, from thereon, I have tried to seek an alternative library that utilizes uWebSockets. Finally, I have found Socketify.

Socketify

Based on my observations on Socketify, it actually has the infrastructure and potential to outperform the libraries of my preference (aiohttp and fastapi). Even, I have actually observed that it can reach a much better throughput than those frameworks. However, there is a much more important aspect of a library: reliability. Unfortunately, Socketify lacks that a lot. It has a lot of unresolved critical bugs (https://github.com/cirospaciari/socketify.py/issues). Especially I have consistently observed that after websocket connections end, Socketify segfaults (corresponding bug can be seen here: https://github.com/cirospaciari/socketify.py/issues/141).

In short, one must follow the updates on Socketify closely (alongside the other emerging libraries based on uWebSockets). However, in the short term, it would be a bad practice to utilize such an unreliable library.

aiohttp & fastapi

At present, concerns about unreliability, lack of support, and the absence of a community discourage us from using a uWebSockets-based library. Consequently, we are now evaluating two competent Python frameworks: aiohttp and fastapi. aiohttp boasts a strong and active community that ensures regular updates and robust support. On the other hand, fastapi, known for its high performance, has been gaining popularity with its engaged user base. As we evaluate these options, the strength of their communities becomes a crucial factor in our decision-making, ensuring ongoing support and development for our projects.

HARDWARE & OS

For this project I have used a MacBook Pro 2020 with processor: “1,4 GHz Quad-Core Intel Core i5” and memory: “8 GB 2133 MHz LPDDR3” with MacOS Sonoma 14.3. ulimit -n is set to 1000000.

ON METHOD

As for method, I have implemented 2 servers for each framework. One is random_float_server and the other is signal_server. Moreover, I have realized 3 benchmarks on these servers: One Way Messaging Latency, Input/Output Handling Efficiency, Two Way Messaging Latency (RTT).

random_float_server: Basically generates a random float then sends it as a string to the client.

signal_server: This is a bit more sophisticated than random_float_server. After connection is established, clients start receiving random floats (as in random_float_server) then, with a 20% probability they send a BUY order and with a 20% probability they send a SELL order. Finally, Server recieves these orders and answers “Order FILLED” or “Order UNFILLED”.

latency_benchmark: Connects to random_float_server. Tracks how long does it take to receive a message from the server. A performance indicator mainly for “low-latency communication”.

io_benchmark: Connects to random_float_server. Tracks how long does it take to receive a message from the server and write it to a file. A performance indicator both for “real-time data processing” and “low-latency communication”.

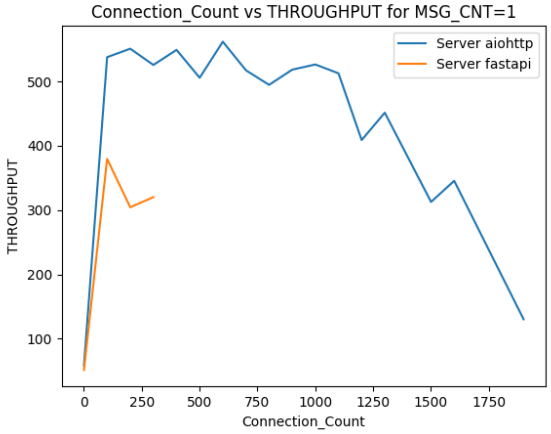
signal_benchmark: Connects to signal_server. Assesses the performance of the two-way client-server communication. Performs above-mentioned actions. A performance indicator both for “real-time data processing” and “low-latency communication”.

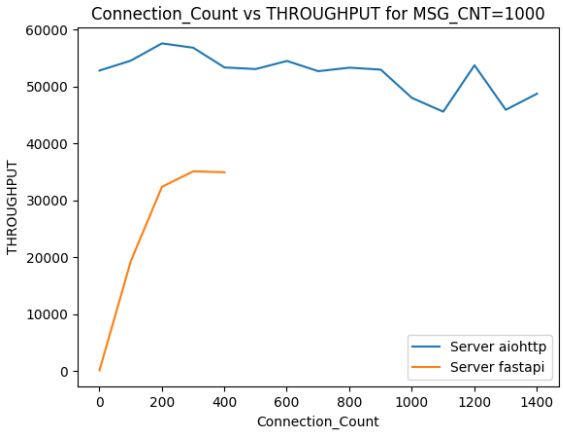
MSG_CNT: Message count variable. With this variable, I have observed the performances and the behaviors of the servers under different loads.

MAX_CONN: Maxiumum connection variable. Iterating from 1 to MAX_CONN, I have observed the performances and the behaviors of the servers with various number of connections.

By simply utilizing only 3 cases, I believe I have gathered enough information to make a decision. From these cases one can infer the CPU bounded cases (latency_benchmark and signal_benchmark), IO bounded cases (io_benchmark), one way communication (latency_benchmark and io_benchmark), and two-way communication (signal_benchmark). Therefore, by only these cases, one can infer the performance on speed, CPU utilization, reliability, scalability and behavior in various circumstances and objectives (such as real-time data processing and low-latency communication).

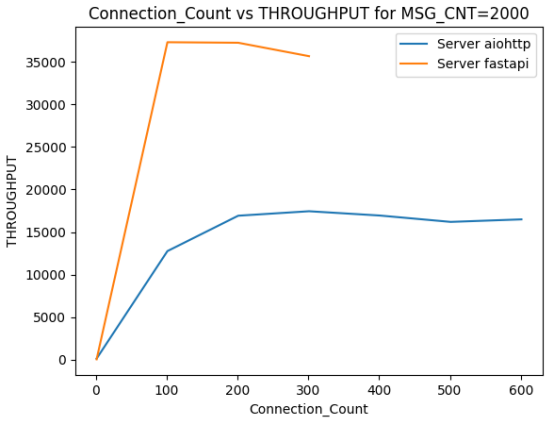
EMPIRICAL COMPARISON OF AIOHTTP AND FASTAPI

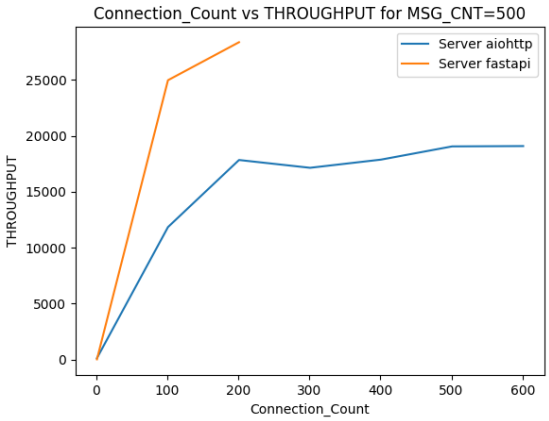




These two figures above (outputs of latency_benchmark) are a great summary for my findings through my numerous observations for one-way messaging throughput. **aiohttp** consistently outperforms **fastapi** while staying reliable and scalable. In each case wheter it is latency_benchmark, io_benchmark, or signal_benchmark, **fastapi** always becomes erroneous after a number of concurrent connections while aiohttp remains functioning and continues to work with even much more connections.

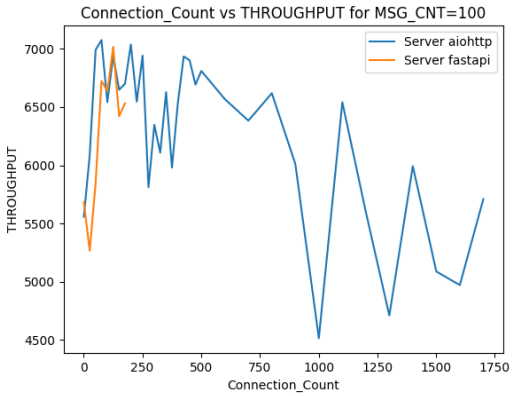
However, results start to become more interesting and rather unpredictable when checking the other cases such as these two outputs of signal_benchmark below:

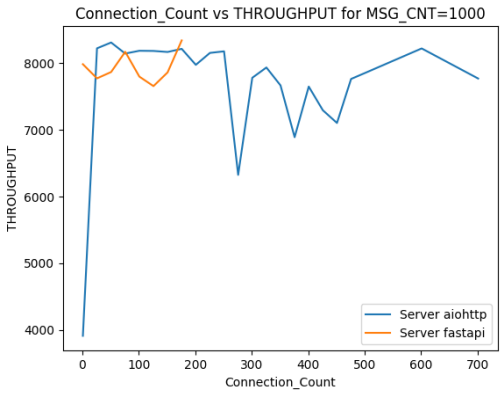




In these figures above, we see that fastapi outperforms aiohttp to some extent. However, it is still underperforming by means of scalability by becoming erroneous and losing functionality after a certain number of concurrent connection.

Finally, in io_benchmark results below, there is a tie by means of throughput:





However, fastapi still lacks scalability in comparison to aiohttp who is preserving its performance as the concurrent number of connections increases.

FINAL REMARKS

In conclusion, my exploration of WebSocket libraries and Python frameworks led me to reconsider uWebSockets-based options due to support issues. While Socketify, a potential alternative, demonstrated impressive throughput, its lack of reliability rendered it unsuitable for my considerations. Focusing on Python frameworks, **aiohttp** proved to be both reliable and high-performing indicating that it is a brilliant fit for real-time data processing and low-latency communication applications. Although fastapi exhibited strengths, it faced drastic scalability challenges. **My preference leans strongly towards aiohttp due to its high-performance, consistency, scalability, robust community support, and reliability.**

APPENDIX: HOW TO RUN THE CODES?

For aiohttp (io & latency):

- Open a terminal
- Go to the directory: “aiohttp_server”
- Run Python3 random_float_server.py (pip3 install any required libraries)
- Open another terminal
- Go to the directory: “aiohttp_client”
- Run Python3 latency_benchmark.py or io_benchmark.py (pip3 install any required libraries)

For aiohttp (signal):

- Open a terminal
- Go to the directory: “aiohttp_server”
- Run Python3 signal_server.py (pip3 install any required libraries)
- Open another terminal
- Go to the directory: “aiohttp_client”
- Run Python3 signal_benchmark.py (pip3 install any required libraries)

For fastapi (io & latency):

- Open a terminal
- Go to the directory: “fastapi _server”
- Run python3 -m uvicorn random_float_server:app --host 0.0.0.0 --port 5001 --reload
- Open another terminal
- Go to the directory: “aiohttp_client”
- Run Python3 latency_benchmark.py or io_benchmark.py (pip3 install any required libraries)

For fastapi (signal):

- Open a terminal
- Go to the directory: “fastapi _server”
- Run python3 -m uvicorn signal_server:app --host 0.0.0.0 --port 5001 --reload
- Open another terminal
- Go to the directory: “aiohttp_client”
- Run Python3 signal_benchmark.py (pip3 install any required libraries)

For comparison charts (fastapi vs aiohttp)(precondition: programs above have been run):

- Open a terminal
- Go to the directory: “aiohttp_client”
- Run python3 wrangler.py