Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Comparative Performance Benchmarking of WebSocket Libraries on Node.js and Golang

Louis Fernando[1)], Mychael Maoeretz Engel[)*]
[1,2)] Informatika, Universitas Ciputra, Surabaya, Indonesia
[1)]lfernando@student.ciputra.ac.id, [2)]mychael.engel@ciputra.ac.id

Submitted :Aug 22, 2025 | **Accepted** : Sep 18, 2025 | **Published** : Oct 2, 2025

Abstract: The demand for responsive real-time web applications continues to grow, making the selection of backend technology and WebSocket libraries a crucial factor in determining performance. Node.js and Golang are popular platforms for real-time applications. However, the WebSocket library within them offers a trade-off between features and efficiency, the impact of which has not been comprehensively measured. This research aims to fill this gap by conducting a quantitative performance analysis to compare the efficiency and scalability of four WebSocket libraries: ws and socket.io on Node.js, and gorilla/websocket and coder/websocket on Golang. This research uses a benchmarking experimental method with client load simulations that gradually increase from 100 to 1000 concurrent clients. The experiment was conducted through two scenarios, namely the Echo Test and Broadcast Test. In the Echo Test, the performance metrics measured were Connection Time, Round Trip Time (RTT), and Throughput. Meanwhile, in the Broadcast Test, the performance metric measured was Broadcast Latency. The results from the Echo Test show a significant performance disparity. At a peak load of 1000 clients, socket.io achieved a throughput of only 27,152 messages/second, whereas the lightweight libraries (ws, gorilla/websocket, and coder/websocket) all achieved over 44,000 messages/second. In the Broadcast Test with a high load, the latency difference between the four libraries became insignificant. Therefore, for applications prioritizing raw performance in point-to-point communication, certain WebSocket libraries such as ws, gorilla/websocket, and coder/websocket are more suitable for future development.

Keywords: Benchmark, Golang, Node.js, Performance Analysis, WebSocket

INTRODUCTION

In the digital age, many web applications have implemented dynamic and real-time interactions, driven by an increase in demand for improved user experience and operational efficiency, particularly in high-stakes industries such as financial technology (Fintech) for high-frequency trading, interactive online gaming, and large-scale Internet of Things (IoT) data monitoring (Sekar, 2025). Various modern applications, such as collaboration platforms, online gaming, and similar applications, heavily rely on the ability to perform data exchanges within the application instantly and continuously, thereby enhancing dynamic interaction (Gote, 2024). With the demand for interactive and immersive user experiences, there is a need for a communication architecture that can facilitate this. Consequently, the traditional request-response model faces significant challenges in meeting the low-latency and resource-efficiency demands of modern real-time applications (Murley, Ma, Mason, Bailey, & Kharraz, 2021).

The HTTP protocol has become one of the most used forms of communication in web applications. There are similar solutions to handle real-time needs, such as HTTP polling, where the client sends requests continuously. However, this results in excessive network traffic and resource waste (Murley et al., 2021). Furthermore, the request process carried out through HTTP has a significant header overhead, causing the server to continually send data to the client even when the requested data is not yet available, resulting in high latency and wasteful bandwidth usage (Murley et al., 2021). WebSocket can be an alternative solution to HTTP, as it provides real-time communication that HTTP cannot optimally provide. WebSocket features a full-duplex communication format that enables the client and server to exchange data directly, unlike HTTP, which requires periodic requests, thereby reducing the overhead and latency inherent in the HTTP model (Murley et al., 2021). With the ability to maintain a persistent connection, WebSocket improves bandwidth efficiency and supports continuous data exchange with minimal overhead (Puranik, Feiock, & Hill, 2013). This efficiency has been empirically validated in various real-time scenarios; for instance, in smart home applications, WebSocket has been shown to be superior to traditional

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

polling methods for real-time monitoring (Soewito et al., 2019). Furthermore, the effectiveness of WebSocket for continuous and real-time data exchange has been demonstrated in various modern remote laboratory architectures (Bisták, 2020; Bistak et al., 2024).

However, the selection of the backend for the WebSocket server and the optimal WebSocket library can affect overall performance, because each library offers a trade-off between features and performance. In this research, Node.js and Golang were selected as the main backends, because both are recognized for their advantages in terms of scalability, processing efficiency, and widespread adoption in the modern technology industry. Node.js has an event-driven architecture and non-blocking I/O that can handle thousands of connections simultaneously (Huang, 2020; Luo, Zhou, Zheng, & Pan, 2024). Meanwhile, Golang has concurrency features through goroutines, which can run multiple tasks simultaneously and also use memory efficiently (Suwarno & Putri Yulandi, 2023). In Node.js, this research will compare the ws library, a minimalist library with high performance and have similar implementation to the WebSocket standard, with socket.io, a library that provides many features such as fallback to HTTP long-polling but has the potential to add overhead to each data packet. Meanwhile, in Golang, this research will compare gorilla/websocket, a library that has become an industry standard due to its proven implementation and widespread adoption, with coder/websocket, a library designed with a focus on a minimalist API and high performance.

The significance of a head-to-head comparison lies in this very trade-off: a developer might choose a high-performance backend like Golang for a fintech application but inadvertently select a library with high overhead, thereby negating the platform's performance advantages. This creates a crucial knowledge gap, as a lack of direct comparative data forces developers into making decisions based on popularity rather than empirical evidence. The need for precise and replicable performance testing is more relevant than ever, especially in complex modern environments like hybrid clouds where performance can be affected by numerous variables (Ramu, 2023). Therefore, controlled benchmarks like this research are essential to provide the clear, specific data needed to bridge the gap between architectural theory and real-world implementation.

Although numerous research has compared the performance of WebSockets on various backends in general, research specifically benchmarking two different WebSocket libraries on each backend, namely Node.js and Golang, simultaneously has not yet been explored. This is a significant research gap because each library has a unique architecture and performance characteristics, whose impact on crucial metrics such as Round Trip Time (RTT), Throughput, Connection Time, and Broadcast Latency has not been comprehensively evaluated in Echo and Broadcast testing scenarios under high load conditions. Therefore, this research aims to address this gap by conducting an in-depth performance analysis to quantitatively compare the efficiency and scalability of the ws and socket.io libraries on Node.js with gorilla/websocket and coder/websocket on Golang, based on empirical data obtained from the designed testing scenarios.

LITERATURE REVIEW

Foundational studies consistently demonstrate that WebSocket provides a more efficient solution for continuous, bidirectional data exchange by minimizing latency and overhead, a conclusion supported by benchmarks in diverse fields such as IoT and healthcare (Friendly, Padly Sembiring, Faza, & Luckyhasnita, 2022; Kirilov, 2024). As the protocol has matured, its application has expanded into more demanding, latency-sensitive domains. Recent research explores WebSocket's role in modern architectures like IoT systems over 5G networks (Mitrovic, Eordevic, Veljkovic, & Dankovic, 2021) and as a reliable communication channel in mobile edge computing (Abdelfattah et al., 2020). Furthermore, in the pursuit of ultra-low latency, comparisons are now being drawn between WebSocket and emerging technologies like WebRTC, particularly for time-critical video and data streaming applications (Ariffin, Hamdan, & Kamarulzaman, 2023; Chodorek & Chodorek, 2025).

However, these recent studies often treat the WebSocket implementation as a monolithic entity, focusing on the protocol's application rather than the performance variability of its underlying libraries. This overlooks a crucial factor, as prior benchmarks have definitively shown that the choice of library within a single ecosystem can lead to significant performance disparities. For instance, research by Hansson (2020) found that a plain WebSocket implementation in JavaScript was up to 3.7 times faster than the feature-rich socket.io library. Similarly, Tomasetti (2021) concluded that different libraries within the same programming language yield highly varied performance results, a finding that underscores the importance of library-level analysis. Research by Alexeev et al. (2019) highlights that intelligent load balancing is necessary to handle different types of client connections effectively, confirming that performance under high concurrency is a significant research area.

This reveals a distinct gap in the literature. While some studies have benchmarked libraries within one platform (JavaScript) and others have compared backend platforms (Node.js vs. Golang) at a high level (Suwarno & Putri Yulandi, 2023), a comprehensive, head-to-head benchmark that assesses two popular and architecturally different libraries on each of these leading backend platforms is still unexplored. This research directly addresses this gap. By systematically evaluating ws and socket.io on Node.js, also gorilla/websocket and coder/websocket on Golang, this research moves beyond a general protocol comparison to provide granular, actionable insights into

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

how specific library choices impact performance, thus offering a more practical and nuanced guide for developers building modern real-time systems.

METHOD

This research employs a quantitative experimental method with a performance benchmarking approach to analyze and compare the performance of four WebSocket libraries, utilizing two libraries each on Node.js and Golang backends. This method was chosen because it enables the objective and controlled measurement of performance metrics in a replicable environment, and it is consistent with the methodology used in similar benchmarking researches (Hansson, 2020; Tomasetti, 2021). This experimental method aims to explicitly measure the impact of library and backend platform selection (independent variables) on performance metrics (dependent variables) under various workloads through two primary testing scenarios: the Echo Test and the Broadcast Test.

Experimental Architecture Design and Research Flow

The experimental architecture is designed to simulate the interaction between many clients and a single server endpoint. A Node.js-based Client Simulator application was developed to generate connection and message loads programmatically. These clients connect to a WebSocket Server implemented alternately using four different configurations (two for Node.js and two for Golang). During the testing, performance data from the client and server sides is automatically recorded into separate CSV files for analysis. The general architecture of this experiment flow is illustrated in Figure 1.

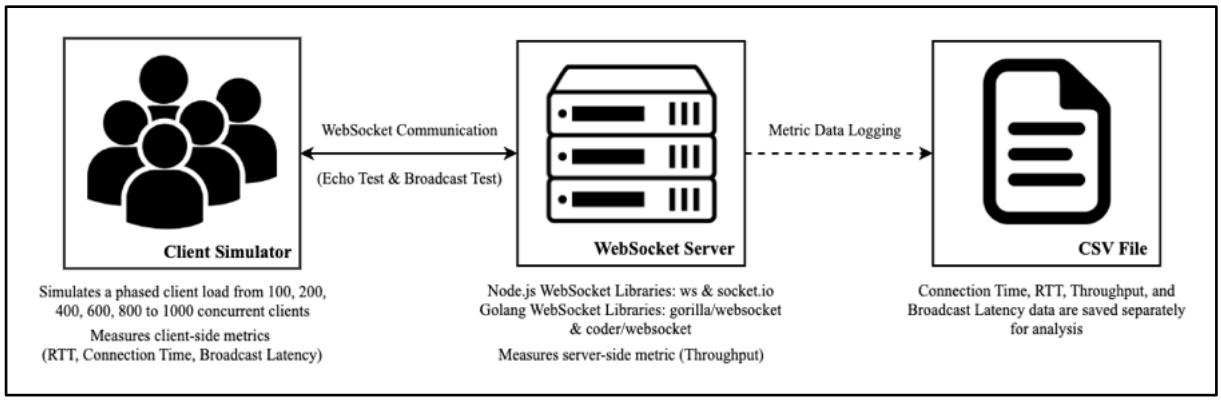

Fig. 1 WebSocket Performance Experimental Architecture

Figure 1 illustrates the high-level architecture of the testing environment. It consists of three main components: a Client Simulator responsible for generating phased client loads and measuring client-side metrics; a WebSocket Server representing the system under test, which runs one of the four library configurations; and a CSV File for storing the collected performance data. The components interact through WebSocket communication, with a separate data logging process for recording the metrics.

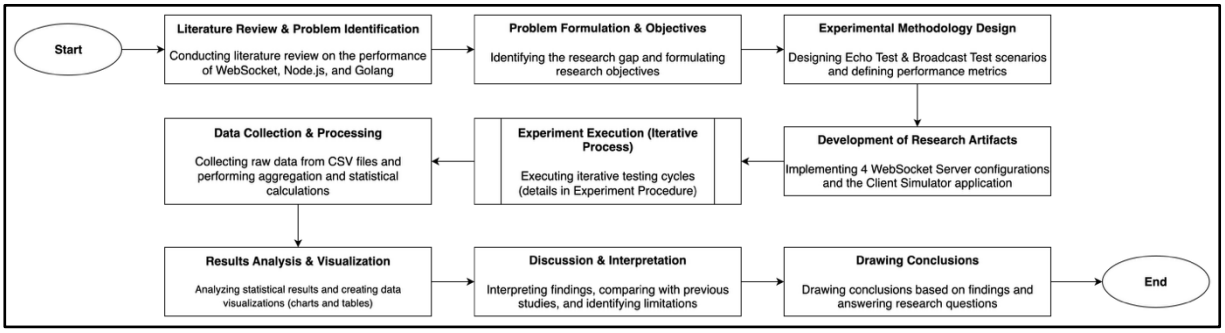

Fig. 2 Comprehensive Research Flow Diagram

Figure 2 depicts the chronological flow of the entire research methodology. The process begins with the foundational stages of literature review and problem formulation, followed by the design of the experimental methodology and the development of research artifacts. It then proceeds to the technical execution of the iterative testing cycles, data collection, and processing. The flow concludes with the final stages of results analysis, discussion, and drawing conclusions to answer the research questions.

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

Testing Environment and Configuration

To ensure consistent and comparable results, all experiments were run on the same hardware environment. It is important to note that the tests were conducted in a controlled localhost environment to isolate performance to the libraries themselves, minimizing network variables. Consequently, the results represent a best-case scenario and may differ significantly under real-world Wide Area Network (WAN) conditions, which involve unpredictable latency and packet loss. The hardware and software configurations used in this research are listed in Table 1.

Table 1. Environmental Specifications and Test Configuration

Component	Specification
Hardware	Apple M4 Pro (CPU 14-core, GPU 20-core), 24GB RAM, 1TB SSD
Operation System	macOS Sequoia (15.3.1)
Backend	Node.js (v20.19.1), Golang (v1.23.4)
Network	Localhost
Node.js Libraries	ws (v8.18.3), socket.io (v4.8.1)
Golang Libraries	gorilla/websocket (v1.5.3), coder/websocket (v1.8.13)
Client Simulator	Node.js (websocket v1.0.35, socket.io-client v4.8.1)
Data Analysis	Python 3 with Pandas, Matplotlib, and Seaborn
Data Output	CSV File (Comma-Separated Values)

Experiment Procedure

Testing procedures for each WebSocket server configuration are conducted systematically and in chronological order to ensure data comparability. Client load stages are selected, starting from 100, 200, 400, 600, 800, and up to a peak of 1000 clients, specifically designed to observe scalability trends comprehensively. The low-load stages (100-400 clients) aim to establish a baseline performance, while the high-load stages (600-1000 clients) are used to evaluate how each library handles pressure and identify potential performance bottlenecks.

The testing steps begin with preparation, where one of the four WebSocket server implementations is set up and run on a specified port. Next, the Client Simulator is initialized with the appropriate parameters, such as the server's name and the total number of clients to be simulated to start the test. The load simulation process is then run in stages, starting with 100 concurrent clients and then increasing to 200, 400, 600, and 800 clients, until reaching a peak of 1000 clients. At each load stage, the message delivery simulation lasts for 60 seconds to collect representative data. During this period, one of the two main test scenarios, either the Echo Test or the Broadcast Test, is run. The Echo Test is a scenario that measures point-to-point communication performance, where clients independently send messages to the server continuously. The server then immediately replies (echoes) the message back only to the sending client. Meanwhile, the Broadcast Test is a scenario that simulates the use case of a collaborative application where a special client acts as the sender, sending messages every 3 seconds. The server then broadcasts the message to all other connected clients (receivers). During the simulation process, all performance metrics are recorded in real-time and written to a CSV file immediately.

Each test scenario at each load level was run three times. Finally, after one testing cycle for one WebSocket server configuration is complete, the server is shut down, and the entire procedure is repeated from the beginning for the following WebSocket server configuration, continuing until all four server implementations have been thoroughly tested in both the Echo Test and Broadcast Test scenarios.

Validity and Reliability

To ensure the validity and reliability of the findings, a rigorous testing protocol was implemented as a core strength of this methodology. Each test scenario at every client load level was executed three independent times. This repetition is a standard practice in performance benchmarking designed to mitigate the risk of anomalous results caused by transient system fluctuations (Tomasetti, 2021). The final data presented in the Results section is the arithmetic mean of these three runs. This approach ensures that the findings are stable, representative, and provides a higher degree of confidence in the conclusions drawn.

Measurement, Testing, and Evaluation of Results

To fulfill the research objectives, raw data collected from CSV files will be measured, tested, and evaluated using a structured quantitative approach. Measurements are performed programmatically, with several

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

performance metrics being measured: Connection Time, which is the time required for the client to successfully establish a connection with the WebSocket server; Round Trip Time (RTT), which is the total time from the client sending a message to the WebSocket server until the message is received back in the Echo Test; Throughput, which is the total number of messages successfully processed and sent back per second in the Echo Test; and Broadcast Latency, which is the time required for a message to be received by the receiving client since it was sent by the sending client in the Broadcast Test. Therefore, for the Echo Test, the performance metrics measured are Connection Time, Round Trip Time (RTT), and Throughput. For the Broadcast Test, the performance metric measured is Broadcast Latency.

The measurement results from each test scenario are stored in separate CSV files. To facilitate recording and analysis of the measured metrics, server configuration, and client load levels tested, the file naming structure is designed systematically. An example of the file naming convention is rtt_nodejs-ws_100clients.csv, where the file contains Round Trip Time (RTT) data for a Node.js server using the ws library in a test with 100 concurrent clients. As a result, each performance metric data point can be easily identified and accessed for comparison analysis with other WebSocket libraries.

Testing and evaluation of the results were conducted by processing the data stored in the CSV file using a Python script with the Pandas library for statistical analysis and Matplotlib and Seaborn for data visualization. In the initial stage, data aggregation was performed, where thousands of data obtained for each test scenario were processed to calculate key descriptive statistics such as mean, median, and standard deviation. Next, a comparative analysis is performed by directly comparing the mean values of each metric for the four server configurations at each client load level. As an evaluation step, the comparison results are visualized in the form of a line chart to clearly present performance trends. This aims to assist in interpreting scalability and identifying bottlenecks in each WebSocket library. Therefore, the conclusions drawn are based on valid statistical analysis and easy-to-understand data visualization.

RESULT

Based on the tests that have been conducted, the quantitative data obtained from the performance tests are presented. The results from each test scenario were processed to calculate the average values of the relevant metrics. The data is then visualized using a line chart to present the performance comparison trend as the concurrent client load increases.

Echo Test Results

The Echo Test is designed to measure the basic communication performance of one client to one server. The metrics measured and presented in this test are Connection Time, Round Trip Time (RTT), and Throughput.

The average Connection Time measurement results are presented in Figure 3. The graph shows that the connection time for all libraries increases as the number of clients increases. The socket.io library on Node.js has the highest average connection time, at 185.96 ms, when running at a load of 1,000 clients. Meanwhile, other WebSocket libraries such as ws, gorilla/websocket, and coder/websocket have relatively similar values and are lower, with average connection times below 50 ms at peak load.

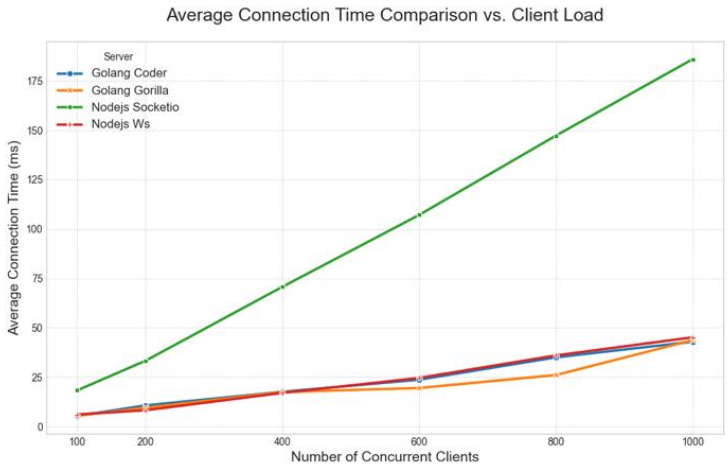

Fig. 3 Comparison of Average Connection Time vs. Client Load

For the Round Trip Time (RTT) metric, the increasing trend is illustrated in Figure 4. All libraries show a linear increase in average RTT with the addition of client load. socket.io again has the highest RTT value with an

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

average of 31.23 ms at 1,000 clients. The ws, gorilla/websocket, and coder/websocket libraries show very similar values, with average RTT values around 19 ms at the same load level.

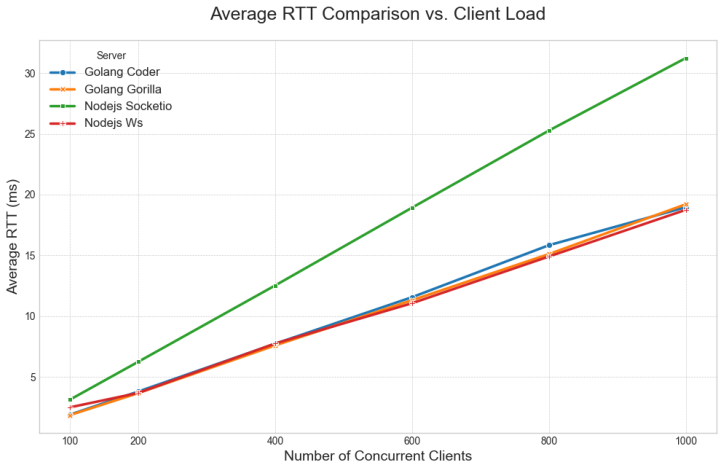

Fig. 4 Comparison of Average Round Trip Time (RTT) vs. Client Load

A comparison of throughput performance is shown in Figure 5. The socket.io library produced the lowest throughput among all configurations, with an average value of approximately 27,152 messages/second at a load of 1,000 clients. In contrast, the other three libraries showed much higher throughput. The gorilla/websocket library achieves the highest throughput at low to medium loads, while at peak loads, all three libraries (ws, gorilla/websocket, and coder/websocket) exhibit convergent values above 44,000 messages/second.

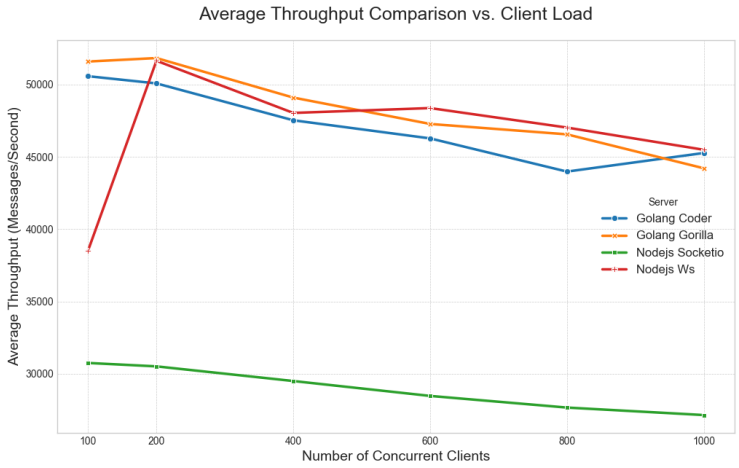

Fig. 5 Comparison of Average Throughput vs. Client Load

Broadcast Test Results

Broadcast Test is designed to measure the efficiency of a server in distributing messages to multiple clients. The main metric observed is Broadcast Latency.

The average Broadcast Latency test results are illustrated in Figure 6. The graph shows that all libraries experience an increase in latency as the number of receiving clients increases. At the lowest load level (100 clients), socket.io shows the lowest latency at 5.12 ms, while ws shows the highest latency at 7.18 ms. However, at peak load (1,000 clients), the latency values of the four libraries became very close, with coder/websocket showing the highest latency at 19.11 ms and ws showing the lowest latency at 17.81 ms.

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

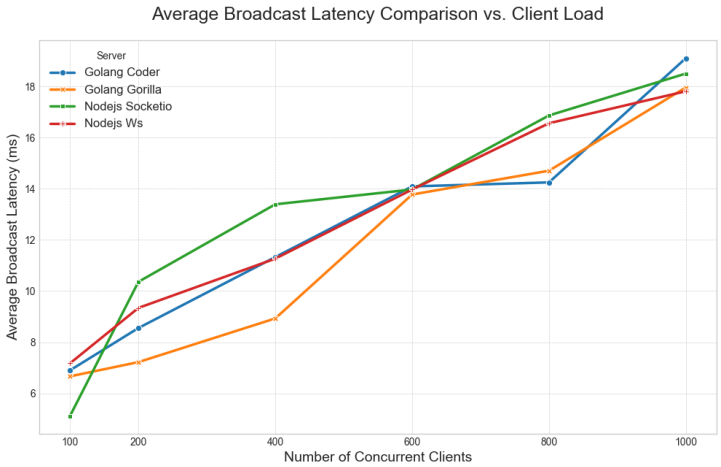

Fig. 6 Comparison of Average Broadcast Latency vs. Client Load

Statistical Summary at Peak Load

To provide a more detailed and academic summary that complements the visual trends, Table 2 presents a comprehensive overview of the descriptive statistics for all measured metrics at the peak load of 1,000 concurrent clients. This table includes measures of central tendency (Mean and Median) as well as data variability (Standard Deviation), offering deeper insight into the performance and stability of each library under maximum stress.

Table 2. Descriptive Statistics of All Metrics at 1,000 Clients Load

WebSocket Server	Metric	Mean	Median	Std. Dev. (STD)
Node.js - Ws	Connection Time (ms)	45.14	41.30	29.24
	RTT (ms)	18.75	19.64	8.39
	Throughput (msg/s)	45,493.14	45,171.0	5,657.83
	Broadcast Latency (ms)	17.81	18.02	7.70
Node.js - Socket.IO	Connection Time (ms)	185.96	186.24	113.04
	RTT (ms)	31.23	33.78	12.52
	Throughput (msg/s)	27,152.22	27,107.0	2,739.99
	Broadcast Latency (ms)	18.50	18.19	9.22
Golang - Gorilla	Connection Time (ms)	43.72	41.09	25.92
	RTT (ms)	19.20	22.04	6.72
	Throughput (msg/s)	44,213.69	43,996.0	5452.37
	Broadcast Latency (ms)	17.96	18.39	7.07
Golang - Coder	Connection Time (ms)	42.93	41.28	25.29
	RTT (ms)	18.97	21.50	6.53
	Throughput (msg/s)	45,271.16	44,984.0	2,718.08
	Broadcast Latency (ms)	19.11	19.73	7.62

*name of corresponding author

Sinkron : Jurnal dan Penelitian Teknik Informatika
Volume 9, Issue 4, October 2025
DOI : https://doi.org/10.33395/sinkron.v9i4.15266
e-ISSN : 2541-2019
p-ISSN : 2541-044X

DISCUSSIONS

Analysis and Interpretation of Results

Based on the test results obtained, a significant finding was noted, specifically the contrasting performance differences between the socket.io library and the three other libraries (ws, gorilla/websocket, and coder/websocket). In all Echo Test metrics, namely Connection Time, RTT, and Throughput, socket.io consistently showed lower performance. This is because socket.io has a more complex architecture with additional features such as fallback to HTTP long-polling and namespaces. Although these features are handy in some scenarios, they inherently add overhead to each connection and data packet, which negatively impacts latency and throughput in this benchmarking test. This highlights a fundamental trade-off, where socket.io prioritizes developer experience and cross-browser compatibility with fallbacks, whereas libraries like ws prioritize a lightweight, efficient implementation that adheres closely to the WebSocket protocol standard.

On the other hand, the performance between ws (Node.js), gorilla/websocket (Golang), and coder/websocket (Golang) shows a very high level of competitiveness, especially at peak loads. In the RTT and Throughput metrics, the values of these three libraries converge as the number of clients increases. This indicates that under heavy workloads, the performance bottleneck no longer lies in the efficiency of the library itself, but instead shifts to other factors, such as platform runtime limitations (the event loop in Node.js or the scheduler in Go) or even hardware limitations. This also demonstrates that for simple point-to-point communication scenarios, both Node.js and Golang with the appropriate libraries can offer excellent scalability and performance. The strong performance of the Golang libraries, which matched the highly optimized Node.js ws library without complex manual tuning, can be attributed to Golang's native concurrency model. The efficiency of its default scheduler in handling thousands of goroutines for I/O tasks allowed it to effectively rival Node.js's event-driven architecture in this specific point-to-point communication scenario.

In the Broadcast Test scenario, an interesting result is that the latency performance of the four libraries became very similar at a load of 1000 clients. However, at low loads, the differences were more noticeable. This indicates that in broadcast operations with a large number of receiving clients, the primary task of the server is to perform I/O operations to send data to hundreds or thousands of TCP connections simultaneously. In this scenario, the minor overhead of each library becomes less significant compared to the network I/O load itself, resulting in their performance being relatively equivalent.

Practical Implications and Library Selection Guide

The results of this benchmark offer clear, practical insights for developers and architects designing real-time systems, as the choice of a WebSocket library should be directly aligned with an application's specific performance requirements. For high-performance, low-latency applications, such as those in online gaming or fintech, where the highest possible throughput and lowest RTT are critical, the minimalist design of ws on Node.js, alongside gorilla/websocket or coder/websocket on Golang, makes them the superior choices. Conversely, for enterprise and general-purpose applications like chat apps or collaborative tools, where feature completeness and reliability might outweigh raw speed, socket.io remains a viable option. Its built-in features, such as automatic reconnection and a crucial fallback to HTTP long-polling, provide a resilient solution for restrictive corporate networks, making the performance trade-off acceptable. Furthermore, for applications involving large-scale data dissemination, like IoT dashboards or live news feeds, the minimalist libraries (ws, gorilla, coder) are still preferable. Although the latency difference at peak broadcast loads was minimal, their superior resource efficiency, as also supported by Hansson (2020), becomes a critical factor in managing server costs when handling thousands of persistent connections.

Comparison with Previous Studies

When compared to previous studies, the findings of this research are consistent with some results but also show interesting differences. The results of this research strongly support the findings of Hansson (2020), who also found that implementing plain WebSocket (in this case, represented by the ws library) is significantly faster and has better memory scalability compared to socket.io in the JavaScript ecosystem. Therefore, this research confirms that the overhead in socket.io is a consistent factor affecting performance at high concurrency levels.

However, when compared to Tomasetti (2021), a difference is observed. Tomasetti concluded that Node.js exhibits high-speed performance, whereas Golang (using gorilla/websocket) tends to be slower without manual concurrency optimisation. Conversely, this research found that gorilla/websocket and coder/websocket can match the performance of ws on Node.js directly in the Echo Test scenario without requiring complex multi-goroutine configuration. This difference is due to the differing nature of the tests, where the Echo Test scenario is highly efficient when handled by Golang's default scheduler, allowing Golang's concurrency advantages to offset the event-driven architecture of Node.js sufficiently.

*name of corresponding author

e-ISSN : 2541-2019
p-ISSN : 2541-044X

Research Limitations

This research has several limitations that need to be acknowledged, and these limitations may also provide opportunities for future research. First, all testing was conducted on hardware with an ARM-based CPU architecture (Apple M4 Pro). Performance results may differ when run on more common server architectures in production environments, such as x86-based CPUs. This is due to fundamental differences in instruction sets and operating modes, where x86 CPUs often demonstrate superiority in intensive single-threaded workloads, while ARM CPUs excel in power efficiency for parallel tasks (Gupta & Sharma, 2021)ad not measure the impact of security layers such as TLS/SSL. A research by Paris, Habaebi, & Zyoud (2023) shows that the implementation of SSL/TLS encryption can increase energy consumption by up to four times and drastically increase network overhead compared to unencrypted connections.

CONCLUSION

Based on the empirical data analysis obtained, several key conclusions can be drawn. First, in the point-to-point communication scenario (Echo Test) with a load of 1,000 clients, there is a clear trade-off between features and performance, where the socket.io library consistently shows the lowest performance with an average throughput of only 27,152 messages/second and an RTT of 31.23 ms. Second, the other three libraries (ws, gorilla/websocket, and coder/websocket) show significantly superior and highly competitive performance, with throughput exceeding 44,000 messages/second and RTT in the range of 19 ms. Third, in the Broadcast Test scenario with high load, the latency performance difference among the four libraries became insignificant, ranging from 17.81 ms to 19.11 ms. This indicates that under massive network I/O load, the primary bottleneck shifts from library efficiency to system limitations. Therefore, this research answers the research question by concluding that for applications that prioritize raw performance in point-to-point communication, the ws library in Node.js and gorilla/websocket and coder/websocket in Golang are superior choices. However, for large-scale broadcast scenarios, the choice between the four libraries has less impact on latency.

The findings of this research have both academic and practical implications. Academically, this research contributes to the field of communication protocol benchmarking by providing a novel, head-to-head comparative methodology that assesses multiple libraries across different backend platforms (2 libraries on each backend in two test scenarios: Echo Test and Broadcast Test), filling a significant gap in the existing literature. Practically, the results serve as an empirical guide for developers and companies, enabling them to make data-driven decisions when selecting a technology stack. The data clearly indicates that for performance-critical applications (fintech, online gaming), minimalist libraries are superior, whereas for applications requiring robust fallbacks, the performance cost of a library like socket.io may be a justifiable trade-off.

While this research provides a foundational benchmark, several avenues for future research are recommended to build upon these findings. Future work should focus on validating these results in more realistic production environments, such as by conducting tests over real-world networks (WAN and cloud infrastructures) to account for variable latency and packet loss. Furthermore, a crucial next step would be to quantify the performance overhead of implementing a TLS/SSL security layer, which is standard practice in production. Finally, extending these benchmark scenarios to specific, high-demand use cases like high-frequency IoT data ingestion and 5G communication protocols would provide even more valuable insights for developers building next-generation real-time applications.

ACKNOWLEDGMENT

This research was successfully conducted thanks to the assistance and support from various parties. Therefore, the researcher would like to express sincere gratitude to Author[2)], for his guidance, direction, and valuable input throughout the research process and the preparation of this journal. Additionally, the researcher would like to extend their gratitude to all parties who have provided support and contributions, which cannot be mentioned individually.

REFERENCES

Abdelfattah, A. S., Abdelkader, T., & EI-Horbaty, E.-S. M. (2020). RAMWS: Reliable approach using middleware and WebSockets in mobile cloud computing. *Ain Shams Engineering Journal*, *11*(4), 1083–1092. doi:10.1016/j.asej.2020.04.002

Alexeev, V. A., Domashnev, P. V., Lavrukhina, T. V., & Nazarkin, O. A. (2019). The Design Principles of Intelligent Load Balancing for Scalable WebSocket Services Used with Grid Computing. *Procedia Computer Science*, *150*, 61–68. doi:10.1016/j.procs.2019.02.014

Ariffin, N. I., Hamdan, Muhd. A. S., & Kamarulzaman, S. F. (2023). *Internet of Things Intercommunication Using SocketIO and WebSocket with WebRTC in Local Area Network as Emergency Communication Devices*. In *2023 IEEE 8th International Conference On Software Engineering and Computer Systems (ICSECS)* (pp. 268–273). IEEE. doi:10.1109/ICSECS58457.2023.10256297

*name of corresponding author

Bisták, P. (2020). Remote Control Laboratory for Three-Tank Hydraulic System Using Matlab, Websockets and JavaScript. *IFAC-PapersOnLine*, *53*(2), 17240–17245. doi:10.1016/j.ifacol.2020.12.1766

Bistak, P., Huba, M., Drahos, P., Belai, I., & Vrancic, D. (2024). Magnetic Levitation Remote Control Laboratory Based on Matlab and Websockets. *IFAC-PapersOnLine*, *58*(9), 235–240. doi:10.1016/j.ifacol.2024.07.402

Chodorek, A., & Chodorek, R. R. (2025). Web Real-Time Communications-Based Unmanned-Aerial-Vehicle-Borne Internet of Things and Stringent Time Sensitivity: A Case Study. *Sensors*, *25*(2), 524. doi:10.3390/s25020524

Friendly, Padly Sembiring, A., Faza, S., & Luckyhasnita, A. (2022). Speed Comparison OF Websocket And HTTP In IOT Data Communication. *INFOKUM*, *10*(5), 46–51. Retrieved from http://infor.seaninstitute.org/index.php/infokum/index

Gote, A. (2024). REAL-TIME INTERACTIVITY IN HYBRID APPLICATIONS WITH WEB SOCKETS. *International Research Journal of Modernization in Engineering Technology and Science*, *6*(1), 2459–2463. doi:10.56726/IRJMETS48494

Gupta, K., & Sharma, T. (2021). Changing Trends in Computer Architecture : A Comprehensive Analysis of ARM and x86 Processors. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, *7*(3), 619–631. doi:10.32628/CSEIT2173188

Hansson, J. (2020). *Performance study of JavaScript WebSocket frameworks*. Digitala Vetenskapliga Arkivet. Retrieved from https://liu.diva-portal.org/smash/record.jsf?pid=diva2%3A1459815&dswid=22

Huang, X. (2020). *Research and Application of Node.js Core Technology*. In *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)* (pp. 1–4). IEEE. doi:10.1109/ICHCI51889.2020.00008

Kirilov, N. (2024). Comparison of WebSocket and Hypertext Transfer Protocol for Transfer of Electronic Health Records (pp. 124–128). IOS Press. doi:10.3233/SHTI240023

Luo, J., Zhou, B., Zheng, Y., & Pan, W. (2024). Research on high performance web service construction method based on JavaScript asynchronous programming technique. *Applied Mathematics and Nonlinear Sciences*, *9*(1), 1–16. doi:10.2478/amns-2024-2811

Mitrovic, N., Eordevic, M., Veljkovic, S., & Dankovic, D. (2021). *Implementation of WebSockets in ESP32 based IoT Systems*. In *2021 15th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS)* (pp. 261–264). IEEE. doi:10.1109/TELSIKS52058.2021.9606244

Murley, P., Ma, Z., Mason, J., Bailey, M., & Kharraz, A. (2021). *WebSocket Adoption and the Landscape of the Real-Time Web*. In *Proceedings of the Web Conference 2021* (pp. 1192–1203). New York, NY, USA: ACM. doi:10.1145/3442381.3450063

Paris, I. L. B. M., Habaebi, M. H., & Zyoud, A. M. (2023). Implementation of SSL/TLS Security with MQTT Protocol in IoT Environment. *Wireless Personal Communications*, *132*(1), 163–182. doi:10.1007/s11277-023-10605-y

Puranik, D. G., Feiock, D. C., & Hill, J. H. (2013). *Real-Time Monitoring using AJAX and WebSockets*. In *2013 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS)* (pp. 110–118). IEEE. doi:10.1109/ECBS.2013.10

Ramu, V. B. (2023). Performance Testing for Hybrid Cloud Environments. *The Review of Contemporary Scientific and Academic Studies*, *3*(7). doi:10.55454/rcsas.3.07.2023.005

Sekar, R. (2025). Real-Time Data Streaming: Advancing Technologies, Future Trends, and Industry Applications. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, *11*(2), 513–521. doi:10.32628/CSEIT25112383

Soewito, B., Christian, Gunawan, F. E., Diana, & Kusuma, I. G. P. (2019). Websocket to Support Real Time Smart Home Applications. *Procedia Computer Science*, *157*, 560–566. doi:10.1016/j.procs.2019.09.014

Suwarno, & Putri Yulandi, A. (2023). Analisis Performa Backend Framework: Studi Komparasi Framework Golang dan Node.js. *Jurnal Riset Sistem Informasi Dan Teknik Informatika*, *8*(1), 155–168. Retrieved from https://www.tunasbangsa.ac.id/ejurnal/index.php/jurasik/article/view/551

Tomasetti, M. (2021). An Analysis of the Performance of Websockets in Various Programming Languages and Libraries. Retrieved from http://dx.doi.org/10.2139/ssrn.3778525

*name of corresponding author