UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Performance analysis of synchronous and asynchronous parallel network server implementations using the C++ language

Author:
Yuriy PASICHNYK

Supervisor:
Oleg FARENYUK

A thesis submitted in fulfillment of the requirements for the degree of Bachelor of Science

in the

Department of Computer Sciences
Faculty of Applied Sciences

Lviv 2022

Declaration of Authorship

I, Yuriy PASICHNYK, declare that this thesis titled, "Performance analysis of synchronous and asynchronous parallel network server implementations using the C++ language" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

“You learn more from ten days of agony than from ten years of content.”

Sally Jessy Raphael

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Performance analysis of synchronous and asynchronous parallel network server implementations using the C++ language

by Yuriy PASICHNYK

Abstract

The main two paradigms for implementing parallel network servers are synchronous and asynchronous. The first two questions in our heads are: "which one is better?" and "which one should I use?". In this thesis, we show answers to these questions on practice. After overview of existing methodologies and implementation choices, we design and implement the most representative and valuable versions of a stateful TCP echo server. Then we test the server based on five major metrics: throughput, latency, simultaneous client connections number, CPU usage, and memory consumption. We conclude that a hybrid synchronous solution is a superior choice for a server with real world workload.

Acknowledgements

I am infinitely grateful to my dear family and friends who support me in these challenging and dark times. The support and inspiration from my closest people helped me finish this thesis and kept my mental health with me in the middle of the war.

I am also very thankful to my supervisor Oleg Farenyuk for his motivation and precious advice. I truly value the knowledge and experience which I learned from him. He gave me a solid basement for my future academic and industrial growth.

Finally, I want to say that I appreciate the unforgettable time spent as a student at UCU. I want to thank all my groupmates, lectors, faculty, and university.

Contents

List of Figures

List of Tables

List of Abbreviations

CPU	**C**entral **P**rocessing **U**nit
LAN	**L**ocal **A**rea **N**etwork
OSI	**O**pen **S**ystems **I**nterconnection (model)
IP	**I**nternet **P**rotocol
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
PC	**P**ersonal **C**omputer
HW	**H**ard**w**are
OS	**O**perating**S**ystem
IO	**I**nput **O**utput
LTS	**L**ong **T**erm **S**upport
NIC	**N**etwork **I**nterface **C**ontroller
ACL	(Network) **A**ccess **C**ontrol **L**ist
GB	**G**iga**b**yte
MB	**M**ega**b**yte
KiB	**Ki**lo**B**yte
MiB	**M**ega**B**yte
sec	(one) **sec**ond
RAM	**R**andom **A**ccess **M**emory
MU	**M**emory **U**sage
DDR4	**D**ouble **D**ata **R**ate **Fourth** (Generation)
SDRAM	**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory

Dedicated to Freedom . . .

Chapter 1

Introduction

1.1 Motivation

In the age of computers, the demand for highly efficient and scalable servers, which can serve thousands, even millions of clients simultaneously and handle the request from each of them in fractions of a second is higher than ever. Even though the requirements are constantly increasing, CPU single-core did not gain much performance concerning customer needs (Stanford VLSI Group, 2022).

The market came up with multi-core CPUs to fit the order for improvements. Parallelism gains efficiency by executing the work simultaneously in multiple threads. What is the best way to use the capabilities of a multi-thread CPU in terms of performance maximization? Over the years, we get to the point with two main paradigms: synchronous and asynchronous methodologies of executing the tasks.

A network server is a typical I/O related and mainly – I/O-bound task. In this thesis, we compare several synchronous and asynchronous implementations of the representative parallel network server – the echo server, developed using related established methodologies and analyze the performance of those implementations. We use the C++ programming language for implementing the servers.

1.2 C++ Language Overview

An essential choice for the artist is the tool that should have the ability to supplement and visualize his unreachable thoughts. The programming language plays a vital role in this work. The requirements for it are pretty simple but, at the same time, superior. In the first place, the language should allow writing very efficient and complex solutions. Secondly, it should have a reach standard library with well-designed common components. Furthermore, it should be convenient and flexible to fit all user's needs. Last but not least, the industry should trust and use the language in their needs.

By these requirements, we come up with the two best candidates: C and C++. Both of them are great. However, C++ fits the requirements better than C. In the folowin ways:

- It is more convenient than C.
- It provides powerful abstraction mechanisms.
- It has a more prosperous standard library that better suits our research.

As Bjarne Stroustrup, the father of the language writes in his book (Stroustrup, 2013):

> Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types. A programmer can partition an application into manageable pieces by defining new types that closely match the concepts of the application.

In addition to all the positive aspects above, we use C++ because of the low level of control. We need two perform measurements of the methodologies and not the implementation-dependent abstraction or the run-time routines of the programming languages, such as the Java programming language.

1.3 Network Server

According to the basic definition, the network server is a program that offers some functional/service to other programs or computers, called clients (Wikipedia, 2022b). The interaction between server and client looks as follows: the client sends a request to the server, the server processes this task, and sends a response. The processing of the request could be any manipulation of the data and can differ from implementation to implementation. To get meaningful and controllable results in this thesis, we will eliminate the request processing and send the request message back to the client. We will profile the server's core function: receiving requests and sending the response. These types of servers are called Echo Servers as are described in RFC862 (*Echo Protocol* 1983).

Another crucial aspect is the session duration. This aspect is critical because each client session consumes the server's memory resources and requires constant monitoring for requests. This option divides servers into two classes: stateless and stateful. Stateless servers establish a connection with the client only for one request. After the server receives the data, processes it, and sends a response, it immediately closes the connection, and no state of the client is saved.

On the contrary, the stateful server does not close the session between the client's requests. This server type can remember what actions the client performed in his session. The session is closed when the user terminates it on his side. Then server closes the connection end on its side. This paper focuses on stateful echo servers, potentially more valuable and applicable to common usage scenarios.

1.3.1 Transmission Control Protocol vs. User Datagram Protocol

There are many types of protocols on different network logical layers in the network. The standard definition of the network layers is described using the Open Systems Interconnection (OSI Wikipedia, 2022a) model. In simple words, protocols consist of standardized conventions which define how to transfer data from one host in the network to another. Depending on the user's requirement, a protocol type is chosen. The major used protocols of the transportation layer of the OSI model are Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

The main difference between TCP and UDP in this paper context is that TCP opens a session and can hold it. On the contrary, UDP does not support sessions at all. It resolves the client request and then sends the response to the destination as a separate connection (Andrew S. Tanenbaum, 2014).

We interpreted TCP as stateful and UDP as stateless in the previous section. So as we decided to work with a stateful Echo Server, the best option is to choose TCP as the layer four protocol. In such a way, we aligned our requirements to the server with the standard requirements for the TCP Echo Service description in RFC862 (*Echo Protocol* 1983).

1.3.2 Server Requirements

To conduct a performance analysis of synchronous and asynchronous versions of parallel network servers, we should choose meaningful requirements to get results representing the actual common server core functionality. The requirements for the server determine the implementation prerequisites to stay in the scope of the research and get valuable results:

- Server should send back received data from the client as the calculated response.
- Server should hold the client session until the client terminates it.
- Server should use TCP as the transport level protocol.
- Choose C++ as the best suitable programming language for implementation.

Chapter 2

Literature Review

2.1 Synchronous Model

There are few research papers regarding synchronous models compared to asynchronous one. First of all, we will overview the basics. The best point to start is the definition of the synchronous model.

The synchronous methodology is also referred to as thread-based. The main criteria of this methodology is when the associated thread stack maintains the execution of the program (Ilie et al., 2009). In other words, it means that the decision-making of what should be executed is delegated to the OS. The main characteristics of synchronous models are:

- Usually easier to use.
- Less efficient because of context switches.
- Have higher memory consumption.

Those conclusions about synchronous models are common among a number of articles (Karabyn, 2019; Zhang et al., 2020).

2.2 Asynchronous Model

There is an enormous number of works related the asynchronous model topic written. In any case, we should start with the definition of the asynchronous methodology.

The asynchronous methodology also referred to as event-based, is the one which behavior is defined by several (non-nested) event handlers called from inside an event loop. The execution state of a concurrent process is maintained by an associated record or object (Ilie et al., 2009). Meaning that a kind of worker processes dispatches and prioritizes the asynchronous routines.

The main characteristics of asynchronous models are:

- Complicated to use in large designs.
- Usually more efficient.
- Have less memory consumption than synchronous models.

Those conclusions are repeated in a number of related works (Ilie et al., 2009; Andrew S. Tanenbaum, 2014; Krohn, Kohler, and Kaashoek, 2007).

The following work provides a solid overview of asynchronous models: "The Impact of Event Processing Flow on Asynchronous Server Efficiency“ (Zhang et al., 2020). The main message of this thesis is to show that real-world asynchronous solutions are not implemented really efficiently. The common problems faced by developers are the following:

- The widely adopted *one-event-one-handler* event processing is not good because it causes frequent unnecessary context switches between event handlers.
- Repeatedly making unnecessary I/O system calls.
- Underestimate hybrid solutions, which can give from 10 to 90 percent higher throughput compared to pure asynchronous.

2.3 Kernel

The other option for consideration of accelerating the implementation would be to execute it as a module in kernel space. In such a way, we would avoid time-expensive context switches for system calls. We would have the ability to develop a solution that would use kernel resources directly to avoid unnecessary overhead.

In the article "TAS: TCP Acceleration as an OS Service" (Kaufmann et al., 2019), the authors show that in their case, executing the logic in kernel mode increases the throughput by up to 90% and lowers the tail latency by 57%. Also, this approach provides 2.2 times higher throughput for 64000 simultaneous connections. Similar results we can observe in another article: "Kernel Paxos" (Esposito, Coelho, and Pedone, 2018).

Chapter 3

Design Strategies and Tools

3.1 Work Outline

This thesis aims to create synchronous and asynchronous implementations of a parallel network server and analyze their performance. To carry out the analysis, we should gather valid and meaningful measurements by the presentable metrics in server performance. Another essential point is that we have to establish a setup that can hold the maximum possible workload that we can reach with our tool-set. We should not limit the implementations by the network throughput to get valid results.

The work plan is the following:

- Implement the server using all the practical approaches in the scope of this thesis.
- Perform local validation of the implemented solutions.
- Design and assemble a Cluster and Network setups that will be able to generate and pass enough traffic to simulate a real-world high load scenario and measure the performance metrics.
- Assemble and configure the testing setup.
- Measure the Cluster and Network limits.
- Test the implementations and measure their performance.
- Verify that we do not encounter the limits of the cluster and network but push the implementation to the maximum CPU load.
- Analyze the collected metrics and summarize the results.

3.2 Performance Analysis

3.2.1 Performance Metrics

The first step in performance testing is to choose metrics that will be valuable for our case study. There is no need to reinvent the wheel. There are common standardized benchmarking tests in the network performance area that are described in the RFC2544 (*Benchmarking Methodology for Network Interconnect Devices* 1999). So the main performance concern points for our case are:

- Throughput,
- Latency.

The general approach metrics are not enough for a stateful client-server. In this case, an important place is the client connections number, as each of them uses some resources such as CPU, memory, and process file descriptors. By extending our metrics model, we get a satisfying set of metrics to design the test setup:

- Throughput,
- Latency,
- CPU consumption,
- Memory consumption,
- File descriptors utilization.

3.2.2 Cluster Setup

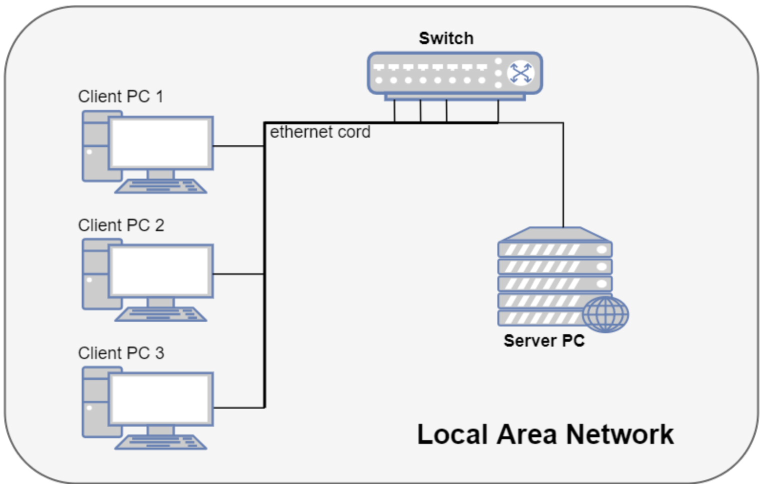

FIGURE 3.1: Cluster setup.

Nodes and Hardware

Cluster hardware specification is presented in the Table 3.1. The Switch node is described in the following Network setup subsection related to the Network setup.

Node name	CPU model	Phys./Logic cores	RAM, GB	NIC, MB/sec	OS
Client PC 1	AMD 2990WX	32 / 64	64	125	Ubuntu 20.04 LTS
Client PC 2	i7-1165G7	4 / 8	16	125	Ubuntu 20.04 LTS
Client PC 3	i5-7300HQ	4 / 4	8	125	Ubuntu 20.04 LTS
Server	i5-1135G7	4 / 8	32	125	Ubuntu 20.04 LTS

TABLE 3.1: Cluster nodes hardware description

The Server takes the primary role in the setup. For a better understanding of the processes and performance data results, it is crucial to keep in mind the detailed characteristics of the server node, which are listed in Table 3.2.

Characteristic	**Value**
CPU Architecture:	x86_64
CPU Model name:	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
Logical CPUs:	8
Physical CPUs:	4
CPU max MHz:	4200
CPU min MHz:	400
CPU Byte Oserver	Little Endian
L1d cache:	192 KiB (4 instances)
L1i cache:	128 KiB (4 instances)
L2 cache:	5 MiB (4 instances)
L3 cache:	8 MiB (1 instance)
RAM	12.0 GB
RAM type	DDR4 SDRAM
OS	Ubuntu 20.04 LTS
OS Kernel	Linux Kernel 5.4
NIC	Gigabit Ethernet LAN

TABLE 3.2: Cluster nodes hardware description

Network setup

Our primary concern about network setup is that it should not limit the Server workload. The bottleneck of the traffic flow should be the server logic, not the cluster network HW or configurations. The network should be designed so that the only limit for traffic throughput should be the limits of the Server NIC, namely 1000GB.

For the interconnection of the cluster nodes, the best available variant was a Cisco Switch with 4 Ethernet Gigabit ports (Cisco Systems, Inc., 2022) and Ethernet cords to connect the machines to the switch. As to characteristics of the switch, it can sustain 1-gigabit egress traffic on one of the Gigabit Ethernet ports when the traffic is consumed from 3 other Gigabit Ethernet ports, which should sum up to 1 gigabit. For our test purpose, there is no need for more than one-gigabit server data flow as the Server NIC can not handle more, and this limitation was not reached during the server implementations tests.

After we connect the nodes to the switch, we should set for each of the nodes a static IP address to finish the network setup and for the PCs to have the ability to communicate over the TCP/IP stack. In Linux, to set a static IP address, we used the IP command-line tool (Kerrisk, 2022). The IP address and IP masks which were used we can find in the Table 3.3 below.

Node name	**IP address**	**IP mask**
Client PC 1	192.168.0.1	255.255.255.0
Client PC 2	192.168.0.2	255.255.255.0
Client PC 3	192.168.0.3	255.255.255.0
Server	192.168.0.4	255.255.255.0

TABLE 3.3: Cluster nodes IP addresses setup.

3.2.3 Network Limitations

In the scope of measuring network limits, we used an open-source command-line tool – iperf3 (Mah, Poskanzer, Tierney, et al., 2019). We start an iperf3 server instance on the Server node and an iperf3 client instance on any host node using this program. Then the client starts flooding the Server and measuring the maximum throughput. To get the relevant result, we set the traffic payload size to the expected message size during the implementations performance testing in Section 4.2 that is equal to 64B.

After measuring the throughput by the specifications in the previous paragraph, we found out that the throughput is 72 MB/sec. That is slightly less than the edge capabilities of the NIC, and we will not get close to those limits, so the results are excellent: we have a working test setup that satisfies all requirements for proper testing.

3.3 Implementation Choices

After investigating the standard solutions which we overviewed in Chapter 2 we can get to the point of choosing our solution building blocks and mechanisms from the set of available and commonly used in the industry. In this section, we will go through the different implementation-specific choices options, which we will filter out and assemble a high efficient and working solutions in Section 4.1.

3.3.1 Syscalls

Let us overview syscalls that will be useful for implementation purposes. As a source, we will use Linux manual pages (Kerrisk, 2022). We have three main parts for a TCP server where syscalls could be needed Socket management, Network IO, and Clients management. The first part is the TCP socket management syscalls used for creation, closing, and management through the whole socket lifecycle: from creation to closure. Here is the list of them (Kerrisk, 2022):

- socket – create an endpoint for communication,
- setsockopt – set options on sockets,
- getsockopt – get options on sockets,
- bind – bind a name to a socket,
- listen – listen for connections on a socket,
- accept – accept a connection on a socket,
- shutdown – shut down part of a full-duplex connection,
- close – close a file descriptor.

At this point, all syscalls have their place in the TCP server work algorithm and do not have a common replacement. So then continue to the Network IO syscalls (Kerrisk, 2022):

- read – read from a file descriptor,
- write – write to a file descriptor,

- recv – receive a message from a socket,
- send – send a message on a socket,
- fcntl – manipulate file descriptor.

Also, we will need so-called Clients management syscalls which will provide information about some new connections pending on the server socket or requests pending on the client file descriptors.

- select – wait for some event on a file descriptor – old and outdated syscall,
- poll – wait for some event on a file descriptor – a modern way of performing the task of the select syscall.

In our asynchronous and hybrid synchronous solutions we will use asynchronous syscalls such as poll, non-blocking read, and write.

Now, when we have the general view of all related syscalls, let us outline the synchronous TCP server algorithm from the syscall point of view. We illustrate a stateless server for simplicity as stateful is much more complex. In Figure 3.2, we can see the steps which are performed by a standard stateless TCP server.

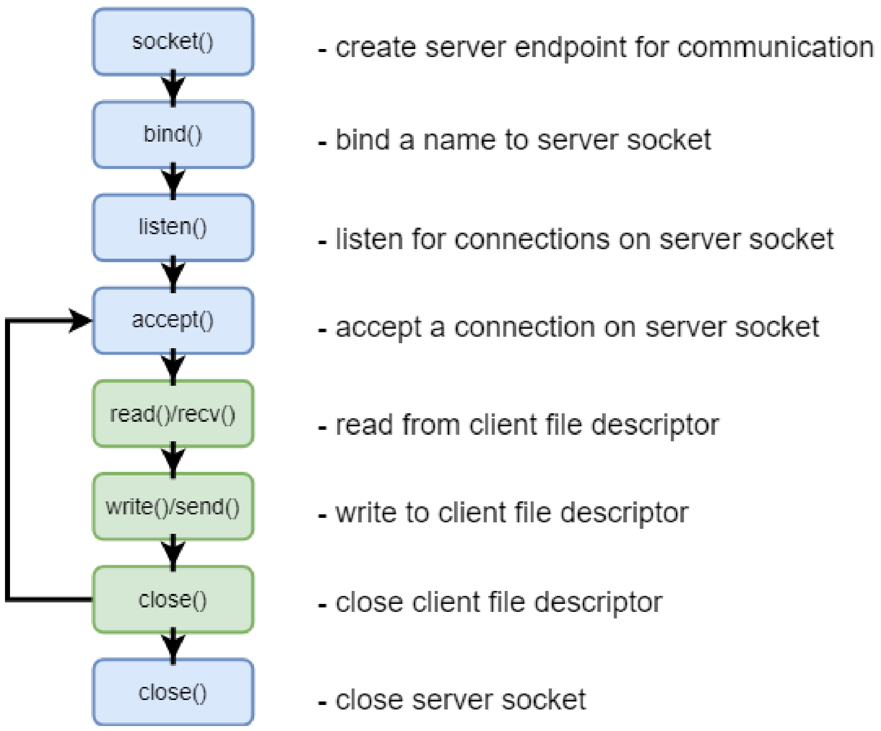
socket()
- create server endpoint for communication
bind()
- bind a name to server socket
listen()
- listen for connections on server socket
accept()
- accept a connection on server socket
read()/recv()
- read from client file descriptor
write()/send()
- write to client file descriptor
close()
- close client file descriptor
close()
- close server socket

FIGURE 3.2: Synchronous stateless TCP server algorithm from syscalls perspective.

3.3.2 Libraries

In the scope of the thesis, we will use the C++ boost library (Kohlhoff et al., 2019) as an industrial asynchronous toolkit. We need one feature set from this library – asynchronous IO component, which we are referencing as boost::asio. We will use this feature to implement asynchronous solutions. The major infrastructure that we will need can be categorized in such a way:

- socket management structures,
- asynchronous IO structures,
- asynchronous management structures.

3.4 Synchronous strategies

3.4.1 Multithreaded Synchronous

For our server requirements, we implemented the most straightforward pure synchronous implementation possible. The fundamental concept of this strategy is that in the main execution loop in the main thread, we accept new connections, and for each new connection, we create a separate execution thread to handle the client requests. The thread work routine is to receive requests and responds to them by sending them back to the client.

3.4.2 Single Threaded Hybrid Synchronous

The standard design of a single-threaded synchronous version does not satisfy the server requirements. If we accept a new connection and do a blocking read, no one else will be able to connect to our Server until the read is finished. To overcome this limitation, we will use file descriptor manager syscalls such as poll or select, which will give us information when we will have new incoming connections or new clients' requests. This solution is not pure synchronous implementation as the call to the manager syscall is considered asynchronous, but such a hybrid strategy is valuable to implement and investigate.

3.4.3 Multithreaded Hybrid Synchronous

One more strategy that we will discover is slightly different from single-threaded hybrid synchronous (Section 3.4.2). The difference will be that we will make it multithreaded. In the main thread, we will connect new clients and schedule request handle jobs in response to file descriptor management syscall events. In other threads, we will be handling the client requests. For work organization and thread management, we will use a custom-written thread pool.

3.5 Asynchronous Strategies

The asynchronous strategies are designed to be implemented using the boost::asio library.

3.5.1 Single-threaded Asynchronous

We configure our Server as usual but use the boost::asio interface in this strategy. Then we schedule the first asynchronous connection accept. Then the handler of this asynchronous operation will schedule two asynchronous jobs: a new accept and read for the newly connected client. The next step is that the asynchronous client read will schedule an asynchronous write in its handler. Finally, the asynchronous write handler will form a loop by scheduling an asynchronous read.

Furthermore, we start the worker in the main execution thread after scheduling the asynchronous accept. The worker will process the scheduled jobs when they are ready.

3.5.2 Multithreaded Asynchronous

Similar to the single-threaded Asynchronous strategy (section 3.5.1), the Multithreaded Asynchronous differs only in the aspect of how many workers are started in separate threads. In the main worker thread after scheduling an asynchronous accept call, we start the required number of workers in separate threads.

Chapter 4

Experiments

4.1 Implementation

All implementations are available on GitHub repository (Pasichnyk, 2022). C++20 standard was used. The compilation for the test was done using CMake files with release optimization.

4.1.1 Version echo_server_simple_threaded – sync

This implementation was the most trivial one. We used the read/write syscalls for IO, the standard C++ threads library for creating threads, and the typical server syscalls configuration sequence illustrated in Figure 3.2. The algorithm was the same as described in Section 3.4.1.

4.1.2 Version echo_server_simple – hybrid-sync

We also used the read/write syscalls for IO for this implementation. For client management, we used the poll syscall and the typical server syscalls configuration sequence, which is illustrated in Figure 3.2. The algorithm was described in Section 3.4.2.

4.1.3 Version echo_server_custom_thread_pool – hybrid-sync

This implementation was the most complex and time-consuming one. The syscall choice is the same as for the echo-server-simple version(Section 4.1.2). The threads are used from the standard C++ thread library. A custom thread pool was developed and integrated for thread management and work schedule. The thread count in is tuned to the server logical core count to maximize the implementation performance. The algorithm was described in Section 3.4.3.

4.1.4 Version echo_server_boost_asio – async

This implementation is entirely based on the C++ boost::asio library. All IO functions used are non-blocking. The algorithm was described in Section 3.5.1.

4.1.5 Version echo_server_boost_asio_threaded – async

This implementation is almost the same as the echo-server-boost-asio(Section 4.1.4) except for the working threads count. The workers are started in separate threads using the standard C++ thread library.This difference is also described in the algorithm from Section 3.5.2. The thread count is adjusted to the logical cores number of the server host used in the tests.

4.2 Performance Testing

The performance testing is done using the Fortio command-line tool (Demailly et al., 2022). The Fortio command was issued at a specific time on each test cluster instance. Before scheduling the test job, the echo node was synchronizing its time with a local time server running on one of the Client PC 1. In the next step, a job for a test run is scheduled. A Fortio instance is started with the same parameters at the scheduled time on each client.

- -qps 0 – try to send maximum number of queries per second,
- -t 60 s – test duration 60 seconds,
- -c <client-num> – client number parameter (values used: 8, 25, 50, 100, 250, 500, 1000, 2000, 5000, 10000),
- -payload-size 64 – set the client message size,
- -uniform – de-synchronize parallel clients' requests uniformly,
- -json <file-path> – the JSON output file path.

The result was then collected and merged. A running server implementation was restarted for each test run on the server. Also, the server was running a background task that constantly logged CPU usage, memory usage, and network read/write bytes and appended to each read a timestamp by which the results were filtered.

4.3 Results

Results are visualized for more straightforward perception and presented once in the original scale and a logarithmic one. Every implementation is visualized on the plots as lines or bars of a unique specific color.

4.3.1 Throughput in respect to Clients number

Figure 4.1 shows that single-threaded hybrid synchronous (Section 4.1.2) and asynchronous multithreaded (Section 4.1.5) implementations have the best results in terms of throughput. For the average connection count values, the asynchronous one is dealing better in the range of (250-2000). But starting from 2000 and higher, the synchronous one seems to manage a higher number of connections better. The asynchronous single-threaded version (Section 4.1.4) shows the worst results when the number of connections goes over 2000. The synchronous multithreaded implementation (Section 4.1.1) shows average results, but as numbers of connections goes higher – greater than 2000, this version has a descending dynamic in terms of throughput. Though, taking into account simplicity of this implementation, it performs really well. The hybrid synchronous multithreaded implementation (Section 4.1.3) shows the lowest results from the lowest connection number to 2000. Starting from 5000 connections, it has average values among all our versions. If we compare how much time it takes to implement, it is not worth that effort.

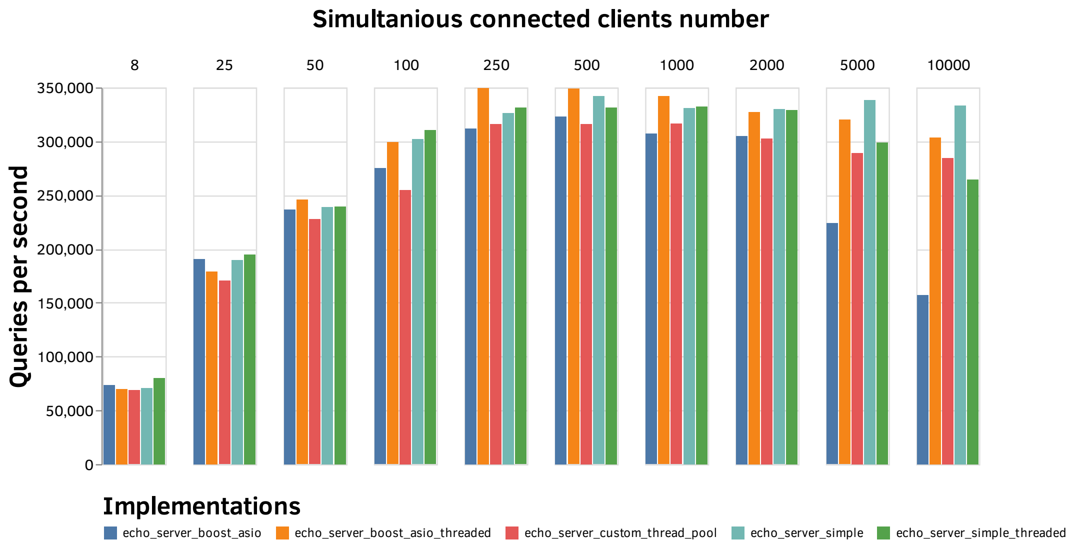

FIGURE 4.1: Throughput in respect to clients number by implementation version.

4.3.2 Latency in respect to Clients Number

Average Latency

On the average latency plot in Figure 4.2 we observe that all except single-threaded asynchronous implementation (Section 4.1.4) have similar results and constant stable growth in latency. The latency of the single-threaded asynchronous implementation beginning from 2000 connections grows much faster than other implementations. The synchronous multithreaded implementation (Section 4.1.1) starting from 5000 connections has worse results than the hybrid synchronous multithreaded version (Section 4.1.3), so the difference between those two implementations was relatively small at less than 5000 connections. We can summarize that the hybrid synchronous multithreaded version has better results than the synchronous multithreaded one in terms of average latency. However, in the perspective of a real-world server at such a high server load, the difference between all implementations, except the single-threaded asynchronous one, is not noticeable.

Percentile 90

Based on the average latency results, we expect that as the client number increases, the latency will grow appropriately. It is easily visible on a plot with the 90 percentile latency on Figure 4.3.

We can observe that latency percentile for all implementations is similar among all versions when there are less than 100 simultaneous clients connected to the server. Then from 100 to 2000 connections, the synchronous version(Section 4.1.1) demonstrates better results, probably because of better and faster dispatcher mechanisms and fewer unnecessary syscalls. However, when we reach 5000 connections, we observe that the synchronous version latency values severely increase, so this implementation takes the next to last place based on the 90s latency percentile. The most stable and predictable result shows the hybrid synchronous version (Section 4.1.2). As the connection number will grow to over 10000 probably this implementation will be not able to manage such a number of connections in a single thread. On middle range connection numbers such as 5000, the asynchronous, multithreaded

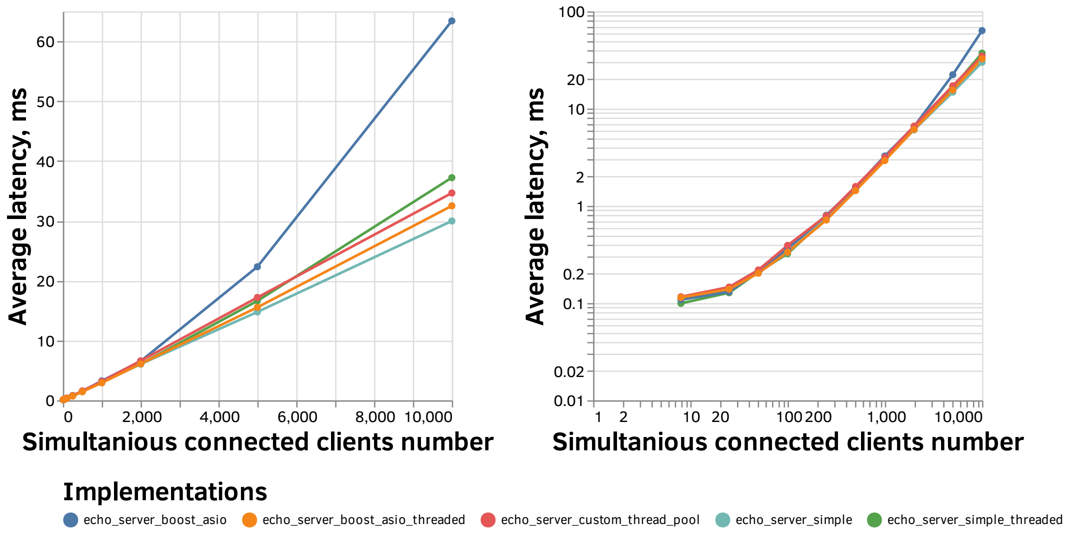

FIGURE 4.2: Average latency in respect to clients number.

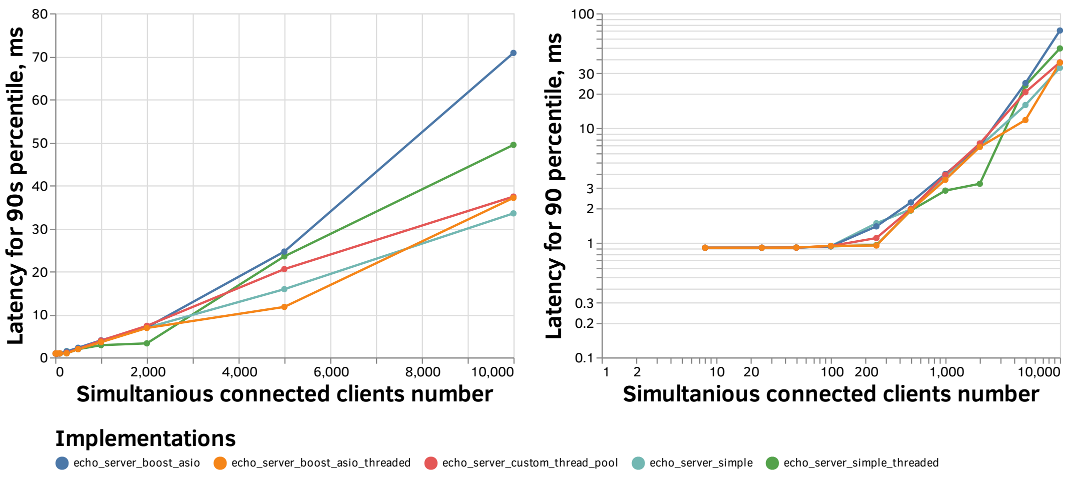

FIGURE 4.3: 90 percentile latency in respect to clients number.

version (Section 4.1.5) shows better results. It could be considered the best version for this number of connections. However, we cannot make such a conclusion with such poor resolution of the plot. Based on current test latency data, this implementation can take the place of the second-best solution. Right behind asynchronous multithreaded would be a multithreaded hybrid synchronous implementation(Section 4.1.3). The worst values are from the asynchronous single-threaded implementation(Section 4.1.4).

Percentile 99

For plot on Figure 4.4 with 99 percentile we see that all threaded implementations (from Sections 4.1.5, 4.1.1, and 4.1.3) have 1 percent of responses with latency over 200 milliseconds when they have 5000 or more clients. Other two implementations (from Sections 4.1.2 and 4.1.4) have stable results at 99 percent of responses. The most stable latency results on our server load are still shown by the hybrid synchronous implementation (Section 4.1.2).

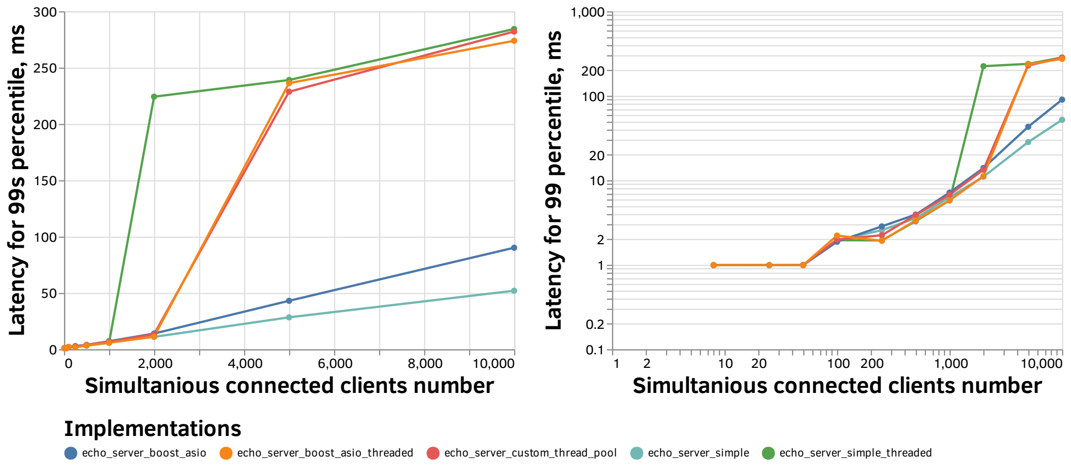

FIGURE 4.4: 99 percentile latency in respect to clients number.

Percentile 99.9

At this point, taking into account the 99.9 percentile plot (Figure 4.5), we can get to the conclusion that the hybrid synchronous single-threaded version (Section 4.1.2) is the most stable in the range up to 10000 connections. Also, from this plot, we see that asynchronous single-threaded implementation (Section 4.1.4) tends to have outliers responses with latency over 800 milliseconds with a probability of about 0.01 percent. Other implementations have, on average, latency outliers with similar values.

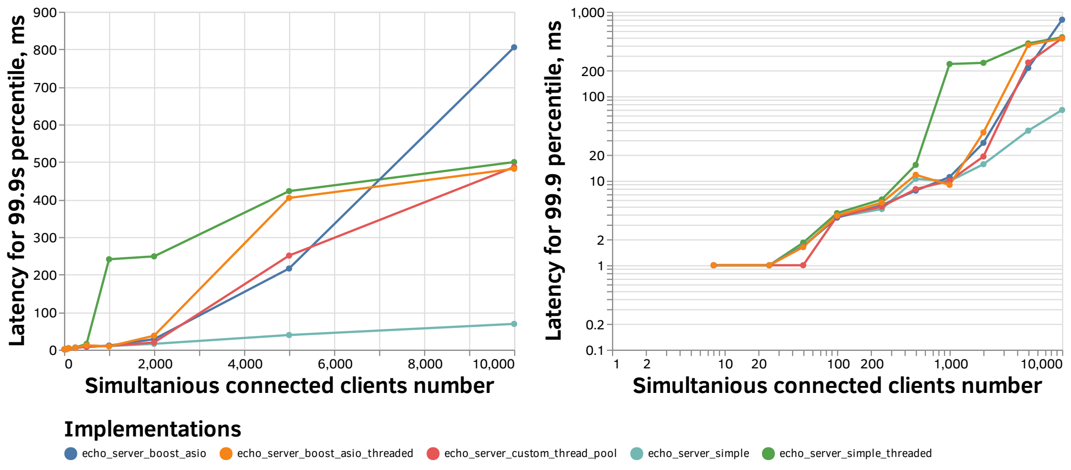

FIGURE 4.5: 99.9 percentile latency in respect to clients number.

4.3.3 CPU usage in respect to Clients Number

The Figure 4.6, demonstrates the utilization of CPU by each of the implementations in a real-world load scenario with different numbers of connected clients. The hybrid synchronous single thread (Section 4.1.1) and the single thread asynchronous (Section 4.1.4) have the lowest CPU usage. However, they both fully utilize their single execution thread. Furthermore, in the middle of the rating is the synchronous multithreaded version which consumption rises from 10 to 60 percent as the connection number grows from 8 to 1000. For more than 1000 client connections, the CPU usage varies from 55 to 75 percent. The asynchronous multithreaded (Section 4.1.5) and hybrid synchronous threaded (Section 4.1.3) implementations consume much more CPU time than others. However, they do not use the full potential of the CPU. This behavior can be explained due to not having enough load on each client, or there are not enough connections to utilize the CPU fully. They both perform almost the same way until the number of the connections reaches 200. After that point, the asynchronous, multithreaded version consumes 10 percent more CPU than the hybrid synchronous threaded.

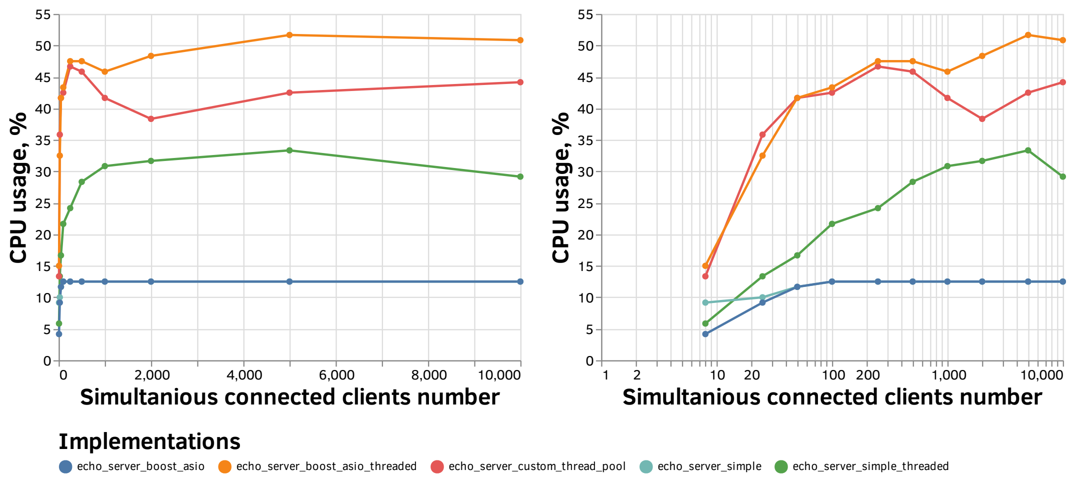

FIGURE 4.6: CPU usage in respect to clients number by implementation version

4.3.4 Memory usage in respect to Clients Number

From the memory usage plot (Figure 4.7), we can indeed say that the synchronous implementations (Section 4.1.1) consume more memory. Memory usage grows linearly with the connected clients because separate threads are created to handle each connection. As we can observe, each thread consumes significantly more memory for each connection than all other implementations.

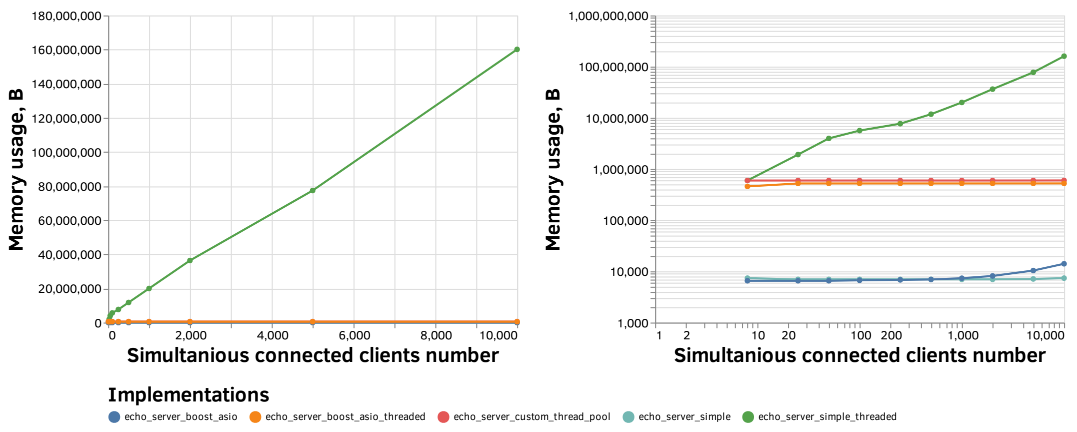

FIGURE 4.7: Memory usage in respect to clients number by implementation version.

Chapter 5

Conclusions

5.1 Conclusions

This thesis was aimed to implement and analyze the performance of the several typical synchronous and asynchronous implementations of parallel network servers using C++. We implemented five versions of the parallel server. The maximum supported simultaneous connection number is up to 10000. The last stage is to make meaningful conclusions based on test results that will answer the questions: Which methodology should we use? Synchronous or Asynchronous?

There is no master key solutions or methodology. Concerning our needs, we choose the methodology that will suit our particular case the best. The main parameter is how many connections we expect to have. Furthermore, from this perspective, we will divide the conclusions into three ranges:

- Low region – expected connection number is above 100,
- Middle region – expected connection number is between 100 and 2000,
- High region – expected connection number is above 2000.

For the low connection numbers region, the latency is quite similar among all implementations, so we will focus on the throughput, CPU, and memory utilization. So the first place takes the synchronous multithreaded implementation (Section 4.1.1), which shows the best throughput results on average. It is in the middle of the rating by CPU consumption. However, the memory usage is much more significant than for the other versions. If we are concerned about memory usage, our second-best choice would be the hybrid synchronous single-threaded implementation (Section 4.1.2). It is the most memory efficient, with no high CPU utilization and second-best throughput values.

Regarding the middle range, the latency, on average, is the same except for the synchronous multithreaded version (Section 4.1.1), which has colossal latency values, jumping from around 10 to around 200 milliseconds. The best choice in this range will be the asynchronous threaded version (Section 4.1.5), with the best throughput results. The downside here is high CPU usage and average memory consumption. So if we care about CPU consumption, we have the second-best choice – hybrid synchronous single-threaded implementation (Section 4.1.2). It has second-best throughput results, low CPU and memory consumption, and highly stable and predictable latency values.

Now about the high client connection numbers region. The finest choice here for all parameters is the hybrid synchronous single-threaded implementation (Section 4.1.2). It has the best throughput, latency stability, and efficient CPU and memory usage. The 10000 client number, which is supported now, is not able to fully load the multithreaded implementations (see Figure 4.6), so the overhead introduced by using threads does not pay off.

The hybrid synchronous single-threaded implementation (Section 4.1.2) is the best choice for a server with the requirements:

- Support up to 10000 simultaneous connections.
- Handle up to 333000 QPS with 10000 connected clients.
- Easy and cheap in implementation and maintenance.
- Work stably on real-world server load.
- Have relatively small latency values.

5.2 Future Improvements

We investigated the main implementation choices in this work and tested them on a real network setup. There are many options to extend the work by exploring other methodologies and improving the analysis of implemented and investigated versions. The possible further steps to improve this research are:

- Use BPF performance tools (Gregg, 2019) to investigate were are the bottlenecks of each implementation.
- Investigate the Kernel extensions which could be implemented in the scope of this thesis.
- Investigate the possible traffic offload options and implement them.
- Perform longer performance tests for more accurate results.
- Try out different testing tools or implement our own.
- Investigate and try out the possibility of configuring NIC Network ACLs to act as an echo server and analyze the results.

Bibliography

Andrew S. Tanenbaum, Herbert Bos (2014). *Modern operating systems*. 4ed. Pearson. ISBN: 0-13-359162-X,978-0-13-359162-0. URL: http://gen.lib.rus.ec/book/index.php?md5=15415ec4b87bc657692cf040890f5e4a.

Benchmarking Methodology for Network Interconnect Devices (Mar. 1999). RFC 2544. RFC Editor. DOI: 10.17487/RFC2544. URL: https://www.rfc-editor.org/info/rfc2544.

Cisco Systems, Inc. (2022). *Cisco Catalyst 3560 Series Switches*. [Model: WS-C3560-48TS-S V02]. URL: https://www.cisco.com/c/en/us/support/switches/catalyst-3560-series-switches/series.html (visited on 04/07/2022).

Demailly, Laurent et al. (May 2022). *Fortio*. Version v1.31.1. URL: https://github.com/fortio/fortio.

Echo Protocol (May 1983). RFC 862. RFC Editor. DOI: 10.17487/RFC0862. URL: https://www.rfc-editor.org/info/rfc862.

Esposito, Emanuele Giuseppe, Paulo Coelho, and Fernando Pedone (2018). "Kernel Paxos". In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pp. 231–240. DOI: 10.1109/SRDS.2018.00037.

Gregg, Brendan (2019). *BPF Performance Tools*. Addison-Wesley Professional. ISBN: 0136554822,9780136554820. URL: http://gen.lib.rus.ec/book/index.php?md5=4ff4065a0c77dd48cde5a6ff0b14cff2.

Ilie, L. et al. (2009). *Theoretical Computer Science*. Vol. 410. Elsevier. DOI: 10.1016/j.tcs.2008.09.019. URL: http://gen.lib.rus.ec/book/index.php?md5=d8cde2cf8e3ff817333bc35bc31e9a6c.

Karabyn, Petro (2019). "Performance and scalability analysis of Java IO and NIO based server models, their implementation and comparison". Bachelor's Thesis. Ukrainian Catholic University. URL: https://s3.eu-central-1.amazonaws.com/ucu.edu.ua/wp-content/uploads/sites/8/2019/12/Petro-Karabyn.pdf.

Kaufmann, Antoine et al. (2019). "TAS: TCP Acceleration as an OS Service". In: *Proceedings of the Fourteenth EuroSys Conference 2019*.

Kerrisk, Michael (2022). *The Linux man-pages project*. URL: https://www.kernel.org/doc/man-pages/ (visited on 05/30/2022).

Kohlhoff, Christopher et al. (Aug. 2019). *Boost.Asio*. Version boost-1.71.0. URL: https://github.com/boostorg/asio.

Krohn, Maxwell, Eddie Kohler, and M. Frans Kaashoek (June 2007). "Events Can Make Sense". In: *2007 USENIX Annual Technical Conference (USENIX ATC 07)*. Santa Clara, CA: USENIX Association. URL: https://www.usenix.org/conference/2007-usenix-annual-technical-conference/events-can-make-sense.

Mah, Bruce, Jef Poskanzer, Brian Tierney, et al. (June 2019). *iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool*. Version 3.7. URL: https://github.com/esnet/iperf.

Pasichnyk, Yuriy (June 2022). *Gaining efficiency in echo server performance by appropriate design and implementation choices*. Version master. URL: https://github.com/Fenix-125/hpc-echo-server.

Stanford VLSI Group (2022). *CPU DB Clock Frequency*. URL: `http://cpudb.stanford.edu/visualize/clock_frequency` (visited on 05/31/2022).

Stroustrup, Bjarne (2013). *The C++ Programming Language*. Fourth Edition. Pearson Education, Inc. ISBN: 9780321563842,0321563840,2013002159. URL: `http://gen.lib.rus.ec/book/index.php?md5=379DB9454D7EFF9ACB19DA9DA565D909`.

Wikipedia (2022a). *OSI model — Wikipedia, The Free Encyclopedia*. URL: `https://en.wikipedia.org/w/index.php?title=OSI_model&oldid=1089978003` (visited on 05/26/2022).

— (2022b). *Server (computing) — Wikipedia, The Free Encyclopedia*. URL: `https://en.wikipedia.org/wiki/Server_(computing)&oldid=1086236959` (visited on 05/05/2022).

Zhang, Shungeng et al. (2020). "The Impact of Event Processing Flow on Asynchronous Server Efficiency". In: *IEEE Transactions on Parallel and Distributed Systems* 31.3, pp. 565–579. DOI: `10.1109/TPDS.2019.2938500`.