LibBench: An Open Source Framework To Benchmark Game Networking Libraries

Pelin Kuran
2821141
p.r.kuran@student.vu.nl
Faculty of Science, Vrije Universiteit Amsterdam

Kaustav Kumar Choudhury
2801147
k.k.choudhury@student.vu.nl
Faculty of Science, Vrije Universiteit Amsterdam

Roy de Regt
2647734
r.b.de.regt@student.vu.nl
Faculty of Science, Vrije Universiteit Amsterdam

Patrycja Stępien
2828422
p.a.stepien@student.vu.nl
Faculty of Science, Vrije Universiteit Amsterdam

Thijs van der Heijden
2824561
t.a.j.van.der.heijden@student.vu.nl
Faculty of Science, Vrije Universiteit Amsterdam

Pragya Tyagi
2798423
p.tyagi@student.vu.nl
Faculty of Science, Vrije Universiteit Amsterdam

***Abstract*—A large portion of modern video games can or must be played online, providing the user interactivity with other players. Over the last decades, many game networking libraries have been developed. When developing a new game, the developer may be overwhelmed with options in terms of networking libraries alone. In this paper, we aim to provide developers data to make informed decisions on their networking library choice.**

To this end, we develop a basic video game, ToyGame, and implement three commonly used networking libraries: Raknet, Enet and Yojimbo. We benchmark the libraries and compare their performance. We also contribute an open source framework, LibBench, that others can use to benchmark networking libraries.

***Keywords*—** Online Gaming, Game Networking Libraries, Raknet, ENet, Yojimbo

I. Introduction

Nowadays it is impossible to imagine the world of video games without online multiplayer. Online video games bring people together–even when miles apart. The incorporation of online multiplayer in a video game presents numerous benefits for players; however, it also introduces new challenges for the development team. Unlike single player and local multiplayer games, online games have to deal with all the complexity that comes with a distributed system. Developers are suddenly also faced with the development, maintenance, and control of game servers.

Massively multiplayer online games gained traction over the last decades, and battle royale [1] games have become popular much more recently. These games can have up to a hundred players in one session. Such games require proper tools to tackle the issues that are bound to arise from hosting and catering to tens of thousands of interconnected systems across the globe. A game networking library provides game developers a ready implementation of network protocols [4]. The developers implement the library, rather than implementing low-level network communication. Over the last decades, as online multiplayer has evolved into the mature state it is in now, many such game networking libraries have been created [4]. While these libraries are easing the lives of game developers, new questions arise: since many options are available, which should the developer choose? Does the choice of library impact performance, on client side or server side?

To help game developers make informed decisions regarding the choice of game networking library, we conducted benchmarks of three such libraries. The libraries we implemented are SLikeNet [6] (maintained fork of RakNet [5]), ENet [2], and yojimbo [7]. To this end, we created a basic video game, ToyGame, implementing multiplayer using the three libraries. Additionally, we developed an open-source framework named LibBench [3], intended to serve as a tool for the game development community to benchmark any other game networking library. The framework can be extended with new libraries, and allows the user to run tests under various networking conditions. The results of the tests can be automatically visualized.

II. Background

This lab report is focused on evaluating and comparing various game networking libraries, with a specific emphasis on their scalability and consistency, two critical performance metrics in game development. The development of multiplayer games has seen a significant evolution, driven by advances in networking technology and an increasing demand for seamless, real-time interactions in a virtual environment. The choice of a networking library is a crucial decision in this development process, as it directly impacts the game's performance, player experience, and overall success.

Scalability refers to the ability of a game networking library to handle a growing amount of work or its potential to accommodate growth. In the context of multiplayer games, scalability is essential for managing an increasing number of simultaneous players, ensuring the game remains stable and responsive regardless of the load. This aspect of performance is particularly relevant in games with large player bases or those aspiring for such growth.

Consistency, on the other hand, pertains to the uniformity and reliability of the gaming experience across different players' sessions. In multiplayer environments, consistency is crucial for maintaining fairness and synchronization among players, regardless of their individual network conditions. This includes the accurate and timely update of game states and the synchronization of player actions.

In this report, we analyze various game networking libraries, which are fundamental tools in building the network architecture of multiplayer games. These libraries provide the framework and protocols for data transmission between the client and server. For our evaluation, we create a basic video game. The video game supports a set of game networking libraries, such that the user can configure which library to use. We also provide a framework to launch the server and n different clients. Any subset of clients can be configured to have a certain network latency, packet loss and/or network bandwidth.

III. System Design

For each game instance, there is one central game server III-C and upto n game clients (see Section III-B) that connect to it, as is depicted in figure 1. Each client connects to the game server, and exchanges state with the server. We want to test the behavior of each network library (see Section III-A), under various realistic circumstances. Our experimental setup is used in a single computing cluster, and is not connected via remote connections. Therefore, we instrument the clients and server with tools to emulate realistic network behavior. The system is developed primarily using C-language, along with bash scripts for logging system and network metrics for client as well as server processes.

A. Networking libraries

When preparing the benchmark, we chose a batch of game networking libraries to compare. We selected them based on the set of functional and non-functional requirements that have been listed at the end of this section. From them, we selected libraries that have distinct features, which distinguish them from others. We ended up with the following libraries.

- **SLikeNet** - SLikeNet, unlike other benchmarked libraries, provides extensive support in mitigating various network failures. It automatically orders packages that arrive out of order and re-transmits messages that did not arrive. It also protects transmitter data and informs if the data was externally changed. We were interested in examining additional overhead generated by those features and the influence they have on the game latency and performance.
- **Enet** - Enet is a general-purpose library that is widely used for distributed systems, especially for games as it was mainly developed for a multiplayer shooter game called Cube. The main purpose of this library is to provide a relatively thin, simple, and robust networking library. Enet provides a transient synchronized communication protocol, and it was initially designed for a shooter game, therefore, we wanted to explore this library more in-depth.
- **Yojimbo** - Yojimbo is dedicated to competitive multiplayer games, such as first-person shooter games. We wanted to check if creating a game networking library with a targeted focus on supporting fast-paced gameplay is reflected in the performance of such a game.

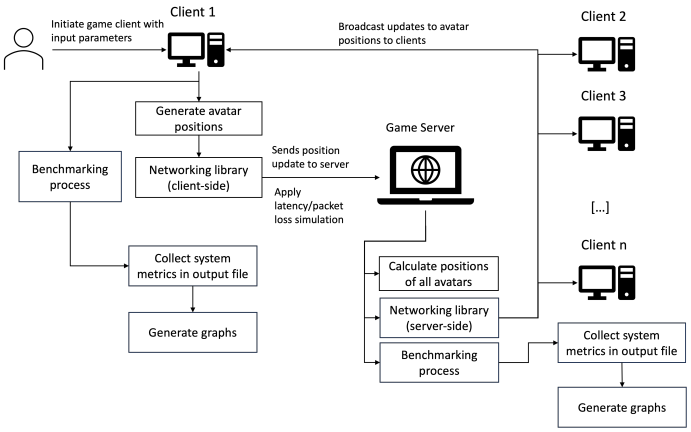

Fig. 1: ToyGame system model.

B. Game client

The game clients, along with the game server, run a very simple game. The player is spawned in an N by M sized area, and is allowed to move one space in any of the cardinal directions every game tick. Two players can't be in the same spot at once, which is enforced by the game server.

Every game tick the player chooses a random direction and proceeds to move one space in that direction. This new position is sent to the server, which checks the validity of all new positions and then broadcasts the updated positions for

all players.

C. Game server

The game server receives an updated player position from all connected clients, processes these updates, then broadcasts the updated player positions to all clients. While processing the updated positions, the server performs two important checks, namely; making sure no two players move to the same location and making sure no player moves more than one space in a single game tick.

The requirements that apply to the complete system (both client and server systems as well as general requirements pertaining to the networking libraries and system design) are listed below.

Non-functional requirements

- **NFR1** *Open source.* The source code (server, clients, testing framework) used to run the experiments is available as an open-source project. Apart from our implementation, the game networking library should be open-source, so that it is possible to reproduce our experiments.
- **NFR2** *Packet loss.* Clients do not disconnect when some packet loss occurs.
- **NFR3** *Fast updates.* Clients get updates from the server in a reasonable time, such that it feels direct for the player.
- **NFR4** *Play with latency.* Clients can connect to the server and play the game, even when there is a relatively high latency.
- **NFR5** *Popular.* We want to compile a comparison of libraries widely used among developers to make our benchmark relevant and help programmers make informed decisions.
- **NFR6** *Similar Behaviour Model* We want to select libraries that use similar architecture and behavior for data transferring to be able to propose consistent and fair experiments.
- **NFR7** *Fast processing.* The server can process updates from all clients in reasonable time, such that the response feels direct for the player, in normal circumstances.

Functional requirements

- **FR1** *Networking libraries.* Clients and servers support at least three different game networking libraries.
- **FR2** *Local game state.* Clients maintain a local game state.
- **FR3** *Push game state.* Clients push their local game state at a configurable frequency.
- **FR4** *Correct game state.* Clients correct their invalid game state when the server pushes a corrected global game state.
- **FR5** *Log metrics.* Clients track and log CPU, Memory and Network usage metrics.
- **FR6** *Configure client.* Clients accept arguments or configuration files to alter the following emulated parameters: latency, restricted bandwidth, packet loss.
- **FR7** *Togglable Reliable Messaging.* Client can turn on and off reliable messaging.
- **FR8** *Run headless.* Clients can run headless.
- **FR9** *Documentation* As the purpose of a game networking library is to accelerate and ease the development process, its documentation should allow one to write code efficiently, ideally without studying the source.
- **FR10** *C# or C++ language support* Selected library should support either C++ or C#, or expose an API in one of those languages, as they are one of the most popular game development languages.
- **FR11** *Linux* We decided to choose a library with C++ support, and ensure compatibility on a Linux-based system.
- **FR12** *Global game state.* The server maintains a global game state.
- **FR13** *Allow valid states.* The server only accepts valid game states.
- **FR14** *Log metrics.* The server tracks and logs CPU, Memory and Network usage metrics.
- **FR15** *Togglable Reliable Messaging.* Server can turn on and off reliable messaging.

IV. Implementation

To develop a benchmarking library for the systematic evaluation of selected networking libraries, we introduced a benchmark game implemented separately for each library. The standardized game is, cross-platform, headless, and written in C++. To ensure consistency and fairness in our evaluation we implemented features of the game uniformly across each networking library. The libraries were built from source, sourced from their established GitHub repositories[2][6][7], CMake was employed for the build process of these libraries.

The development included two distinct executables, one for the server and another for clients, for each networking library implementation. The experimentation pipeline, from compilation to execution, was automated using Bash scripts.

1) Client Implementation: On the client side, the game loop initiates by updating the position of game entities. This updated state is then transmitted to the server, where it is essential for maintaining a synchronized game state across all clients. Upon sending the updated positions, the client anticipates a response, which comprises game state data that includes the positions of other entities within the game world, ensuring the local game state remains current and accurate. The client-side loop concludes with a benchmarking process that likely serves to validate the consistency and reliability of the received data against the game's internal state.

2) Server Implementation: The server side is tasked with receiving position updates from clients and subsequently

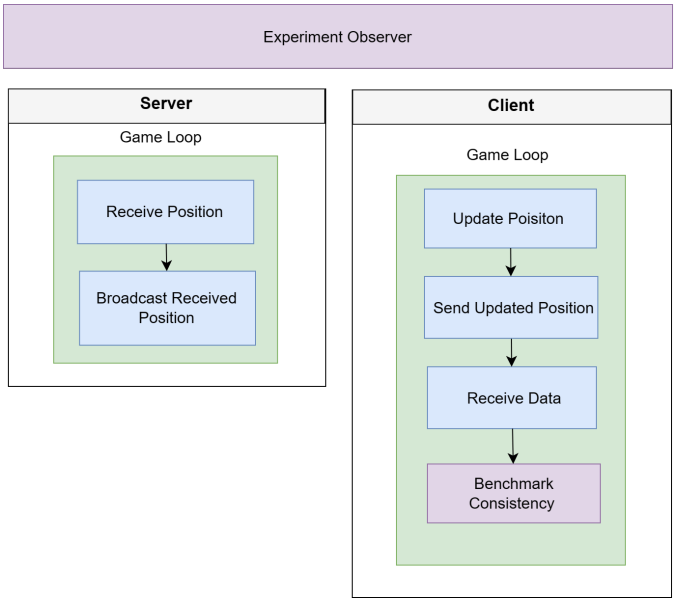

Fig. 2: Game Components

TABLE I: Tools

Purpose	Tool	Version
Build	Cmake	3.27
Programming Language	C++	20
Automation	Bash Scripts	-
Network Simulation	Clumsy	0.3
Network Simulation	Dummynet	1.0

broadcasting this information to all connected clients. This mechanism ensures that each client has a coherent and up-to-date understanding of the game world, which is crucial for gameplay continuity and integrity.

A key focus was placed on maintaining a uniform messaging structure across all implementations to ensure fair and precise cross-library comparisons. During development we used Github for source control and it is made open source [3]. For measuring corrections and response of the library under unstable network conditions, we simulated network conditions using third-party applications such as Clumsy, Dummynet. List of tools and applications may be found in Table I. Figure 2 represents the flows within components, a client and a server within a networked game architecture.

V. Experimental Setup

A. Baseline Library Comparison

An important factor when choosing a game networking library is the overhead the library adds, in both consumed hardware and network resources. A baseline benchmark comparison between libraries gives insight into the overhead a library adds. Multiple important metrics are measured to enable an accurate and reliable comparison between libraries.

Hardware Metrics

1) *CPU Time (Seconds)* The total CPU time consumed by the process.
2) *RSS (Resident Set Size) (KB)* The total RAM being consumed by the process and its linked dynamic libraries.

Network Metrics

1) *Packet Count* The total number of packets sent between clients and server, in both directions.
2) *Total Bandwidth (Bytes)* Total bandwidth consumed.
3) *Average Bandwidth/Second (Bytes/Second)* Average bandwidth consumed per second (total bandwidth / benchmark duration).

Measuring the network metrics is done via a simple script. First the script starts up tshark (Wireshark CLI), logging all traffic to and from port 60000 (server port) to a CSV file. Second, the script starts the game server, after which four game clients are started. Third, the script waits for 60 seconds while the clients and server communicate and play the game. Finally, the script kills all game processes and terminates the tshark logging process, a simple Python script then parses the resulting CSV file, counting the number of packets and total bandwidth.

Measuring the hardware metrics is a little more involved. For this, a similar script was written, which starts the game server and clients, after which it calls the ps command 10x per tick, logging the CPU time and RSS metrics to a separate CSV file for every game process.

Reliable messaging is an important feature of many game networking libraries, but it can add large overheads. To be able to benchmark this, all clients and servers can toggle reliable messaging on and off. All benchmarks were run three times per library, and both with reliable messaging on and off.

B. Measuring Library Scalability

This experiment involves testing how well the libraries could handle more clients, and whether they could scale without a drop in performance. The process involves gradually increasing the number of clients from a baseline value of 4. The same scripts for hardware and network metrics measurement are used, while the number of clients is increased to 8, then 16, and then finally to 32.

C. Game Consistency

With this experiment we aim to detect inconsistencies in the game states, across the clients and the server, by comparing consecutive positions of entities, assuming an entity would move 1 unit in each game tick. It considers factors like latency and packet loss that might affect the reliability of the messaging system.

As managing game state consistency and addressing issues related to network latency, packet loss, and other factors is a common practice in online multiplayer games, this experiment

was implemented on the client side by following algorithm 1, and without relying on external scripts. The provided pseudocode 1 represents the consistency benchmarking approach, where the position of a client is tracked through the variable 'prevPosition' standing for the previous position. The algorithm updates a corrections tracker, invoking methods 'updatesReceivedInTick' and 'correctionMade' based on received updates. It includes a conditional check to determine whether the difference between the tracked previous position and the current position exceeds a predefined threshold ('Maximum Position Difference'). If the condition is met, the corrections tracker acknowledges a correction. Additionally, the algorithm updates the position of the corresponding client with the received data. As in our headless game each entity could only move 1 unit in each tick if any of the clients moves more than 1 unit it is an indication of a correction need that is caused by the unstable network environment we create. We also measured how many ticks no messages were received as in normal conditions clients receive position updates in each tick.

By using third-party applications mentioned in Table I network conditions are simulated, such as latency and packet loss, allowing testing of how the game behaves under less-than-ideal network scenarios.

Algorithm 1 Correction Check

```
0: prevPosition ← positions[clientId]
0: correctionsTracker.updatesReceivedInTick()
0: if prevPosition − current position ≥
   Maximum Position Difference then
0:     correctionsTracker.correctionMade()
0: end if
0: m_others[data.ownerId] ← data =0
```

VI. Results

This section of the lab report presents the comprehensive data collected from our experiments listed in section V. Organized into three subsections, each subsection focuses on a specific aspect of the comparison metrics. Initially, the raw data obtained from the experiments is summarized in the form of bar graphs, providing a straightforward representation of the measurements and observations recorded. These visual aids are designed to facilitate a clear understanding of the experimental outcomes and are accompanied by brief descriptions highlighting key observations.

A. Baseline Library Comparison

Developers want their game clients to minimize resource consumption–the clients should perform well on many gaming devices–and networking libraries should therefore ideally satisfy all functional requirements without sacrificing performance and other non-functional requirements.

On the server side, there may be even more incentive to use minimize resource consumption as much as possible: since resources are typically hosted by the developer, the developer pays for those resources. Therefore, a game networking library that uses less computing resources than others on both client side or server side is more attractive to developers.

Developers may trade off one for the other: they may prefer to choose a library that uses more resources on the client side, but fewer resources on the server side, to reduce cost of running servers.

On the other hand, developers may prefer performance on the client side, especially when developing for devices with relatively minimal computing resources, such as mobile devices.

Our baseline shows that the ENet clients and servers consume less memory 4 6 and CPU cycles 35, where yojimbo clients and servers exchange fewer packets 7 and consume less bandwidth 8.

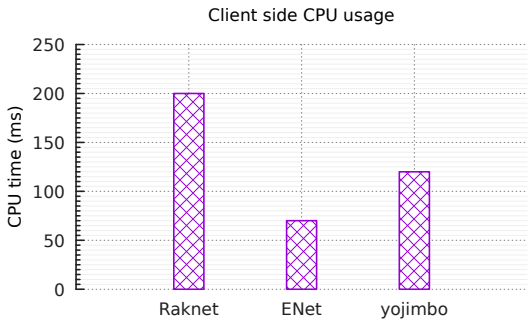

Fig. 3: CPU time of the game client for all three implemented libraries, measured over 10 seconds. ENet uses fewer CPU cycles than Raknet and yojimbo.

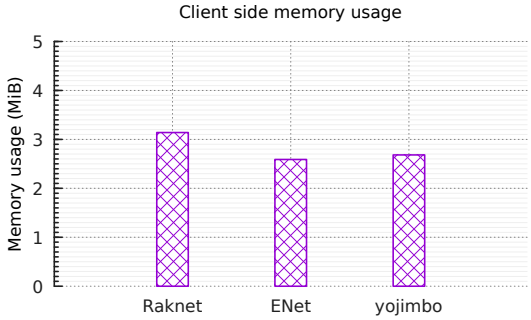

Fig. 4: Memory usage of the game client for all three implemented libraries, measured over 10 seconds. ENet consumes less memory than Raknet and yojimbo.

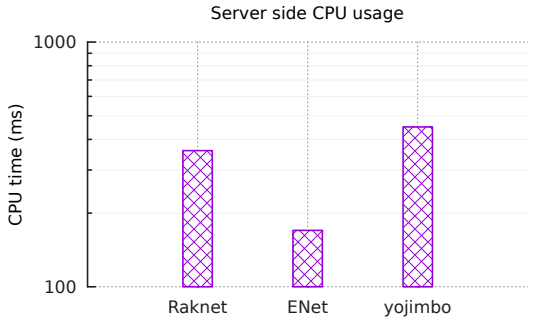

Fig. 5: CPU time of the game server for all three implemented libraries, measured over 10 seconds. ENet uses fewer CPU cycles than Raknet and yojimbo. Note that the scale is logarithmic.

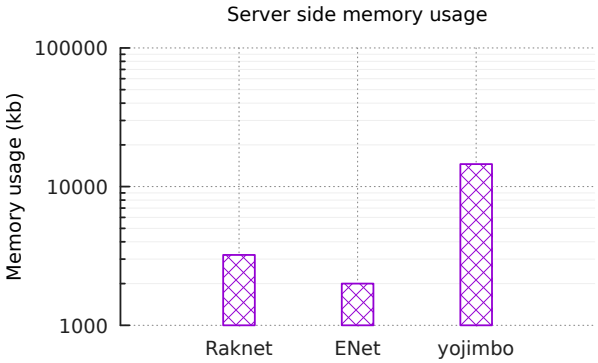

Fig. 6: Memory usage of the game server for all three implemented libraries, measured over 10 seconds. ENet consumes less memory than Raknet and yojimbo. Note that the scale is logarithmic.

B. Measuring Library Scalability

Measuring the scalability of a game networking library is crucial for assessing its performance under various conditions and determining its ability to handle an increasing number of players or connections. Game networking libraries can be used for a wide variety of games, ranging from 2 clients in a game like Street Fighter to 100 clients in games like PlayerUnknown's Battlegrounds (PUBG). However the number of clients is not guaranteed to be a particular number in every scenario, and it is not possible to switch libraries mid-session. Therefore, it is important for every networking library to be able to individually scale according to requirements. We decided to start from a baseline number of 2 clients and gradually increase the number by powers of 2 up to a maximum of 64 clients. We recorded the CPU time and memory usage on the client and server processes using both reliable and unreliable channels of communication for all 3 libraries. The metrics were measured

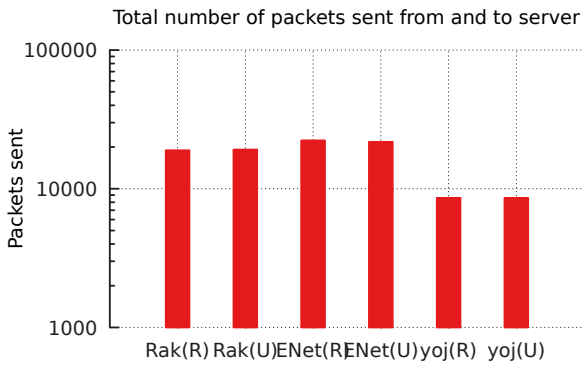

Fig. 7: Number of total packets sent from and to the server for all three implemented libraries, measured over 10 seconds. (R) and (U) indicate reliable or unreliable communication modes used. We observe that yojimbo uses less packets for both communication modes. Note that the scale is logarithmic.

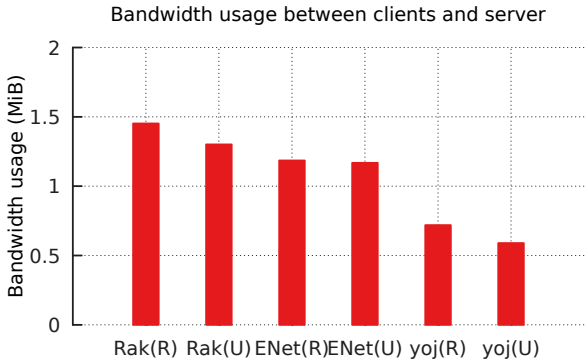

Fig. 8: Total bandwidth used between the server and all clients for all three implemented libraries. (R) and (U) indicate reliable or unreliable communication modes used. We observe that yojimbo has a lower bandwidth usage for both communication modes. Note that the scale is logarithmic.

on a MacBook Pro with 10 cores and 32 gigabytes of RAM. In Figure 9 CPU time utilized for all three implemented libraries by the server process is visualized. The time gradually increases for all libraries as expected, but there is a large jump in the case of Raknet. The reliable mode channel usually performs better except in the case of Yojimbo, where it is quite similar to the unreliable mode. Memory usage for all three implemented libraries represented in Figure 10. There is a noticeable jump in memory usage when going from 32 to 64 client processes in Raknet, and the unreliable mode consumes more memory here. There is a similar jump in the case of ENet, but the difference in memory consumption between the

reliable and unreliable modes is quite pronounced here in the case of 64 client processes. Yojimbo maintains roughly the same memory usage statistics throughout for both reliable and unreliable modes. This could suggest that Yojimbo servers are geared towards handling setups with more clients, since there is barely any difference between 2-client and 64-client experiments. In 11 the values display a slight decreasing trend until 16 clients and then shows a slight increase in the case of Raknet. Notably, ENet demonstrates the lowest CPU time usage among the three libraries, with the usage being minimal when handling 64 clients. However, in the case of 12 ENet, the reliable communication channel consumes more system memory than the unreliable mode, whereas the reverse holds true for ENet server memory usage. In contrast, akin to the server process memory usage, client memory usage remains relatively stable across an increasing number of clients for Yojimbo, which exhibits the lowest memory usage among the three.

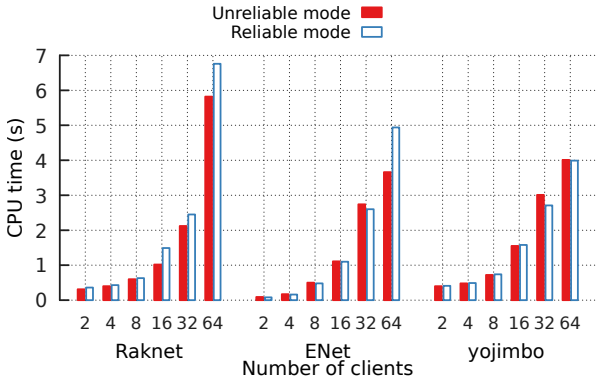

Fig. 9: CPU time utilized for all three implemented libraries by the server process.

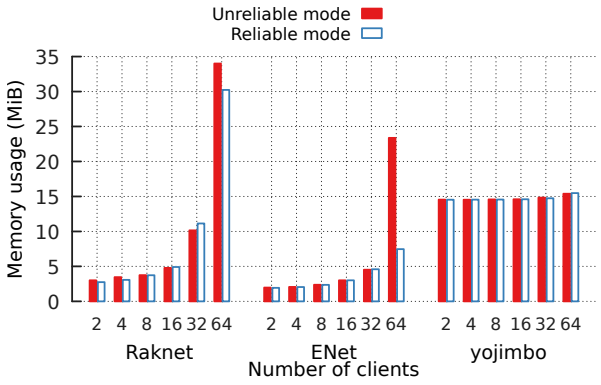

Fig. 10: Memory usage for all three implemented libraries by the server process.

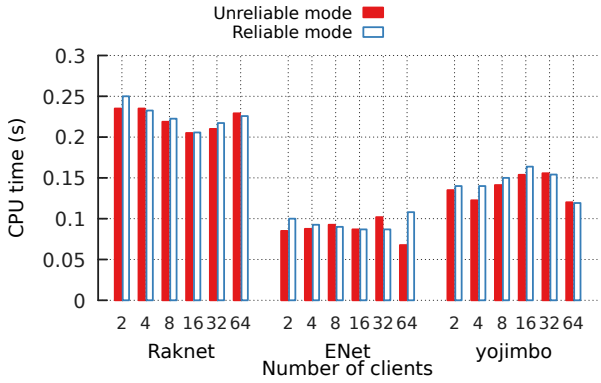

Fig. 11: CPU time utilized for all three implemented libraries by the client processes.

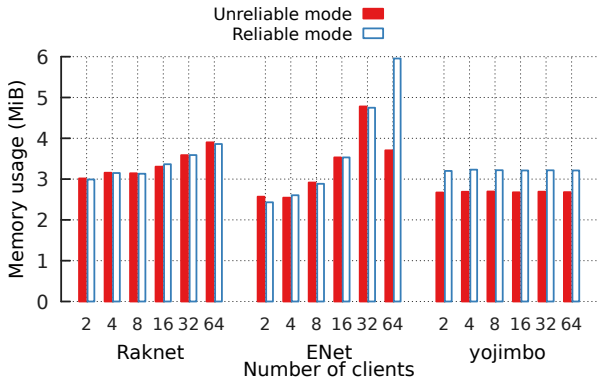

Fig. 12: Memory utilization among the three implemented libraries for client processes.

C. Game Consistency

Our last experiment is conducted as a proactive measure to identify and address issues that could impact the quality of online games, as managing game state consistency is crucial for delivering a seamless and fair gaming experience in online multiplayer games.

By implementing the consistency benchmarking approach on the client side, the experiment simulates real-world conditions and assesses the effectiveness of the algorithm in managing corrections in response to potential network instabilities. From our benchmarks we find out that Yojimbo does not make any corrections under volatile network conditions, this may indicate that all packages queued and sent in order under abnormal network conditions. For Enet and Raknet we observed a few corrections. However, it seems that the observation of yojimbo not making any corrections could be related to the implementation, especially considering the concurrent observation of certain ticks where no packages were received. In the comparison between RakNet and ENet, our observations revealed that both networking libraries implemented correction

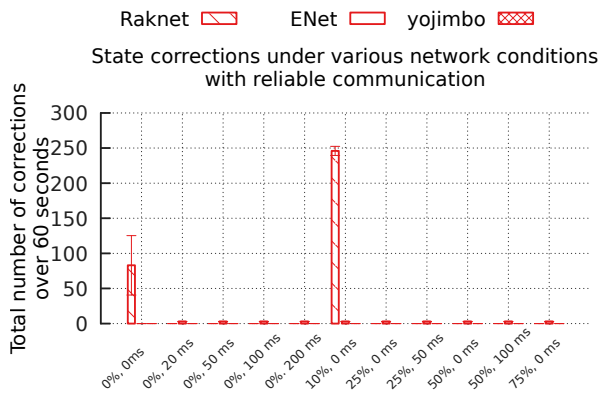

Fig. 13: Consistency Benchmark on Reliable Messaging

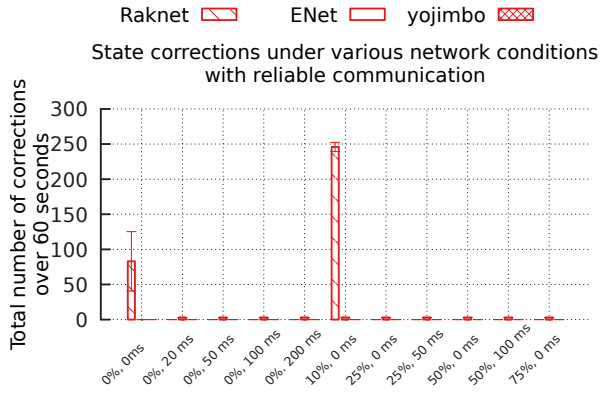

Fig. 14: Consistency Benchmark on Unreliable Messaging

mechanisms in response to inconsistencies in the game states. Notably, RakNet exhibited a slightly higher frequency of corrections compared to ENet in both reliable and unreliable messaging 13 14.

Additionally, the measurement of ticks with no received messages helps evaluate the game's robustness in handling disruptions, providing valuable information for game developers to optimize network-related features and enhance the overall gameplay experience. For all implemented libraries we observed some ticks with no messages and the results were similar for Enet and Yojimbo, yet Raknet had more ticks with no updates.

VII. Discussion and Future Work

The research we conducted has provided valuable insights into the performance and scalability of different game networking libraries. Our experiments highlighted key differences in resource consumption, scalability, and game consistency across the libraries RakNet(SLikeNet), ENet, and Yojimbo. Notably, ENet demonstrated efficient resource usage in terms of CPU and memory, both on the client and server sides. Yojimbo, while performing similarly to ENet in some respects, excelled in maintaining game state consistency under unstable network conditions, and it might be an indication of its robustness in handling network disruptions. By taking this in consideration we may say that, Enet's non-compromising resource efficiency makes it a strong candidate for games with limited resources. However, its slightly lower consistency under various network conditions may indicate Yojimbo is a better candidate for action-intensive multiplayer games. Additionally, RakNet's higher frequency of corrections in-game states compared to ENet could be a point of concern for developers prioritizing game consistency.

Even though the results may be insightful for game developers, it has certain limitations. Firstly, the experiments were conducted under controlled network conditions, which may not fully replicate real-world scenarios. Secondly, the choice of game for benchmarking was relatively simple and may not reflect the complexities of a real multiplayer game. Thirdly, since we implemented the games ourselves without any prior experience with these libraries, we cannot guarantee that the behaviour of the libraries in terms of resource consumption and consistency will be similar in other implementations. As an example, we can see consistently high memory consumption for yojimbo1210. Similar implementation dependent behaviour may also be represented in other results. Due to time and scope constraints, we opted to leave further investigation of these results to future work.

For future work, the experiments could be expanded. One interesting work would be testing with more complex games by implementing the networking libraries in more complex and resource-intensive games. The research also uncovered areas that warrant further investigation. For instance, the variations in resource consumption between reliable and unreliable communication modes could be interesting to benchmark. Last but not least, conducting experiments over public networks would also be an effective experiment to better understand how these libraries perform under real-world conditions.

VIII. Conclusion

In this study, we conducted an extensive comparison of three notable game networking libraries: Raknet(SLikeNet), ENet, and Yojimbo, focusing on their scalability, resource consumption, and consistency in managing game states. Our findings demonstrate that each library possesses unique strengths and faces specific trade-offs, thereby providing valuable insights for game developers in choosing the most suitable networking library for their projects.

ENet distinguishes itself with its resource efficiency, exhibiting lower CPU and memory usage across both client and server sides. This characteristic renders ENet an attractive option for games operating under limited resource conditions. However, its slightly lower consistency in variable network environments suggests that Yojimbo could be a better fit for action-intensive multiplayer games where maintaining game state consistency is paramount.

Yojimbo's robust performance in maintaining game state consistency, particularly under unstable network conditions,

highlights its potential for use in games where reliable network communication is critical. Its ability to handle network disruptions efficiently is a key factor that developers might consider, especially for games with intensive multiplayer interactions.

RakNet, while showing a higher frequency of corrections in game states compared to ENet, may raise concerns for developers who prioritize consistency and seamless gameplay experience. Its performance indicates a need for further optimization in this aspect.

While our research provides comprehensive insights, it is not without limitations. The controlled network conditions of our experiments may not fully capture the complexities of real-world gaming scenarios. Additionally, the use of a relatively simple game for benchmarking might not reflect the challenges present in more complex multiplayer games.

Future research could expand on our work by implementing these networking libraries in more complex and resource-intensive games, and testing them in real-world network conditions. Further exploration into the variations in resource consumption between reliable and unreliable communication modes could also yield interesting results. Such research would contribute significantly to the field of online gaming, aiding developers in creating more efficient and engaging multiplayer experiences.

In conclusion, this study offers a thorough evaluation of Raknet(SLikeNet), ENet, and Yojimbo, providing critical insights into their respective capabilities and limitations. These findings will aid game developers in making more informed decisions, thereby enhancing the quality and performance of online multiplayer games. The study also lays the groundwork for future research, promising further advancements in the field of game networking technologies.

REFERENCES

[1] G Choi and Mijin Kim. "Battle Royale game: In search of a new game genre". In: *International Journal of Culture Technology (IJCT)* 2.2 (2018), p. 5.

[2] *ENet v.1.3.17*. Reliable UDP networking library [Online]. 2020. URL: http://enet.bespin.org/.

[3] *LibBench GitHub repository*. [Online]. 2024. URL: https : / / github . com / thijsheijden / DS - Game - Networking-Library-Benchmarking.

[4] Fatih Mar. *Game Networking Resources: A Curated List of Game Network Programming Resources*. [Online]. 2024. URL: https://github.com/ThusSpokeNomad/GameNetworkingResources.

[5] *Raknet GitHub Repository*. [Online]. 2014. URL: https://github.com/facebookarchive/RakNet.

[6] SLikeSoft. *Slikenet GitHub Repository*. [Online]. 2021. URL: https://github.com/SLikeSoft/SLikeNet.

[7] *Yojimbo GitHub Repository*. A network library for client/server games written in C++ [Online]. 2023. URL: https://github.com/networkprotocol/yojimbo.

IX. APPENDIX

The following pages contain our timelogs.

Name	analysis-time	dev-time	think-time	wasted-time	write-time	xp-time	Sum
Kaustav		23:00	11:00	4:00	10:00	3:00	51:00
Patrycja		21:15	12:10	0:30	4:25	1:45	40:05
Pelin		26:00	13:45		8:00	2:30	50:15
Pragya		17:00	17:00		0:15		34:15
Roy	10:00	19:30	18:25		25:00		72:55
Thijs		48:00	15:30	0:30	5:00		69:00
Sum	10:00	154:45	87:50	5:00	52:40	7:15	317:30

Date	Name	Type	Hours	Details
2023-11-06	Pragya	think-time	01:00	Experiments
2023-11-07	Patrycja	think-time	02:10	Read task and make notes
2023-11-07	Pragya	think-time	00:30	Meeting with team
2023-11-07	Pelin	think-time	00:30	Meeting with team
2023-11-08	Pragya	write-time	00:15	Experiments
2023-11-10	Roy	think-time	01:00	Meeting with group (including Jesse)
2023-11-10	Patrycja	think-time	01:00	Meeting with group (including Jesse)
2023-11-10	Pelin	think-time	01:00	Meeting with group (including Jesse)
2023-11-10	Kaustav	think-time	01:00	Meeting with Jesse
2023-11-10	Thijs	think-time	01:00	Meeting with group and Jesse
2023-11-12	Patrycja	wasted-time	00:30	Set up confluence and jira
2023-11-12	Patrycja	think-time	01:00	Experiments
2023-11-12	Pelin	think-time	01:30	Experiments
2023-11-12	Patrycja	write-time	01:00	Create initial report design
2023-11-12	Kaustav	think-time	00:30	Jira and Confluence experiment ideas
2023-11-12	Kaustav	write-time	00:30	Jira and Confluence experiment ideas
2023-11-13	Roy	think-time	02:30	Experiments
2023-11-13	Kaustav	think-time	00:30	Jira and Confluence experiment ideas
2023-11-13	Kaustav	write-time	00:30	Jira and Confluence experiment ideas
2023-11-15	Roy	think-time	03:00	Work on requirements
2023-11-15	Thijs	think-time	01:00	Thought of two experiments
2023-11-16	Roy	think-time	01:30	Meeting with group
2023-11-16	Pragya	think-time	01:30	Meeting with group
2023-11-16	Patrycja	think-time	01:30	Meeting with group
2023-11-16	Pelin	think-time	01:30	Meeting w/ group
2023-11-16	Thijs	think-time	02:00	Meeting with group
2023-11-16	Thijs	dev-time	02:00	Worked on POC terminal game
2023-11-17	Patrycja	think-time	01:00	Meeting with group (including Jesse)
2023-11-17	Pelin	think-time	01:00	Meeting with group (including Jesse)
2023-11-17	Kaustav	think-time	01:30	Meeting with group
2023-11-17	Thijs	dev-time	01:00	Continued working on POC
2023-11-17	Thijs	wasted-time	00:30	Combined all experiments into single file, meeting agenda Jesse
2023-11-18	Patrycja	think-time	01:30	Choose networking library
2023-11-18	Pelin	think-time	01:30	Choose networking library
2023-11-18	Thijs	dev-time	04:00	Worked on SlikeNet client and server + game model
2023-11-19	Pelin	dev-time	01:00	Built GGPO &check sample game
2023-11-20	Patrycja	write-time	00:45	Background
2023-11-20	Roy	think-time	01:30	Meeting with group
2023-11-22	Roy	write-time	03:30	System model
2023-11-22	Patrycja	think-time	02:00	Try to run yojimbo
2023-11-23	Roy	write-time	03:30	Requirements
2023-11-23	Roy	think-time	01:25	Prepare and plan meeting, set up docs
2023-11-23	Patrycja	dev-time	03:00	Work on yojimbo client server
2023-11-23	Thijs	dev-time	06:00	Continued working on SlikeNet implementation, struggled with static lib creation and linking
2023-11-24	Kaustav	think-time	03:00	Trying to set up Yojimbo
2023-11-24	Kaustav	xp-time	03:00	Trying to set up Yojimbo
2023-11-24	Pragya	think-time	03:00	Thinking about solutions for metrics experiments - suggested two solutions
2023-11-24	Roy	think-time	01:30	Meeting with group
2023-11-25	Patrycja	dev-time	02:30	Send int in yojimbo
2023-11-25	Pragya	think-time	02:00	Searching for libraries and metrics experiments
2023-11-26	Patrycja	dev-time	01:30	Client server connection yojimbo
2023-11-27	Patrycja	dev-time	00:30	Yojimbo add basic documentation
2023-11-27	Pelin	think-time	00:30	meet with group to discuss game progress
2023-11-27	Pelin	dev-time	01:00	Build Enet implement basic functionality for server and client
2023-11-27	Patrycja	think-time	00:30	meet with group to discuss game progress
2023-11-27	Pelin	think-time	00:30	meet with group to discuss game progress
2023-11-27	Roy	dev-time	03:30	Work on GameNetworkingSockets
2023-11-27	Thijs	dev-time	06:00	Continued working on game implementation using Slikenet
2023-11-28	Roy	dev-time	04:00	Work on GameNetworkingSockets
2023-11-28	Thijs	dev-time	04:00	Continued working on game implementation using Slikenet
2023-11-29	Kaustav	think-time	00:30	Stand-up call
2023-11-29	Pragya	think-time	00:30	Stand-up call
2023-11-29	Thijs	think-time	00:30	Group standup
2023-11-29	Thijs	dev-time	04:00	Finished RakNet game implementation
2023-11-30	Pelin	dev-time	04:00	Merge with the game add broadcasting behavior
2023-12-01	Patrycja	think-time	00:45	meeting with Jesse without Jesse
2023-12-01	Pelin	think-time	00:45	meeting with Jesse without Jesse
2023-12-01	Roy	think-time	01:30	Meeting with group
2023-12-01	Thijs	think-time	02:00	Found libraries and tools to use for hardware metrics tracking and network bandwidth measuring
2023-12-01	Thijs	think-time	01:00	Group meeting with Jesse
2023-12-02	Pragya	dev-time	02:00	Creating common interface for all the libraries to run experiments
2023-12-03	Patrycja	dev-time	04:30	add game logic in yojimbo - game state and config messages
2023-12-04	Patrycja	dev-time	03:30	add game logic in yojimbo - player spawn
2023-12-05	Patrycja	dev-time	04:00	add game logic in yojimbo - player moves
2023-12-05	Pelin	dev-time	03:00	Add unix configuration
2023-12-05	Thijs	think-time	01:00	Designed network usage benchmark experiment
2023-12-05	Thijs	dev-time	01:00	Wrote bash script to perform network usage benchmark experiment automatically
2023-12-07	Thijs	dev-time	02:00	Finished network usage benchmark script and started working on ingesting that data and creating plots
2023-12-08	Roy	think-time	01:30	Meeting with group
2023-12-09	Roy	dev-time	05:00	Work on GameNetworkingSockets
2023-12-10	Patrycja	write-time	00:40	add networking libraries choice to report with Pelin
2023-12-10	Pragya	think-time	05:00	Setting up raknet
2023-12-11	Patrycja	think-time	00:45	meet with group
2023-12-11	Patrycja	xp-time	01:45	run network measurements experiment yojimbo
2023-12-11	Kaustav	think-time	00:30	Group meeting
2023-12-11	Kaustav	dev-time	05:00	Setting up Enet
2023-12-11	Pelin	think-time	00:30	Group meeting
2023-12-11	Roy	write-time	07:00	Spend time on writing the report
2023-12-11	Roy	think-time	01:00	Meeting with group
2023-12-11	Thijs	think-time	01:00	Meeting with group
2023-12-11	Thijs	dev-time	04:00	Wrote readme for SLikeNet. Added network metrics script for Enet
2023-12-12	Pelin	dev-time	08:00	convert string usages to structs for messaging, test introducing latency using Clumsy and add benchmarking logic to server side for game consistency latency part - enet
2023-12-12	Patrycja	dev-time	00:45	add makefile to yojimbo
2023-12-12	Kaustav	dev-time	06:00	etting up pf.conf/dummynet, testing for packet loss
2023-12-12	Pragya	dev-time	06:00	Setting up dummynet config files and testing for packet loss
2023-12-13	Kaustav	think-time	00:30	Meeting with group
2023-12-13	Pragya	think-time	00:30	Meeting with group

2023-12-13	Roy	write-time	03:00	Spend time on writing the report
2023-12-13	Roy	dev-time	04:00	Work on yojimbo
2023-12-13	Thijs	think-time	01:00	Group meeting
2023-12-13	Thijs	dev-time	02:00	Got Yojimbo working!
2023-12-14	Kaustav	dev-time	04:00	Implementing packet loss with 1 client, Pelin helped
2023-12-14	Pelin	dev-time	03:00	Helped Kaustav implementing packet loss with 1 client
2023-12-14	Pragya	dev-time	04:00	Implemented packet
2023-12-14	Roy	dev-time	03:00	Work on yojimbo
2023-12-14	Thijs	write-time	01:00	Wrote parts in the lab report about the game client and server
2023-12-14	Thijs	dev-time	02:00	Improved Yojimbo Makefiles, add common config to specify map size and toggle for reliable networking
2023-12-15	Patrycja	dev-time	01:00	add command line parameters
2023-12-15	Kaustav	think-time	01:30	Meeting with Jesse, team planning
2023-12-15	Pragya	think-time	01:30	Meeting with Jesse, team planning
2023-12-15	Roy	think-time	02:00	Meeting with group, + preparation
2023-12-15	Pelin	think-time	01:30	Meeting with group, + preparation
2023-12-16	Kaustav	dev-time	05:00	Building a script for gathering CPU/memory usage
2023-12-17	Kaustav	dev-time	03:00	Finalizing and pushing bash script for metrics
2023-12-22	Roy	write-time	08:00	Process results, plot graphs, add explanation of results
2024-01-04	Pelin	dev-time	03:00	parse command line parameters
2024-01-05	Pelin	dev-time	02:00	work on third experiment
2024-01-05	Thijs	dev-time	04:00	Wrote scalability bash script. Ran scalability experiment for all libraries. Wrote processing Python script to process and calculate averages for all clients.
2024-01-09	Thijs	dev-time	02:00	Updated Enet code to work with 64 clients. Ran scalability benchmarks for Enet again.
2024-01-09	Thijs	think-time	01:00	Thought about the third experiment, helped Kaustav trouble shoot
2024-01-12	Kaustav	wasted-time	04:00	Trying to figure out latency/packet loss using network link conditioner
2024-01-12	Kaustav	think-time	01:30	Call with Pelin for Enet experiment 3
2024-01-12	Pragya	dev-time	04:00	Figuring out link conditioner and running latency/packet loss experiment for raknet
2024-01-13	Pelin	think-time	01:30	Call for Enet experiment 3
2024-01-13	Pelin	think-time	01:30	Design possible algorithms for third experiment
2024-01-15	Pragya	think-time	01:30	Continuing work on packet loss experiment
2024-01-15	Pragya	dev-time	01:00	Providing output sheet
2024-01-18	Kaustav	write-time	04:00	Working on report
2024-01-18	Roy	analysis-time	04:00	Plot more graphs
2024-01-23	Pelin	xp-time	02:30	work on third experiment for enet and test for experiment
2024-01-23	Pelin	dev-time	01:00	bugfix on third experiment
2024-01-23	Pelin	write-time	03:00	write the implementation part for the report
2024-01-24	Kaustav	write-time	02:30	Working on report
2024-01-24	Kaustav	write-time	00:30	Working on report
2024-01-25	Pelin	write-time	05:00	Working on the report
2024-01-25	Thijs	dev-time	04:00	Implemented correction tracker class and added to all library implementations
2024-01-25	Thijs	think-time	04:00	Ran network resilience experiment for all library implementations
2024-01-26	Thijs	write-time	04:00	Final writing for report
2024-01-26	Kaustav	write-time	02:00	Adding final touches to the report
2024-01-25	Roy	analysis-time	02:00	Process data from third experiment
2024-01-26	Roy	analysis-time	04:00	Process data and plot graphs from third experiment
2024-01-26	Patrycja	write-time	01:30	Prepare time report
2024-01-26	Patrycja	write-time	00:30	Prepare time report