BULLETIN of Cherkasy State Technological University

Journal homepage: https://bulletin-chstu.com.ua/en Vol. 30 No. 3. 2025

UDC 004.42:004.75
DOI: 10.62660/bcstu/3.2025.10

Framework for the comparative analysis of networking software libraries and its application to C++ networking solutions

Yehor Hrushevyy*
Master
Taras Shevchenko National University of Kyiv
01033, 60 Volodymyrska Str., Kyiv, Ukraine
https://orcid.org/0009-0005-7695-1990

Kostiantyn Zhereb
PhD in Physics and Mathematics, Assistant
Taras Shevchenko National University of Kyiv
01033, 60 Volodymyrska Str., Kyiv, Ukraine
https://orcid.org/0000-0003-0881-2284

Abstract. The objective evaluation of networking libraries remains a critical challenge in software engineering, as inconsistent methodologies and application-specific code often obscure meaningful performance and usability differences. The aim of this study was to conduct a comprehensive comparison of three C++ networking libraries (Boost.Asio, Boost.Beast, and Poco.Net) based on the proposed universal framework. The research methodology included the development of a universal comparative framework, experimental implementation of software prototypes, load testing to collect quantitative indicators, and a structured qualitative assessment based on a number of defined criteria. The framework encompassed both quantitative metrics (throughput, latency and resource consumption) and qualitative criteria (Application Programming Interface ergonomics, feature completeness, stability, cross platform compatibility and development complexity), with all libraries assessed under identical software conditions. Using each library, two C++ client-server prototypes were implemented: the RESTful service ResourceMonitor and the real time streaming application GameOfLifeStreaming. The project structure was unified to eliminate variability in application logic and to focus analysis exclusively on library behaviour. Boost.Asio demonstrated advantages in latency sensitive scenarios and scenarios requiring fine grained control. Boost.Beast offered effective HTTP support with minimal performance overhead but limited protocol coverage. Poco.Net achieved the lowest memory footprint and maintained stable performance at high loads while supporting the widest range of protocols, albeit at the expense of higher latency and lower raw throughput, and requiring more complex configuration. The practical contribution lies in the reusable framework for evaluating other networking libraries, the integration of created components into diverse development workflows, and the provided guidelines for choosing the most appropriate library

Keywords: application performance evaluation; load testing; client-server architecture; quantitative and qualitative metrics collection; testbed configuration; modular architecture; throughput and latency measurement

INTRODUCTION

In modern software engineering, objectively comparing and benchmarking libraries is essential to isolate the true impact of each implementation on performance, resource usage and developer ergonomics. Without a rigorous framework, application-specific logic or measurement inconsistencies can obscure the real strengths and weaknesses of candidate libraries. Numerous prior studies have addressed the challenge of selecting

Article's History: Received: 16.06.2025; Revised: 10.08.2025; Accepted: 15.09.2025.

Suggested Citation:
Hrushevyy, Ye., & Zhereb, K. (2025). Framework for the comparative analysis of networking software libraries and its application to C++ networking solutions. *Bulletin of Cherkasy State Technological University*, 30(3), 10-23. doi: 10.62660/bcstu/3.2025.10.

*Corresponding author

suitable software libraries by proposing structured methodologies and metricbased evaluation frameworks.

A variety of academic studies have also examined C++ networking stack. However, a comprehensive, side-by-side evaluation of Boost.Asio, Boost.Beast and Poco.Net within full-fledged client-server applications has yet to be conducted. E. Kadusic *et al*. (2022) investigated IPv6 address validation and security support in both Boost.Asio and Poco.Net, demonstrating that while both libraries offer flexible Application Programming Interfaces (APIs) for Internet of Things (IoT) device integration, their memory-handling strategies differ substantially, with Asio requiring more manual buffer management and Poco.Net leveraging hingher-level abstractions for certificate validation. Y. Pasichnyk (2022) and D. Butynets (2023) compared synchronous, asynchronous and hybrid server designs in pure C++ – often contrasting raw POSIX system calls against Asio's eventdriven model – and concluded that hybrid approaches can outperform purely threaded or purely asynchronous solutions under continuous stress, though each paradigm exhibits unique trade-off in central processing unit (CPU) utilisation and latency.

Several studies have employed Boost.Asio – and, to a lesser extent, Boost.Beast and Poco.Net – within broader application contexts, yet without focusing on detailed comparative analysis between the libraries. S. Kayum *et al*. (2020) integrated Boost.Asio into a fault-tolerant implementation of a massively parallel seismic processing framework, demonstrating its reliability and scalability in high performance computing environments with tens of thousands of cores. J. Ardarve *et al*. (2019) leveraged Poco.Net for a privacyenforcement agent under COPPA constraints, noting the library's ease of message-passing but not quantifying its runtime characteristics. In addition, cross-language comparisons shed light on C++ networking's relative strengths and weaknesses. C. Pflugfelder (2022) evaluated asynchronous programming across IoT platforms in C, C++, Rust, Lua and Python on ESP32 devices, finding that C++ (using Asio) consistently achieved lower end-to-end latency at the cost of increased code complexity.

Taken together, these works highlight the maturity and versatility of high-level C++ networking libraries but also reveal significant gaps. Prior research has tended to focus on individual libraries, specific protocols, or broader language-level comparisons, without offering a unified and empirical evaluation of Boost.Asio, Boost.Beast and Poco.Net across consistent, real-world client-server scenarios. Although isolated benchmarks have shed light on the strengths of these libraries, no prior work has systematically compared their behaviour under identical conditions using reproducible, modular setups.

In parallel to the focused evaluation of C++ networking libraries, the broader ecosystem of software comparison and communication frameworks offers valuable insights that inform this methodology. P. Sai *et al*. (2024) developed a Python-based real-time task manager with Psutil and Tkinter for system monitoring, illustrating the importance of modular, responsive data collection. Y. Baddi *et al*. (2023) proposed an SDNbased multicast protocol for largescale IoT deployments, demonstrating how flexible group communication can optimise latency and scalability. A. Samoydiuk & O. Ostapchuk (2024) investigated asynchronous programming techniques in smart-home management systems, demonstrating significant performance gains in high-load scenarios. Collectively, these works reinforce the value of a dedicated, structured framework for reproducible, multi-dimensional evaluation of networking solutions across diverse environments.

The aim of this study was to evaluate and contrast three widely used C++ networking libraries – Boost.Asio, Boost.Beast, and Poco.Net using a consistent comparative framework. To achieve this aim, the following tasks were undertaken. First, to define a consistent set of criteria and metrics suitable for the comparative analysis of networking libraries, and to apply them to the assessment of the selected libraries via two representative client-server applications. Second, to collect both quantitative and qualitative data through controlled experiments and structured observation. Third, to analyse the resulting data to produce a structured comparison across key dimensions, including performance, API ergonomics, feature completeness, reliability, portability, and community support.

MATERIALS AND METHODS

The comparative framework was centred on seven core dimensions (performance and efficiency, architecture and API, functionality, developer ergonomics, reliability and stability, portability and compatibility, community support), each of which was chosen to capture a distinct facet of a networking library's suitability for real-world deployment. The performance and efficiency dimension was used to quantify raw throughput and latency under representative workloads, to ensure that libraries could satisfy application-level service-level agreements. Metrics such as requests per second, mean response time, and high-percentile latencies were selected because they directly reflect user-perceived responsiveness and system scalability. The architecture and API design were assessed for clarity, consistency and modularity of the library's interfaces; well-designed APIs reduce cognitive load and lower the risk of integration errors, so measures of API surface area and adherence to established design idioms formed this dimension. Functionality examined protocol coverage, extensibility mechanisms and built-in utilities – libraries that natively support a broader set of protocols can address a wider range of use-cases without resorting to custom implementations.

Developer ergonomics were evaluated, recognising that ease of learning and use heavily influences time-to-market and long-term maintenance costs; to capture this, the complexity of API required for common tasks,

the prevalence of synchronous versus asynchronous patterns, and the availability of diagnostic messages were estimated. Reliability and stability were used to measure a library's fault-handling model, resource-cleanup guarantees and behaviour under stress, since production services must recover gracefully from network anomalies. Portability and compatibility were used to gauge the effort needed to build and deploy across different platforms and toolchains, ensuring that a library can +serve diverse environments from embedded devices to cloud services. Finally, community support was assessed to reflect each library's ecosystem health-release cadence, issue-tracker responsiveness and third-party contributions – as sustained maintenance and peerreviewed improvements are vital for longlived systems.

To operationalise the performance and efficiency dimension, a set of quantifiable indicators (such as latency, throughput, memory footprint and CPU utilisation) was defined and uniform procedures were established for collecting the relevant measurements using industry-standard tools, including ApacheBench and Windows Performance Monitor. A central tenet of the framework was the isolation of the networking layer via a uniform client-server interface. Each library under test was encapsulated within a plugin module conforming to the same IClient/IServer API. Buildtime selection of the desired networking module was handled by build variables, to guarantee that all other application logic remained constant. This modular architecture not only ensured that observed differences in behaviour could be attributed solely to the libraries themselves but also provided a reusable pattern for applying the framework to other programming languages or network-oriented libraries with minimal additional effort.

All experiments and prototype development were conducted on a Windows 11 environment using Visual Studio Code as the primary IDE. The build system was configured using CMake version 3.25.1-msvc1, targeting the Microsoft C++ compiler cl.exe version 19.35.32216.1 with the C++20 standard. Python 3.13.0 was used for the data-collection and database agents. Source control and version management were performed using Git version 2.39.0.windows.2. The networking libraries under investigation included Boost version 1.86.0 (specifically the Boost.Asio and Boost.Beast components) and POCO C++ Libraries version 1.14.1 (Poco::Net and associated modules). Additional dependencies included the nlohmann/json library version 3.11.3 for JSON serialisation and deserialisation, and raylib version 5.5 for graphical rendering in the streaming prototype. Performance metrics were collected using Windows Performance Monitor version 10.0. HTTP benchmarking was performed using ApacheBench from the apache2-utils package version 2.4.52-1ubuntu4.14 on the Ubuntu 22.04.5 LTS guest system with access via VirtualBox 7.0.14. Source code size was evaluated using cloc (Count Lines of Code) version 1.90. SQLite version 3.45.3 was employed as the database backend.

Two complete client-server applications were developed to exercise and compare the network libraries under study. These applications were chosen to represent distinct categories of network communication – request-response and real-time streaming – which enabled the evaluation of each library across contrasting usage scenarios. The first, ResourceMonitor, was designed as a distributed monitoring system for computer resource usage. It comprised five interchangeable modules (Fig. 1): a data-collection agent, a database-interaction agent, a central server, a command-line client, and a web-based client. The data-collection agent periodically sampled CPU, memory, disk and network metrics and transmitted them via HTTP to the central server, which in turn validated and forwarded the metrics to the database agent. Both user-facing clients (console and Vue.jsbased GUI) issued HTTP requests to the server and rendered the returned JSON data. To isolate networking behaviour, all three C++ libraries (Boost.Asio, Boost.Beast and Poco.Net) were encapsulated behind a common IClient/IServer interface and were selected at build time via CMake variables.

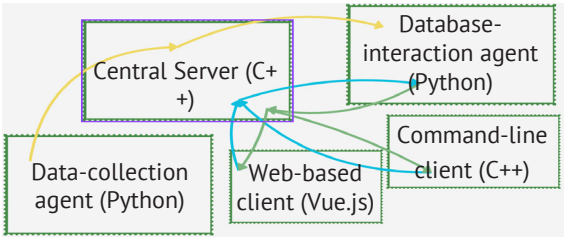

Figure 1. Components of ResourceMonitor
Source: created by the authors

For the Boost.Asio variant, the code was divided into three logical parts – client-specific, server-specific, and shared between client and server. A shared core defined HttpRequest and HttpResponse classes inheriting from a base HttpMessage, responsible for constructing and parsing the textual representation of HTTP messages. The client class managed a single io_context, driving asynchronous domain name system (DNS) resolution, connection establishment, request transmission and response reception – each operation performed by explicit calls to Asio's asynchronous functions and coordinated on a dedicated Input/Output (I/O) thread. Request cancellation was supported through Asio timers. Correspondingly, the server class used an Asio acceptor to receive incoming transmission control protocol (TCP) connections, then issued asynchronous read and write operations to parse requests, invoke the database agent, and send back JSONformatted responses.

In contrast, the Boost.Beast implementation leveraged Beast's higher-level HTTP constructs. A single Session class encapsulated both client and server logic, using Beast's http::async_read/async_write to process entire messages in one operation. The Beast client simply composed a http::request, submitted it via a Beast stream,

and awaited a http::response object, while the server acceptor spawned a new Session for each connection, automatically handling header framing, body buffering and protocol compliance. By reusing the same Boost-Common primitives, such as the I/O service singleton and the database request manager, both the Asio and Beast implementations shared identical lowerlevel infrastructure.

The Poco.Net version took a yet more high-level approach. In ResourceMonitor, HTTP functionality was provided by HTTPClientSession, HTTPRequest and HTTPResponse classes. The client constructed and dispatched requests, while the server was realised via Poco's HTTPServer framework, with a custom HTTPRequestHandlerFactory producing HTTPRequestHandler instances per request. Custom threadpool manager was introduced to overcome Poco's default pool limitations, queueing tasks and dispatching them only when worker threads became available. Data collection was handled by a Python-based agent that ran as a background service on each monitored host. This agent periodically invoked native system APIs to sample CPU, memory, disk and network metrics, packaged the results as JSON, and transmitted them via HTTP requests to the central server. The choice of Python for this component was motivated by its rich ecosystem for system monitoring and its rapid development cycle, which permitted the agent to be implemented and refined with minimal effort.

Central server performed validation of user requests and database responses and organised communication between other components. It had a list of configurable options such as port, number of threads or coordinates of database agent. Its implementation included controller with high-level logic and configuration parser. The central server's networking layer was exclusively targeted and measured during the performance tests. Python database agent exposed a simple Flaskbased HTTP interface for accepting and querying metric data. This agent mapped each request to operations on an SQLite database. The database schema – initialised on first run – comprised a primary mapping of IP addresses to internal host IDs and a series of timeseries tables (e.g. CPU, memory, disk and network usage), each keyed by host ID and timestamp. By isolating persistence logic in Python, the central C++ server and client components could rely on a uniform HTTP API without embedding any database-specific code. User interaction in ResourceMonitor was facilitated by two distinct frontends. A web dashboard, implemented in Vue.js, issued asynchronous HTTP GET requests to REST endpoints, parsed the returned JSON payloads, and rendered real-time charts to visualise resource trends. In parallel, a C++ command-line client offered a text-based interface: users configured server address, logging level and output paths through Boost.Program_options; entered commands such as request or cancel; and received formatted responses via IClient/IServer interface employed by the underlying networking module.

The second prototype, GameOfLifeStreaming, simulated John Conway's Game of Life and transmitted its evolving universe state as a real-time data stream to multiple clients. In this simpler architecture, a C++ server carried the simulation logic and periodically emitted the entire universe state over UDP multicast. Clients subscribed to the multicast group, reassembled each complete simulation state – hereafter referred to as a "frame" – and rendered it using the lightweight raylib graphics library. Networking for the GameOfLifeStreaming prototype was encapsulated behind similar IClient/IServer abstraction layer employed in the ResourceMonitor application. Each library's implementation – Boost.Asio, Boost.Beast and Poco.Net – was compiled as a separate static networking module, selectable at build time via a single CMake flag. The core networking interface provided methods to initialise connections, send and receive raw "frames", and register callback handlers for events such as "frame" arrival or connection errors.

Under Boost.Asio and Poco.Net – both of which provide native support for UDP features – the networking layer allowed the application to open a UDP multicast socket bound to the configured group address and port, and to spawn an I/O thread to asynchronously receive datagrams. Incoming UDP packets were passed into a "frame" reassembly component, which detected complete universe "frames" by scanning for the terminating newline marker. In contrast, Boost.Beast does not natively support UDP or multicast communication. To preserve the integrity of the comparison – and ensure that each library is evaluated solely on its own capabilities – it was opted not to introduce third-party dependencies. Instead, Beast's performance was assessed using a semantically equivalent TCP-based streaming approach via WebSocket. The IClient implementation used Beast's websocket::async_read and websocket::async_write calls to receive and send text "frames", while the server maintained a pool of session objects, each driving its own asynchronous read/write operations.

The server comprised three cooperating C++ classes: a coordinator that initialised and gracefully shut down the application, a configuration parser built with Boost.Program_options that validated command-line arguments (e.g. universe dimensions, tick rate, thread count, logging options), and a simulation class that applied Conway's rules to update a two-dimensional grid. Each update generated a concise text "frame" – prefixed by three-digit width and height specifiers and followed by a stream of # and space characters – then delivered it to the networking layer. Each project's source tree was partitioned into a root directory containing global configuration files (e.g. gitignore, CMakeLists.txt) and subdirectories for the client, server, utility and networking components. The CMake build scripts were structured hierarchically: a toplevel CMakeLists.txt defined the overall project, required CMake version and C++ standard, and delegated control to per-module CMakeLists.txt

files. In each module, source and header files were added to either a static library or an executable target.

A central config.cmake file provided paths to external dependencies (Boost, Poco, raylib, nlohmann/json) and compile-time options to select the desired networking implementation. By toggling a single CMake variable, the build system included exactly one of the three static networking libraries – Asio, Beast or Poco.Net – thereby producing three variants of each prototype without altering application code. Common C++ utility components were factored into their own static library to avoid duplication. It included logging system, built on the Strategy pattern, input/output helper functions for argument parsing and working with JSON, and a Singleton pattern.

The performance and stability of each prototype variant were evaluated under controlled laboratory conditions to ensure reproducibility. A uniform hardware and network testbed was established comprising three hosts (Fig. 2): two Windows 11 machines and Ubuntu 22.04 virtual guest (hosted via VirtualBox). Ubuntu guest was configured with a bridged network adapter, thereby obtaining a distinct IP address on the same LAN as its host. Entries for test nodes were added to each machine's hosts file to permit name-based addressing. Windows Firewall rules were adjusted on the physical hosts to allow incoming TCP, UDP and ICMP traffic from the other test nodes.

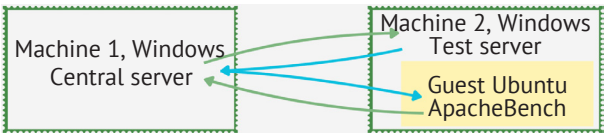

Figure 2. Testbed for ResourceMonitor
Source: created by the authors

HTTP performance was measured using ApacheBench (invoked via the ab command-line tool), selected for its wide adoption and ability to generate configurable request loads. For each test scenario, ab was invoked with parameters -n (total requests) and -c (concurrent clients) against the ResourceMonitor HTTP endpoint, yielding per-request latency and throughput statistics. Streaming latency in GameOfLifeStreaming was measured using embedded timestamps: the server prefixed each multicast payload with its local time, and the client computed time by comparing receipt time against this timestamp. System resource usage on Windows hosts was sampled with Performance Monitor (perfmon), capturing Processor Time, Working Set and Private Working Set at one-second intervals. These metrics were exported as time series for analysis of CPU and memory trends under sustained load.

Two test harnesses were devised to exercise each networking library. For the HTTP workload, the server endpoint under test accepted GET requests, redirected them to test server (that replaced database component and returned preconfigured JSON bodies) and returned its answers to client. ApacheBench was used to generate load from a separate client host, varying concurrency parameters (c = 5, 100, 500) and total request counts (n = 1,000, 15,000, 10,000). Server builds were configured with either two or six threads. Key metrics captured for comparison were requests per second (throughput), mean time per request (ms), 90th percentile latency (ms), 99th percentile latency (ms) and system resource utilisation (CPU and memory footprint) monitored via Windows Performance Monitor.

The streaming scenario measured end-to-end latency and resource consumption when broadcasting "Game of Life" "frames" to many subscribers. Tests spawned 5, 100, 1,000 and 15,000 logical clients within a single process, each collecting 1,000 timestamped "frames" to compute mean latency. Server configurations again alternated between two and six threads (for Beast implementation, not applicable for Asio and Poco.Net implementations), and Performance Monitor logged CPU and memory trends throughout each run.

In addition to empirical performance testing, each networking library was evaluated using a structured comparison across a set of other criteria. These included functionality breadth, developer usability, codebase complexity, runtime reliability, portability and compatibility, and community support. The purpose of this complementary analysis was to capture aspects not visible through benchmarking alone but essential for real-world applicability. A small set of objective questions (Table 1) was defined for each dimension. Binary questions were scored as yes/no (1 or 0), while others were evaluated on a 5-point scale (0, 0.25, 0.5, 0.75, 1). The raw sums of these perquestion scores yielded a dimension-specific total, which can be compared directly across libraries.

Table 1. Questions for qualitative evaluation

No.	Dimension	Question	Scoring
1.1	Architecture and API	Does the library separate connection establishment, data transfer and message processing into distinct modules?	yes/no (1/0)
1.2	Architecture and API	Are all public classes and methods documented with parameter descriptions, behavioural notes and exception details?	yes/no (1/0)
1.3	Architecture and API	How well does the API adhere to standard C++ naming and style conventions?	scale (0-1)
1.4	Architecture and API	How easily can a new protocol be integrated into the API without modifying the library's source?	scale (0-1)
2.1	Functionality	Rate amount of protocols the library support out of the box.	scale (0-1)

Continued Table 1.

No.	Dimension	Question	Scoring
2.2	Functionality	Rate the adequacy of built-in utilities for SSL/TLS certificate management, DNS resolution and connection recovery.	scale (0-1)
2.3	Functionality	How straightforward is it to switch between synchronous and asynchronous modes within the same API?	scale (0-1)
2.4	Functionality	Does the library guarantee threadsafe use of core objects across multiple threads?	yes/no (1/0)
3.1	Developer Ergonomics	Estimate the number of lines of code required to perform basic tasks.	scale (0-1)
3.2	Developer Ergonomics	Rate the quality of error diagnostics (exception detail, error codes).	scale (0-1)
3.3	Developer Ergonomics	How comprehensive are the "end-to-end" usage examples in the documentation?	scale (0-1)
3.4	Developer Ergonomics	Rate learning curve.	scale (0-1)
4.1	Reliability and Stability	Are open sockets and timers automatically closed on unexpected errors?	yes/no (1/0)
4.2	Reliability and Stability	Does the library support automatic reconnection following a lost connection?	yes/no (1/0)
4.3	Reliability and Stability	Rate the library's resilience under sustained load (no hangs, no memory leaks).	scale (0-1)
4.4	Reliability and Stability	Does the library provide built-in timeout mechanisms and deadlock prevention?	yes/no (1/0)
5.1	Portability and Compatibility	Rate the number of supported platforms (Windows, Linux, macOS, embedded).	scale (0-1)
5.2	Portability and Compatibility	Rate the number of external dependencies required for core functionality.	scale (0-1)
5.3	Portability and Compatibility	How simple is integration with common build systems?	scale (0-1)
5.4	Portability and Compatibility	Can both 32bit and 64bit builds be produced without source modifications?	yes/no (1/0)
5.5	Portability and Compatibility	Are prebuilt binaries provided for multiple target platforms?	(yes/no)
6.1	Community Support	How many active contributors are listed in the project repository?	scale (0-1)
6.2	Community Support	What is the average response time to issues in the official tracker?	scale (0-1)
6.3	Community Support	How frequently are new releases published (monthly, quarterly, biannual, annual)?	scale (0-1)
6.4	Community Support	Are there official discussion forums or mailing lists for support?	yes/no (1/0)
6.5	Community Support	Rate the number of commits in the project repository.	scale (0-1)

Source: created by the authors

RESULTS AND DISCUSSION

The following results reflect the application of the comparative analysis framework to three C++ networking libraries – Boost.Asio, Boost.Beast, and Poco.Net – across two representative real-world scenarios. Each metric presented here corresponds to one of the framework's seven evaluation dimensions. In line with the performance and reliability dimensions, all three implementations demonstrated robust results, with no HTTP requests failing during any of the tests. Under a two-thread configuration and lower load (Table 2), Boost.Asio achieved the highest throughput, followed by Boost.Beast. As the load increased, the relative ranking remained unchanged, with Asio leading, Beast in the middle, and Poco.Net trailing (Table 3).

Table 2. Throughput and latency for 15,000 requests and concurrency level 100, ResourceMonitor

Metric	Boost.Asio		Boost.Beast		Poco.Net	
	2 threads	6 threads	2 threads	6 threads	2 threads	6 threads
Requests per second	114.75	66.03	34.08	58.27	88.06	162.11
Mean time per request (ms)	43.57	75.73	146.71	85.82	56.78	30.84
90^{th} percentile latency (ms)	51	79	227	104	91	41
99^{th} percentile latency (ms)	268	1,099	1,135	141	255	59

Source: created by the authors

Table 3. Throughput and latency for 1,000 requests and concurrency level 5, ResourceMonitor

Metric	Boost.Asio		Boost.Beast		Poco.Net	
	2 threads	6 threads	2 threads	6 threads	2 threads	6 threads
Requests per second	351.04	485.57	354.32	333.77	103.48	276.49
Mean time per request (ms)	284.87	205.94	282.23	299.61	966.34	361.67
90th percentile latency (ms)	412	263	354	370	1 245	430
99th percentile latency (ms)	993	404	669	1,046	3,658	562

Source: created by the authors

Under high-concurrency conditions (Table 4), the disparities between the libraries became more pronounced. Boost.Asio preserved its performance advantage, while Poco.Net exhibited a sharp decline in throughput.

Table 4. Throughput and latency for 10,000 requests and concurrency level 500, ResourceMonitor

Metric	Boost.Asio		Boost.Beast		Poco.Net	
	2 threads	6 threads	2 threads	6 threads	2 threads	6 threads
Requests per second	447.15	478.66	386.72	357.15	106.83	266.66
Mean time per request (ms)	1,118.21	1,044.57	1 292.92	1,399.96	4,680.42	1,875.07
90th percentile latency (ms)	1,312	1,355	1,657	1,974	6,110	1,998
99th percentile latency (ms)	2,037	2,121	2,878	4,211	7,877	2,504

Source: created by the authors

Increasing the thread count to six did not alter this ranking, although Poco.Net outperformed the others at low concurrency levels (Table 2). Latency was characterised by mean response time as well as the 90th and 99th percentiles. For two threads (Table 2), Asio exhibited the lowest average latency, marginally ahead of Beast; Poco.Net's latency increased noticeably with higher concurrency (Tables 3, 4). At six threads, Poco.Net delivered the best mean latency under light loads but performed less favourably at higher loads, while Asio consistently outperformed Beast. The 90th percentile latencies followed similar patterns, with the gap between Beast and Poco.Net narrowing; at the 99th percentile, Poco.Net occasionally outperformed Beast, though Asio remained fastest. In the GameOfLifeStreaming tests (Table 5), UDP multicast support in Asio and Poco.Net did not yield a marked reduction in latency, indicating that client-side processing of many concurrent streams dominated delay rather than transport speed. Beast's WebSocketbased, percli-ent session model sometimes achieved lower average latency. Even at high client counts, absolute latencies remained modest, and increasing server thread count further reduced delays.

Table 5. Latency values, GameOfLifeStreaming

Clients number	Boost.Asio	Boost.Beast, 2 threads	Boost.Beast, 6 threads	Poco.Net
5	1.24	5.81	5.34	1.31
100	4.41	9.33	8.85	4.61
1,000	9.96	16.81	13.17	10.76
15,000	40.93	47.39	35.68	39.74

Source: created by the authors

CPU utilisation profiles across all ResourceMonitor test runs (Table 6) remained within expected bounds, exhibiting neither gradual increases nor pronounced spikes, thereby indicating stable processor load for each implementation. Working Set and Private Working Set metrics showed no evidence of memory leaks: values fluctuated steadily around a constant mean with no upward trend. In shorter runs Asio and Beast occasionally displayed a logarithmic growth pattern, but extended-duration tests revealed prompt resource reclamation. At peak load Asio and Beast consumed up to 52.5 MB Working Set compared to Poco.Net's 9.8 MB, a difference negligible on modern servers but potentially significant in resource-constrained environments.

Table 6. Resources usage metrics, ResourceMonitor

Metric	Boost.Asio		Boost.Beast		Poco.Net	
	2 threads	6 threads	2 threads	6 threads	2 threads	6 threads
Processor time, %, max	17	19	17	20	18	20
Processor time, %, mean	6	8	7	8	8	10
Working set, max, MB	48.3	52.5	47.3	49.4	8.1	9.8
Working set, mean, MB	22.1	27.4	20.8	24.5	7.2	7.6
Working set – private, max, MB	44.8	49.3	43.9	45.2	2.2	4.2
Working set – private, mean, MB	19.2	21.6	18.7	42.11	1.7	3.7

Source: created by the authors

In the GameOfLifeStreaming scenario (Table 7), Asio and Poco.Net benefited from UDP multicast to maintain low server resource consumption. By contrast, Beast's perclient sessions imposed greater CPU and memory overhead; with 15,000 concurrent clients, peak memory usage reached around 490 MB. The results of the qualitative evaluation are presented in Table 8. According to these results, Poco.Net leads in most dimensions.

Table 7. Resources usage metrics, GameOfLifeStreaming

Metric	Boost.Asio	Poco.Net	Boost.Beast	
			2 threads	6 threads
Processor time, %, max	6	15	73	98
Processor time, %, mean	4	7	41	45
Working set, max, MB	12.1	5.7	489.1	170.5
Working set, mean, MB	11.8	5.7	263.2	160.6
Working set – private, max, MB	5.8	0.8	260.9	153.2
Working set – private, mean, MB	5.7	0.7	241.7	142.1

Source: created by the authors

Table 8. Questions for qualitative evaluation

No.	Dimension	Question / Sum	Boost.Asio	Boost.Beast	Poco.Net
1.1	Architecture and API	Does the library separate connection establishment, data transfer and message processing into distinct modules?	1	1	1
1.2	Architecture and API	Are all public classes and methods documented with parameter descriptions, behavioural notes and exception details?	1	1	1
1.3	Architecture and API	How well does the API adhere to standard C++ naming and style conventions?	0.75	0.75	0.5
1.4	Architecture and API	How easily can a new protocol be integrated into the API without modifying the library's source?	0.25	0.25	0.75
1	**Architecture and API**	**Sum**	**3**	**3**	**3.25**
2.1	Functionality	Rate amount of protocols the library support out of the box.	0.25	0.25	1
2.2	Functionality	Rate the adequacy of built-in utilities for SSL/TLS certificate management, DNS resolution and connection recovery.	0.5	0.5	0.75
2.3	Functionality	How straightforward is it to switch between synchronous and asynchronous modes within the same API?	0.5	0.25	0
2.4	Functionality	Does the library guarantee threadsafe use of core objects across multiple threads?	1	1	1
2	**Functionality**	**Sum**	**2.25**	**2**	**2.75**
3.1	Developer Ergonomics	Estimate the number of lines of code required to perform basic tasks.	0.25	0.75	0.5
3.2	Developer Ergonomics	Rate the quality of error diagnostics (exception detail, error codes).	0.25	0.5	0.75

Continued Table 8.

No.	Dimension	Question / Sum	Boost.Asio	Boost.Beast	Poco.Net
3.3	Developer Ergonomics	How comprehensive are the "end-to-end" usage examples in the documentation?	0.25	0.5	0.75
3.4	Developer Ergonomics	Rate learning curve.	0.25	0.5	0.75
3	**Developer Ergonomics**	**Sum**	**1**	**2.25**	**2.75**
4.1	Reliability and Stability	Are open sockets and timers automatically closed on unexpected errors?	1	1	1
4.2	Reliability and Stability	Does the library support automatic reconnection following a lost connection?	0	0	1
4.3	Reliability and Stability	Rate the library's resilience under sustained load (no hangs, no memory leaks).	1	1	1
4.4	Reliability and Stability	Does the library provide built-in timeout mechanisms and deadlock prevention?	1	1	1
4	**Reliability and Stability**	**Sum**	**3**	**3**	**4**
5.1	Portability and Compatibility	Rate the number of supported platforms (Windows, Linux, macOS, embedded)?	1	1	1
5.2	Portability and Compatibility	Rate the number of external dependencies required for core functionality.	1	1	0.5
5.3	Portability and Compatibility	How simple is integration with common build systems?	0.75	0.75	0.5
5.4	Portability and Compatibility	Can both 32bit and 64bit builds be produced without source modifications?	1	1	1
5.5	Portability and Compatibility	Are prebuilt binaries provided for multiple target platforms?	1	1	1
5	**Portability and Compatibility**	**Sum**	**4.75**	**4.75**	**4**
6.1	Community Support	How many active contributors are listed in the project repository?	0.25	0.75	0.5
6.2	Community Support	What is the average response time to issues in the official tracker?	0.25	0.25	0.5
6.3	Community Support	How frequently are new releases published (monthly, quarterly, biannual, annual)?	0.5	0.5	0.75
6.4	Community Support	Are there official discussion forums or mailing lists for support?	1	1	1
6.5	Community Support	Rate the number of commits in the project repository.	1	1	0.5
6	**Community Support**	**Sum**	**3**	**3.5**	**3.25**
		Total Sum	**17**	**18.5**	**20**

Source: created by the authors

Boost.Asio offers the lowest level of abstraction and maximum flexibility by providing direct access to sockets, timers and file descriptors. Boost.Beast builds on top of Asio to raise the abstraction for specific application-level protocols (HTTP/HTTPS and WebSocket), but otherwise introduces no additional capabilities in connection management or I/O handling. By contrast, Poco.Net conceals system calls behind a well-structured, high-level API (factories, managers), which accelerates typical development tasks. The consistency of each library's API naturally correlates with its level of abstraction: Asio and Beast emphasise asynchronous operation, whereas Poco.Net favours synchronous patterns, which – as demonstrated by the measurements (Table 3, 4) – can be a notable drawback in latency-sensitive applications.

Boost.Asio supports raw TCP, UDP and ICMP sockets at the lowest level, with built-in asynchronous DNS resolution and timers. It accommodates both IPv4 and IPv6 and integrates readily with OpenSSL for SSL/TLS. However, it provides no application-level protocol implementations, requiring developers to implement or integrate additional libraries for HTTP, WebSocket, etc. Boost.Beast addresses this gap for HTTP and WebSocket, while preserving full access to Asio's low-level primitives; beyond these two protocols, no further application-level support is offered. Poco.Net provides the most comprehensive out-of-the-box functionality: TCP/UDP, IPv4/IPv6, DNS, HTTP/HTTPS, WebSocket, FTP, POP3 and SMTP, with SSL/TLS via NetSSL_OpenSSL.

Boost.Asio demands the steepest learning curve and development effort, owing to its low-level style and multitude of asynchronous operation calls. Documentation within Boost is well-structured but primarily APIcentric; example code is often focused on individual operations rather than complete solutions. Boost.Beast offers demonstration programs for HTTP and WebSocket

clients and servers, but error messages in both libraries tend to be technical, occasionally hindering rapid diagnosis. Poco.Net is designed for rapid prototyping, with extensive ready-made examples for common scenarios and more descriptive exception messages. Its principal downside is the larger surface area of external module dependencies. Code size, as measured by cloc (Table 9), reflects these trade-offs: Poco.Net exhibits big number of files and lines despite its high level of abstraction, owing to the necessity of additional handler and factory classes; Beast's WebSocket usage in GameOfLifeStreaming similarly increases its code footprint.

Table 9. Code size

Project	Boost.Asio		Boost.Beast		Poco.Net	
	Files	Lines	Files	Lines	Files	Lines
ResourceMonitor	15	1,165	13	782	17	929
GameOfLifeStreaming	5	320	9	788	7	561

Notes: this refers specifically to the size of the networking-related application code written using each library, rather than the size of the libraries themselves
Source: created by the authors

Boost.Asio employs both standard C++ exceptions (in synchronous code) and error_code objects (in asynchronous code) for error handling, with RAII ensuring proper resource cleanup. It is battle-tested in critical systems, although its asynchronous model can present complex debugging scenarios. Boost.Beast augments this with protocol-specific checks (e.g. header constraints) that improve resilience to malformed input or attacks, at the cost of additional exception handling. In Poco.Net, all errors derive from the hierarchical Poco::Exception base class, with built-in automatic reconnection mechanisms on connection loss. High-level interfaces in Poco.Net further reduce the incidence of low-level socket errors.

Boost.Asio and Boost.Beast are highly portable, being primarily header-only and depending only on Boost.System (and optionally OpenSSL). They compile cleanly with modern GCC, Clang and MSVC on Linux, Windows, macOS and many embedded environments. Poco, by contrast, is a modular library with various external components (OpenSSL, ODBC, etc.), which can complicate its build and configuration. Nonetheless, each Poco module remains independent, so unused dependencies need not be included. All three libraries support recent C++ standards and integrate smoothly with common build systems. In terms of startup speed and minimal external dependencies, Asio and Beast lead; Poco.Net offers a broader ecosystem at the cost of more involved setup.

All three libraries benefit from active ecosystems. Boost.Asio's repository exceeds 2,300 commits by around twenty core contributors. Boost.Beast has over 2,400 commits from 143 participants. Support is primarily via the Boost mailing lists and GitHub issues, with new Boost releases typically every four months. The Poco.Net include commits from almost 100 authors and more than 1,300 commits. Releases occur five to six times per year, announced on the official blog. Community engagement is available via GitHub, mailing lists and the blog. These repository statistics (Table 10) informed scoring of the Community Support dimension in the comparison framework (Table 8).

Table 10. GitHub repository metrics

Metric	Boost.Asio	Boost.Beast	Poco.Net
Commits	2,314	2,469	1,306
Contributors	19	143	97
Open issues	36	92	120
Releases frequency	4 months	4 months	2-2.5 months

Source: created by the authors

These results were subsequently considered in the context of prior research on software library evaluation and benchmarking. H. Anzt *et al.* (2019) proposed an automated benchmarking infrastructure for high-performance computing software. Their framework supports continuous performance testing and reproducibility by integrating benchmark execution, environment capture and visualisation via a web interface. The tool, originally developed for the Ginkgo library, demonstrates how such methodologies can contribute to sustainable scientific software by enabling community-driven performance monitoring.

Several works have also explored multiperspective and multicriteria evaluation models. K. Dawood *et al.* (2023) addressed the challenge of selecting open-source software in environments with heterogeneous user roles. They proposed a usability evaluation framework that integrates the Best Worst Method (BWM) and Group VIKOR techniques for weighting and ranking alternatives, validating their approach through a real-world case study. E. Denisova *et al.* (2024) introduced a reproducible usability testing framework tailored to medical software, with a focus on ensuring comparability across similar platforms. Their methodology, which

includes standardised tasks and objective performance measurements, reinforces the importance of formalised, domain-specific evaluation strategies. Although these studies cover a diverse range of software ecosystems, from web development to medical imaging and simulation, they collectively underscore the benefits of using formal methodologies and quantifiable criteria in the comparison of software libraries. Several works (Petty *et al.*, 2015; De La Mora & S. Nadi, 2018; Loja & Maita, 2024) also emphasise the importance of usability and performance metrics, the involvement of multiple stakeholder perspectives, and the potential for automation and reproducibility in evaluation processes.

The present work builds on these principles by introducing a structured framework for the empirical comparison of networking libraries, which is then applied to C++ solutions. M. Che & M. Tuo (2016) analysed asynchronous HTTP server designs with Winsock to avoid blocking and enhance responsiveness, while D. Mitrović *et al.* (2015) presented a scalable webagent middleware for automatic load balancing and fault tolerance. B. Amirkhanov *et al.* (2025) compared MQTT over TCP, MQTT over WebSocket, and HTTP in digital twin applications, emphasising the trade-offs between latency and stability. Finally, R. El-Hajj & S. Nadi (2020) introduced LibComp, an IntelliJ plugin enabling metricbased, inIDE comparison of Java libraries. F. Lu *et al.* (2019) observed that MongoDB's reliance on Boost.Asio, while beneficial for portability, introduced performance limitations under data-intensive workloads – limitations that were addressed by replacing Asio-based networking with a high-performance RDMA-capable transport layer. These studies underscore the versatility of Boost.Asio across domains but also highlight the need for a more systematic performance-oriented evaluation.

Several aspects of the findings resonate with and extend insights from both domain-specific benchmarking efforts and broader methodological research. Prior studies have emphasised the value of structured, metric-based evaluation frameworks in guiding software library selection across diverse contexts. For example, F. De La Mora & S. Nadi (2018) demonstrated how repository analytics can support informed library choice, highlighting the significance of popularity, security and performance indicators. M. Petty *et al.* (2015) proposed a formal assessment methodology for simulation frameworks, arguing that well-defined criteria are essential for objectivity and replicability. Multi-criteria evaluation models have also been introduced to address stakeholder diversity, such as the BWM and Group VIKOR integration by K. Dawood *et al.* (2023).

The results obtained in this study are in line with current trends in the use of smart technologies and modelling in the field of energy and network management. In particular, T. Nechaieva *et al.* (2025) emphasised the importance of comprehensive modelling of energy balances of local communities, taking into account environmental factors and the need to reduce greenhouse gas emissions. Their approach to building sustainable and decentralised energy systems complements the practical focus of this work's framework on flexibility and reuse. In turn, O. Turchyn (2025) demonstrated the effectiveness of using neural networks in optimising the management of technical systems, particularly in the oil and gas sector. The introduction of deep learning for adaptive control of sucker-rod pump installations confirms the relevance of using modern approaches to reduce energy consumption and increase efficiency, which also echoes this research findings on the resource stability of various libraries.

The empirical results obtained through the proposed framework align with earlier observations in the C++ networking domain while offering a more unified and reproducible basis for comparison. For instance, E. Kadusic *et al.* (2022) observed that Boost.Asio demands more manual buffer management compared to Poco.Net's higher-level abstractions. The hybrid server paradigms examined by Y. Pasichnyk (2022) and D. Butynets (2023) – which blended threaded and event-driven approaches – suggested performance gains under sustained stress; this was corroborated by the six-thread benchmarks. Finally, cross-language work by C. Pflugfelder (2022) highlighted C++/Asio's low end-to-end latency at the cost of code complexity – an observation mirrored in the latency profiles. Through this unified, empirical comparison, the study confirms earlier insights while providing the comprehensive, side-by-side performance evaluation of Boost.Asio, Boost.Beast and Poco.Net in production-style C++ applications.

The comparative results demonstrate that the proposed framework is capable of capturing both measurable performance characteristics and more qualitative aspects such as developer ergonomics and ecosystem maturity. By enforcing interface uniformity, and isolated testing environments, the framework enables meaningful side-by-side comparisons across different architectural paradigms and abstraction levels. This not only validates the methodology against existing benchmarking approaches in the literature, but also suggests its applicability beyond the C++ ecosystem to other networking stacks and programming environments.

Several directions exist for extending the present work. While the current evaluation was conducted within Windows 11 environment and a single Ubuntu 22.04.5 LTS guest, replicating the study across alternative operating systems, hardware configurations, and resource-constrained embedded environments could improve generalisability. Incorporating a broader set of HTTP benchmarking tools – such as wrk or JMeter – and varying request patterns (e.g., mixed GET/POST, chunked transfers) may help reveal additional behavioural traits under diverse conditions. Likewise, expanding the streaming scenario beyond Conway's Game of Life, by employing more demanding real-time protocols (e.g., QUIC, gRPC) or larger message formats, could provide further insights into performance and

resource trade-offs. Evaluating the impact of SSL/TLS handshake overheads, certificate validation delays, and network-level anomalies (e.g., packet loss or network jitter) could enrich the reliability analysis and inform security-sensitive use cases. Lastly, integrating formal metrics for code complexity, learning curve, and team adoption would complement the current qualitative assessment and support more rigorous comparison from a software engineering perspective.

CONCLUSIONS

This study conducted a structured comparison of three widely used C++ networking libraries – Boost.Asio, Boost.Beast, and Poco.Net – based on a unified evaluation approach. By encapsulating each candidate behind a uniform client-server interface and enforcing build-time selection, the methodology ensured that all non-networking logic remained invariant, enabling a trusted evaluation across performance, resource usage, and qualitative dimensions. The evaluation of network libraries from the perspective of not only performance indicators – such as throughput and latency – but also functionality, development complexity, reliability, and other relevant aspects, contributed to a more holistic and thorough analysis.

Applying this approach within two distinct C++ prototypes – a RESTful HTTP service and a real-time multicast streaming application – revealed clear trade-offs among Boost.Asio, Boost.Beast and Poco.Net. In the HTTP scenario, Boost.Asio delivered the highest request throughput and lowest average and tail latencies under modest threading configurations. Boost.Beast's higher-level framing engines delivered slightly lower performance (on average, up to 20% lower). Poco.Net, while generally trailing in raw throughput, exhibited the most stable hig-hpercentile latency characteristics and the smallest memory footprint, consuming less than a quarter of the working set required by its peers when serving equivalent loads. In the GameOfLifeStreaming prototype, neither Asio nor Poco.Net gained a decisive multicast-related advantage in end-to-end latency, indicating that the cost of managing large numbers of concurrent clients often outweighs the savings from UDP group transmission; Boost.Beast's WebSocket-based model occasionally matched or outperformed its UDP counterparts in the testbed.

Across both applications, CPU utilisation remained well within predictable bounds and showed no evidence of progressive degradation or leakage. Memory consumption in ResourceMonitor was markedly higher for the Asio and Beast implementations, reflecting their more granular buffer management and asynchronous contexts, whereas Poco.Net's synchronous architecture yielded consistently lower working and private working sets. In the GameOfLifeStreaming prototype, the Boost.Beast variant consumed significantly more resources due to the overhead of maintaining individual WebSocket sessions for each connected client.

The architecture and API of each library directly shaped these behaviours: Asio's minimal abstraction offered maximum flexibility at the cost of boilerplate and potential for misuse; Beast struck a middle ground by providing protocol-specific constructs atop Asio; and Poco.Net encapsulated broad protocol support behind a high-level, synchronous interface, trading off some performance for rapid development and ease of use. When evaluated using the qualitative scoring framework – covering modularity, documentation quality, extensibility, ergonomics and ecosystem maturity – Poco.Net achieved the highest total (20 points), followed by Beast (18.5 points) and Asio (17 points). Taken together, these findings suggest that no single library is universally "best". Instead, Asio is most appropriate where finegrained control and minimal runtime overhead are paramount, Beast is well-suited to HTTP or WebSocketcentric workloads that benefit from protocol-aware abstractions, and Poco.Net is favourable when development speed, multi-protocol support and constrained memory budgets prevail. The ability to swap libraries simply by toggling a buildtime flag, without any alteration to application code, proved invaluable for conducting this side-by-side analysis.

Looking forward, the comparative approach may be extended in several directions. Inclusion of additional protocols (for example, QUIC or gRPC), native SSL/TLS stacks, and alternative concurrency models will broaden its applicability. Evaluating across Linux, macOS and embedded environments will enhance generalisability, while more varied traffic patterns and security-focused tests will deepen understanding of each library's real-world robustness. Finally, integrating automated measures of developer productivity, code complexity and maintainability promises to complement the performance-driven insights with human-centred metrics, thereby closing the loop on fully rounded library evaluation.

ACKNOWLEDGEMENTS

None.

FUNDING

None.

CONFLICT OF INTEREST

None.

REFERENCES

[1] Amirkhanov, B., Amirkhanova, G., Kunelbayev, M., Adilzhanova, S., & Tokhtassyn, M. (2025). Evaluating HTTP, MQTT over TCP and MQTT over WEBSOCKET for digital twin applications: A comparative analysis on latency, stability, and integration. *International Journal of Innovative Research and Scientific Studies*, 8(1), 679-694. doi: 10.53894/ijirss.v8i1.4414.

[2] Anzt, H., Chen, Y.-C., Cojean, T., Dongarra, J., Flegar, G., Nayak, P., Quintana-Ortí, E.S., Tsai, Y.M., & Wang, W. (2019). Towards continuous benchmarking: An automated performance evaluation framework for high performance software. In *PASC 2019: Proceedings of the platform for advanced scientific computing conference* (article number 9). New York: Association for Computing Machinery. doi: 10.1145/3324989.3325719.
[3] Ardarve, J., Finne, R., Hrustic, A., & Svennungsson, J. (2019). *Enforcing privacy requirements on oblivious network agents - a software solution of a type system generated from COPPA*. (Bachelor's thesis, Chalmers University Of Technology, Gothenburg, Sweden).
[4] Baddi, Y., Sebbar, A., Zkik, K., Maleh, Y., Bensalah, F., & Boulmalf, M. (2023). MSDN-IoT multicast group communication in IoT based on software defined networking. *Journal of Reliable Intelligent Environments*, 10, 93-104. doi: 10.1007/s40860-023-00203-x.
[5] Butynets, D. (2023). *Performance assessment of the different approaches to implementing network servers using C++ language*. (Bachelor's thesis, Ukrainian Ctholic University, Lviv, Ukraine).
[6] Che, M., & Tuo, M. (2016). Server program analysis based on HTTP protocol. *MATEC Web of Conferences*, 63, article number 05023. doi: 10.1051/matecconf/20166305023.
[7] Dawood, K.A., Zaidan, A.A., Sharif, K.Y., Ghani, A.A., Zulzalil, H., & Zaidan, B.B. (2023). Novel multi-perspective usability evaluation framework for selection of open source software based on BWM and group VIKOR techniques. *International Journal of Information Technology & Decision Making*, 22(1), 187-277. doi: 10.1142/s0219622021500139.
[8] De La Mora, F.L., & Nadi, S. (2018). An empirical study of metric-based comparisons of software libraries. In *Proceedings of the 14th international conference on predictive models and data analytics in software engineering* (pp. 22-31). New York: Association for Computing Machinery. doi: 10.1145/3273934.3273937.
[9] Denisova, E., Tiribilli, E., Luschi, A., Francia, P., Manetti, L., Bocchi, L., & Iadanza, E. (2024). Enabling reliable usability assessment and comparative analysis of medical software: A comprehensive framework for multimodal biomedical imaging platforms. *Health and Technology*, 14, 671-682. doi: 10.1007/s12553-024-00859-2.
[10] El-Hajj, R., & Nadi, S. (2020). LibComp: An IntelliJ plugin for comparing Java libraries. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 1591-1595). New York: Association for Computing Machinery. doi: 10.1145/3368089.3417922.
[11] Kadusic, E., Zivic, N., Hadzajlic, N., & Ruland, C. (2022). The transitional phase of Boost. Asio and POCO C++ networking libraries towards IPv6 and IoT networking security. In *2022 IEEE international conference on smart internet of things (SmartIoT)* (pp. 80-85). Suzhou: IEEE. doi: 10.1109/smartiot55134.2022.00022.
[12] Kayum, S.N., Alsalim, H., Tonellot, T., & Momin, A. (2020). A fault tolerant implementation for a massively parallel seismic framework. In *2020 IEEE high performance extreme computing conference (HPEC)* (pp. 1-8). Waltham: IEEE. doi: 10.1109/hpec43674.2020.9286143.
[13] Loja, A., & Maita, T. (2024). Comparative evaluation of performance efficiency in terms of temporal behavior and resource utilization, according to the ISO/IEC 25,010 model, in a web application developed with Angular, React.js, and Vue.js. In G.F. Olmedo Cifuentes, D.G. Arcos Avilés & H.V. Lara Padilla (Eds.), *Emerging research in intelligent systems. CIT 2023. Lecture notes in networks and systems* (pp. 293-308). Cham: Springer. doi: 10.1007/978-3-031-52255-0_21.
[14] Lu, F., Fang, T., Zhang, Z., Li, S., Chen, J., An, H., & Han, W. (2019). Improving the performance of MongoDB with RDMA. In *IEEE international conference on high performance computing and communications (HPCC)* (pp. 1004-1010). Zhangjiajie: IEEE. doi: 10.1109/hpcc/smartcity/dss.2019.00144.
[15] Mitrović, D., Ivanović, M., Vidaković, M., & Budimac, Z. (2015). A scalable distributed architecture for Web-Based software agents. In M. Núñez, N. Nguyen, D. Camacho & B. Trawiński (Eds.), *Computational collective intelligence. Lecture notes in computer science* (pp. 67-76). Cham: Springer. doi: 10.1007/978-3-319-24069-5_7.
[16] Nechaieva, T., Teslenko, O., Trokhaniak, V., & Makarevych, S. (2025). Modeling energy balances of a community under increased energy independence and greenhouse gas reduction. *Machinery & Energetics*, 16(1), 104-116. doi: 10.31548/machinery/1.2025.104.
[17] Pasichnyk, Y. (2022). *Performance analysis of synchronous and asynchronous parallel network server implementations using the C++ language*. (Bachelor's thesis, Ukrainian Ctholic University, Lviv, Ukraine).
[18] Petty, M.D., Kim, J., Park, S., & Lee, S. (2015). A methodology for quantitative assessment of the features and capabilities of software frameworks for model composition. *International Journal of Modeling, Simulation, and Scientific Computing*, 7(1), article number 1541002. doi: 10.1142/s1793962315410020.
[19] Pflugfelder, C. (2022). *Asynchronous programs on IoT devices*. (Bachelor's thesis, Institute for Formal Methods of Computer Science, Stuttgart, Germany).
[20] Sai, P.C., Karthik, K., Prasad, K.B., Pranav, C.V.S., & Divya, K.V. (2024). Real-time task manager: A Python-based approach using Psutil and Tkinter. In *2024 8th international conference on computational system and information technology for sustainable solutions (CSITSS)* (pp. 1-6). Bengaluru: IEEE. doi: 10.1109/csitss64042.2024.10816758.

[21] Samoydiuk, A., & Ostapchuk, O. (2024). Optimizing high-load systems with asynchronous programming techniques. In *Modeling, control and information technologies: Proceedings of VII international scientific and practical conference* (pp. 130-131). Rivne: National University Of Water And Environmental Engineering. doi: 10.31713/mcit.2024.036.
[22] Turchyn, O. (2025). Introduction of neural network technologies to optimise control of suckerrod pump installation. *Machinery & Energetics*, 16(1), 32-42. doi: 10.31548/machinery/1.2025.32.

Фреймворк для порівняльного аналізу мережевих програмних бібліотек та його застосування до мережевих рішень C++

Єгор Грушевий

Магістр
Київський національний університет імені Тараса Шевченка
01033, вул. Володимирська, 60, м. Київ, Україна
https://orcid.org/0009-0005-7695-1990

Костянтин Жереб

Кандидат фізико-математичних наук, асистент
Київський національний університет імені Тараса Шевченка
01033, вул. Володимирська, 60, м. Київ, Україна
https://orcid.org/0000-0003-0881-2284

Анотація. Об'єктивне порівняння мережевих бібліотек залишається актуальним завданням у програмній інженерії, адже відсутність уніфікованих підходів та залежність від специфіки застосунків часто ускладнюють виявлення важливих відмінностей у продуктивності та зручності використання. Метою дослідження було здійснення комплексного порівняння трьох мережевих бібліотек C++ (Boost.Asio, Boost.Beast та Poco.Net) на основі запропонованого універсального фреймворку. Методологія дослідження охоплювала розробку універсального порівняльного фреймворку, експериментальну реалізацію програмних прототипів, проведення навантажувального тестування для збору кількісних показників та здійснення структурованої якісної оцінки за рядом визначених критеріїв. Створений фреймворк включає збір як кількісних (пропускна здатність, затримка, споживання ресурсів), так і якісних показників (зручність інтерфейсу прикладного програмування, функціональність, стабільність, кросплатформність, складність розробки). Особлива увага приділялась поведінці в умовах однакового програмного середовища. З використанням кожної з досліджуваних бібліотек, у межах роботи реалізовано два клієнт-серверні прототипи: RESTful-сервіс ResourceMonitor і застосунок з потоковим передаванням даних GameOfLifeStreaming. Варіанти реалізації уніфіковано в структурі проєкту, що забезпечило мінімізацію відмінностей на рівні логіки та дозволило сфокусуватись на властивостях самих бібліотек. Boost.Asio продемонстрував переваги у задачах із високими вимогами до затримок і необхідністю низькорівневого контрою. Boost.Beast виявився достатньо ефективним вибором для HTTP-застосунків, але з обмеженим переліком підтримуваних протоколів. Poco.Net найефективніше використовував ресурси, не сильно деградував при великих навантаженнях, дозволяє працювати з найбільшою кількістю протоколів, але показав гірші показники затримок і пропускної здатності, а також потребував складнішої конфігурації. Практична цінність роботи полягає у можливості повторного використання фреймворку для порівняння інших мережевих бібліотек, інтеграції створених компонентів в інші розробки, а також у сформованих висновках щодо доцільності застосування кожної з бібліотек

Ключові слова: оцінка продуктивності застосунків; навантажувальне тестування; клієнт-серверна архітектура; збір якісних і кількісних метрик; конфігурація тестового середовища; модульна архітектура; вимірювання пропускної здатності і затримок