UKRAINIAN CATHOLIC UNIVERSITY

BACHELOR THESIS

Performance and scalability analysis of Java IO and NIO based server models, their implementation and comparison

---

Author:
Petro KARABYN

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

APPLIED
SCIENCES
FACULTY

Lviv 2019

Declaration of Authorship

I, Petro KARABYN, declare that this thesis titled, "Performance and scalability analysis of Java IO and NIO based server models, their implementation and comparison" 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:

UKRAINIAN CATHOLIC UNIVERSITY

Faculty of Applied Sciences

Bachelor of Science

Performance and scalability analysis of Java IO and NIO based server models, their implementation and comparison

by Petro KARABYN

Abstract

This thesis focuses on the scalability and performance analysis of the web applications built with the IO packages present in the Java programming language. One of the objectives is to compare the thread-oriented and event-driven models and examine the implications of using them in Java IO and NIO-based systems. Three types of concurrency strategies are proposed for dealing with blocking and non-blocking IO and then applied in practice to build a chat engine. Results of the experiments show that in the case of active long-lived connections the difference in throughput between Java IO and NIO is marginal. However, NIO has an edge when it comes to large amounts of simultaneous connections and offers minimal memory consumption.

Acknowledgements

I would like to express words of gratitude to my project advisor, Oleg Farenyuk, for guidance throughout my student years. Your professionalism and deep competence in the wide area of subjects have always inspired me to put a lot of effort into learning. Your valuable advice has helped a great deal and directed me in my professional career path.

Also, I would like to acknowledge current and past members of the Applied Sciences Faculty and everyone who at some point contributed to the growth and success of the Computer Science program. The work you did will continue to give back to society for the years to come.

Contents

Declaration of Authorship **ii**

Abstract **iii**

Acknowledgements **iv**

1 Introduction **1**

2 Background **4**
2.1 Thread-based model 5
2.2 Event-driven model 7
2.3 Reactor pattern 9
2.4 The C10K Problem 11

3 Related works **12**
3.1 Duality argument 13
3.1.1 Message-oriented system 13
3.1.2 Procedure-oriented system 14
3.1.3 Differences and a modern look 15
3.1.4 Conclusions 15
3.2 SEDA 16
3.2.1 High-level overview 16
3.2.2 Types of stages 17
3.2.3 Anatomy of a stage 18
3.2.4 Conclusions 19
3.3 Thousands of Threads and Blocking I/O 20

4 Scalable I/O in Java **21**
4.1 Java IO and NIO 22
4.1.1 IO streams vs. NIO channels and buffers 22
4.1.2 Blocking vs. non-blocking IO 23
4.1.3 Selectors and design implications 24
4.1.4 Summary 25
4.2 I/O concurrency strategies 26
4.2.1 Multithreaded Blocking I/O 26
4.2.2 Single Threaded Non-Blocking I/O 27
4.2.3 Multithreaded Non-Blocking I/O 29
4.3 I/O trends in enterprise software 30

5 Experiments **32**
5.1 Implementation 33
5.2 Test setup 34
5.3 Real world simulation test 35
5.3.1 Metrics 36

5.4 High load test 38
5.4.1 Metrics 39
5.5 Analysis 41

6 Conclusion **44**

Bibliography **46**

List of Figures

2.1 Thread-based server design. This figure illustrates a thread-based model for a server where each incoming client request is processed in a separate thread. The request related computations and internal I/O operations are done by the allocated thread which sends the response to the client upon completion. This model is also referred to as thread-per-request. . . . 6
2.2 Event-driven server design. The main thread processes the events from the event queue passing them to the event handler that would execute the associated code. The events can be requests from clients or any other notifications from external systems. Note, that this is a conceptual diagram with a single event handler displayed, in practice, there can be multiple event handlers or one per each type of the expected events. . . . 8
2.3 Reactor pattern UML . . . 10

3.1 Conceptual illustration of a message-oriented system. Processes have message ports to receive messages in queue and their own data. A process operates data contained by another process by means of explicit message passing mechanism . . . 13
3.2 Conceptual illustration of a procedure-oriented system. Global data is shared across the processes. Monitor encapsulates the data and provides methods for synchronized access. In order to manipulate the global data, a process has to first acquire the mutex. Any other process can't change the state of the global data until the mutex is released. This way the shared data is manipulated one process at a time and sychronization is achieved. . . . 14
3.3 A high-level overview of the SEDA processing scheme. . . . 16
3.4 SEDA types of stages . . . 17
3.5 Anatomy of a SEDA stage. Consists of an incoming queue for the events, an event handler, a dynamically sized thread pool, and a controller responsible for scheduling, monitoring the load, and adjusting the parameters at runtime. . . . 18

4.1 Stream oriented IO . . . 23
4.2 Buffer oriented IO . . . 23
4.3 Thread-per connection design schema of standard IO-based application 24
4.4 Design schema of a NIO application with a Selector . . . 25
4.5 Multithreaded Blocking I/O . . . 26
4.6 Single Threaded Non-Blocking I/O . . . 28
4.7 Multithreaded Non-Blocking I/O . . . 29

5.1 Test setup . . . 34
5.2 Chat server latency . . . 36

5.3 Comparison of chat server latency at 10 000 concurrent clients 36
5.4 Chat server CPU utilization 37
5.5 Chat server heap usage 37
5.6 Chat server threads 38
5.7 Chat server throughput 39
5.8 Chat server outgoing messages 39
5.9 Chat server CPU utilization at high load 40
5.10 Chat server heap usage at high load 40
5.11 Chat server threads at high load 41

List of Tables

3.1 Duality mapping. 15
3.2 Duality mapping: modern look. Adapted to thread-based and event-driven concepts (Li and Zdancewic, 2007) 15

4.1 Differences between Java IO and NIO 25

List of Abbreviations

CPU	**C**entral **p**rocessing **u**nit
FD	**F**ile **d**escriptor
GUI	**G**raphical **u**ser **i**nterface
IOCP	**I/O c**ompletion **p**orts
IO	**I**nput/**O**utput
JDK	**J**ava **D**evelopment **K**it
JSR	**J**ava **S**pecification **R**equest
OS	**O**perating **S**ystem
MOS	**M**essage-**o**riented **s**ystem
POS	**P**rocedure-**o**riented **s**ystem
Req/s	**R**equests per **s**econd
RAM	**R**andom **a**ccess **m**emory
SEDA	**S**taged **e**vent-**d**riven **a**rchitecture
TCP	**T**ransmission **C**ontrol **P**rotocol

Definitions

Process an independent active program managed by the operating system that is not sharing the same address space with other programs.

Thread a smallest unit of execution that can be managed by a scheduler and may share memory and resources with its peer threads within a process, often referred to as a *lightweight process*.

Message an item of data that is sent to a specific destination.

Event a signal emitted by a component upon reaching a given state or any other action detected by a program.

Event-driven programming a programming paradigm in which the execution of a program is determined by the events it listens and reacts to.

Synchronous execution a mode in which a program is executed line-by-line in a predetermined sequence of tasks, where each subsequent task has to wait before the previous one is finished.

Asynchronous execution a mode in which a series of instructions don't have to be executed in a sequential order, but where separate tasks can be taken out of the current flow and executed in a background.

Dedicated to my family, who always supported me every step of the way.

Chapter 1

Introduction

The growth of the modern web continuously raises the bar for the web server scalability requirements and sets the new challenges for the tech engineers. Scalable systems are built to handle rapidly increasing amounts of load and adapt to the growing demands. Proper I/O and concurrency strategies are detrimental to the overall performance of the web servers and their ability to scale. This thesis analyses the I/O mechanisms in Java and also discusses the benefits and the drawbacks of different concurrency strategies from the scalability and performance point of view.

As massive amounts of data are getting transported over the network every single moment, it is crucially important to process that data efficiently. Present-day hardware can move gigabytes of data between the CPU and the main memory in a matter of a second. Disks are, obviously, slower, but still, offer respectable speeds of over 100 megabytes per second. Network, however, is several orders of magnitude slower than the hardware the web applications are hosted on. The pitfall that any application wants to avoid is making the high-performing CPU wait for the relatively slow network. This is where the differences between the blocking and non-blocking I/O come into play. Java platform offers the standard IO package which is of a blocking nature and the NIO package that allows building applications that handle the I/O operations in a non-blocking way.

The topics of scalability and performance of web applications built with Java IO and NIO are closely linked to the threads vs. the events debate that was started in the academia decades ago and remains to be an area of split opinions up to this day. Lauer and Needham (1979) argued that the procedure-oriented and the message-oriented systems are duals of each other, neither of them is inherently preferable, and they are similar in performance. John Ousterhout's (1996) stance, however, was in favor of the event-driven systems, claiming that threads are a bad idea for most purposes. Matt Welsh (2001) in his Ph.D. thesis tried to combine the two concepts into one architecture. Just two years later, Rob von Behren (2003b) together with the instructor Eric Brewer, who also supervised the Matt Welsh's work on SEDA argued that the events are a bad idea for high-concurrency servers showing that a well-designed thread-based server can match or even outperform the event-driven equivalent. Elmeleegy et al. (2004) joins the camp of event supporters and proposes a new asynchronous I/O interface for the event-driven programming. Li and Zdancewic (2007) also contribute to the debate by presenting a language-based technique to unify the event-driven and the multithreaded models for building massively concurrent internet services. Threads vs. events is the topic that the researchers can't settle on. One article demonstrating the superiority of event-driven systems is often followed by research favoring threads a couple of years later. One of the goals this thesis pursuits is to build a bridge between the above-mentioned debate in academia and the application of the discussed ideas in the Java programming language. This is the area that lacks the research. Therefore, we try to analyze the arguments for and against the use of threads and events when building highly scalable systems, verify a subset of them in practice and take an unprecedented look on the matter not attempting to join either of the two groups, but providing insights and guidance into which model might be better for Java applications under different scenarios. In particular, a chat server based on Java IO and NIO with different concurrency strategies is implemented and performance tested, which is an interesting example of a system with long-lived connections.

The structure of this work is the following. First, we examine the internals of the thread-based and the event-driven models for the web servers and also get familiar with the notion and importance of the C10K problem. In the next chapter, we describe in more detail the key related works in the threads vs. events debate as

well as what has already been researched and measured in the area of Java IO and NIO performance and scalability. Then we briefly overview the technical concepts behind the IO and NIO packages and the differences between them. Also, we propose the concurrency strategies that can be used in conjunction with blocking and non-blocking IO. The practical part is the implementation of a chat engine with the proposed concurrency strategies based on Java IO and NIO. Finally, a set of experiments is conducted with the customly implemented TCP-level load tester for the needs of this work, and the results are presented.

Chapter 2

Background

There has been a long-lasting debate over which programming model is best suited for building highly performant and at the same time scalable server applications. The more traditional way to design a system that would simultaneously service a large number of clients at a reasonably low processing time per request was to use a thread-based approach. The core idea is straightforward, which is to dedicate a separate either newly created or a reused thread on a server to process an incoming client request. This introduces a certain level of isolation between the requests and is well-suited and natural for the applications operating at the HTTP protocol level. The Apache web server was designed to take advantage of the multithreaded model and has dominated the web servers market share for over two decades, being created in 1995 and taking the lead since 1996. However, the thread-based model has been challenged by the researches (Ousterhout, 1996; Lee, 2006) as well as by the modern requirements to resource consumption, scalability, and performance of the systems available on the web. Another quickly gaining in popularity alternative is the event-driven approach. One of the examples of its successful application is the Nginx web server using the event-driven architecture and the reactor pattern at its core to serve a larger number of concurrent connections than Apache is able to. Nginx has been maintaining a rising trend in the share of web servers used by the websites and became an attractive option for applications that need to scale well, while Apache was on its fall in the 2010s (*Netcraft Web Server survey* 2019).

This chapter will briefly overview the characteristics of the thread-based and event-driven systems, the difference between using threads vs events, and techniques when working with each of them. Also, this work later discusses the tools that Java provides for handling I/O intensive tasks and how to use them to build an application based on each of the introduced models. One of the examples of such system that relies heavily on the I/O performance, has to support many clients concurrently, process requests in real time, and can be built with each of the models is a chat server. To put things into a larger perspective, a C10k problem first introduced in 1999 and still relevant today, but at much larger scales that brought more attention to the asynchronous and event-driven programming is explained.

2.1 Thread-based model

A typical internet service may experience several thousands of active clients with independent requests from each of them coming multiple times per minute or as often as every second. Every such request needs resources on the server to be processed and the quicker the better. A common way to satisfy such needs is to allocate a separate worker thread for the accepted request that will process it. When the number of active threads requiring CPU time to perform a computation is larger than the number of cores on the machine, which is normally the case, the operating system will make sure that each of the threads gets the CPU resources by context switching between them.

It is relatively easy to program systems using the thread-per-request model since the processing of each request is isolated from the other ones, the order of code execution is typically sequential and regardless of the number of clients the code is implemented as there is only one. Not every modern programming language support multithreading, yet most of the ones used for the backend do and they provide convenient to use abstractions for threads. Also, the described approach is similar if used in conjunction with processes instead of threads, however threads are more lightweight than the processes, generally have a smaller memory footprint and are

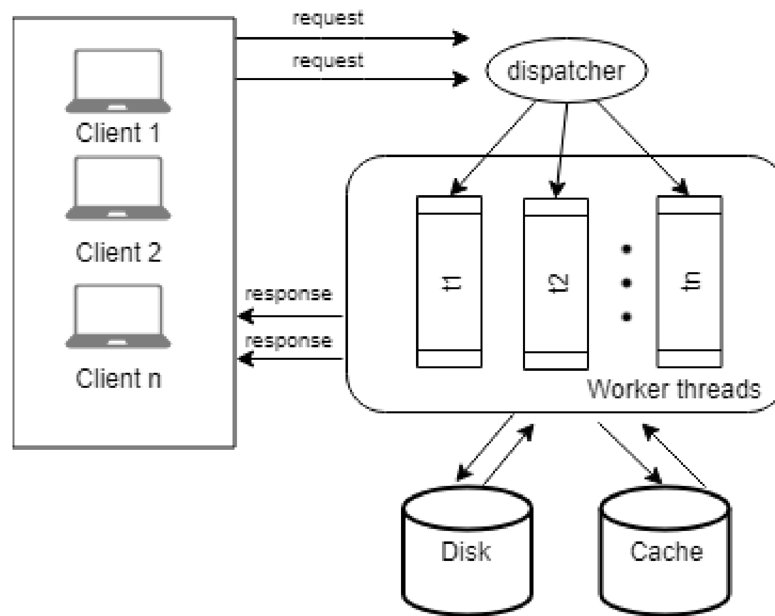


FIGURE 2.1: Thread-based server design. This figure illustrates a thread-based model for a server where each incoming client request is processed in a separate thread. The request related computations and internal I/O operations are done by the allocated thread which sends the response to the client upon completion. This model is also referred to as thread-per-request.

more widely applied. Therefore, the approach is commonly referred to as thread-based, but not limited only to threads. In case of shared data structures and other resources between the threads, the synchronization techniques need to be used in order to prevent data races.

Even though the thread-based model is not complex to implement, allows to process multiple tasks simultaneously, and threads are widely supported by the programming languages it still has certain disadvantages and limitations. Lee (2006) argued that the use of threads makes programs hard to understand, unpredictable, and nondeterministic. A proper programming style has to be maintained to prune the unwanted nondeterminism, which can be achieved using the synchronization techniques. However, synchronization can be difficult to get right and prone to mistakes that lead to unreliability, not to mention hard to debug problems like deadlocks caused by improper synchronization. At the same time, the excessive use of synchronization reduces the share of code executed in parallel and adds locking overheads causing the reduction of performance and doesn't yield a satisfactory result. Threads can be referred to as double-ended swords if you consider both their usefulness and inevitability in the multi-core era as well as the list of potential hazards brought into the program.

Another case against the thread-based model that doesn't involve the human factor is memory consumption. Each thread maintains its own stack in memory. Therefore, the memory usage will grow linearly to the number of threads. At the minimum this will have additional cost in terms of infrastructure, at most this can cause availability problems of the application when the number of threads and their stack size are not limited and too many are created reaching the memory limits of a server. Besides memory concerns, under heavy load when large numbers of threads

are created to handle the requests, they start to compete for the CPU time. This results in a constant context switching between them taking up the CPU resources that otherwise could have been used for computation, adds scheduling overheads, and increases the chance of cache misses. The described issues raise scalability concerns of the proposed model when the number of clients you are expecting to service at the same time is counted in tens of thousands or over that.

With all of the advantages and disadvantages of such an approach, the thread-based model and variations of it have been the most popular choice for server applications for two decades and still remain to be. It is a suitable way for most of the typical server-side systems and helps to utilize the multi-core nature of modern CPUs, however, if the requirements are very demanding to the memory usage or the system has to handle bursts of high loads with a large number of simultaneous clients there are other more effective alternatives to consider.

2.2 Event-driven model

The event-driven approach to writing programs is not a particularly new or recent concept. It was known long before the multi-core processors became the new standard and takes its roots from the message passing systems. The later could be characterized by having a static number of autonomous processes with either no or very limited shared data in memory between them, where the processes would communicate with each other by sending and receiving messages through the established messaging system. The invoker sends a message to another process that would execute a certain sequence of calls associated with the message received, or in other words, react to a message.

In a modern day, the event-driven approach is a largely different concept than the message-passing. The two terms should not be used interchangeably and are not mutually exclusive. However, they share a similar idea and architecture principles. Event-driven architectural design has largely evolved from and was influenced by the earlier research in the message-passing systems. The use of processes as part of a single application has shifted to the use of threads, which on the contrary share the address space. Another key difference between the two is that in message-passing systems the invoker sending the message knows the recipient. In simpler words, a message has the address, whereas in the event-driven system the events are raised for all the subscribers without knowing who will respond to it. *Reactive Manifesto* distinguishes a difference between message and event as follows.

> A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state.

A fundamental component of the event-driven design is the single-threaded event loop. It provides a mapping of a single thread to multiple connections drastically reducing the number of threads required for the system to run. The thread takes an event from the event queue emitted by the external source, passes it to the event handler for processing and then either takes the next event in the queue repeating the same procedure or waits until the new event will be in the queue.

The main thread's job is to loop continuously, processing the event-queue and delegating the event processing. Event handler's responsibility is to know how to process certain types of events and execute the corresponding action. It is crucial that the processing of the event is quick and doesn't hold up the event loop, which would cause performance degradation and increase in latencies.

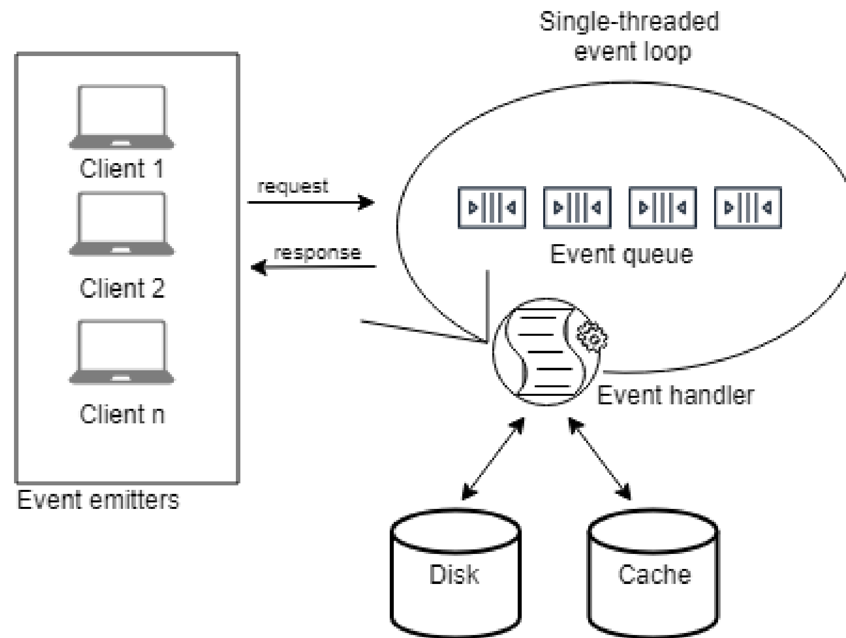


FIGURE 2.2: Event-driven server design. The main thread processes the events from the event queue passing them to the event handler that would execute the associated code. The events can be requests from clients or any other notifications from external systems. Note, that this is a conceptual diagram with a single event handler displayed, in practice, there can be multiple event handlers or one per each type of the expected events.

This leads us to the next essential component in the event-driven design which is non-blocking execution of I/O operations. This way the request to perform I/O operation returns immediately even if it is not finished allowing not to block the execution flow and introducing the I/O level concurrency. Once the I/O operation is completed the OS will raise an event notifying that processing can be continued. This approach relies on the support and performance of non-blocking I/O mechanisms present in the OS, such as epoll in the Linux kernel.

Up to this point, it was described the general workflow of an event-driven system and how it exploits the I/O concurrency and takes advantage of the non-blocking I/O mechanisms in the operating systems. Another question is what happens if the event processing is not bound to the I/O performance, but simply requires longer computation time and how can you take advantage of the multi-core CPUs? For that reason, in practice, the event-driven systems are not limited to the single event handler, but typically have N event-handlers each responsible for a specific event and allocated a separate thread to run. This way there is a limited low number of threads not dependent on the number of simultaneous connections which run in parallel utilizing the multi-core CPU resources and minimizing the context switching overheads present in the thread-based model both at the same time. An application designed in an event-driven manner is not limited to the number of threads that can be effectively managed by the operating system, and the scalability depends instead on the performance of the single-threaded event loop (Zhu et al., 2015).

A disadvantage of the event-driven approach would be that it is challenging to program. Behren et al. (2003b) argued that it complicates the control flow of the application and makes it difficult to restore the sequence of execution and hard to

debug for the programmer, while the control flow of the thread-based application is more natural. Also, using the non-blocking asynchronous interfaces generally requires more skill from the developer.

To sum up, the event-driven model achieves I/O concurrency through the use of non-blocking I/O mechanisms present in the operating systems that signal the completion of the operation, the CPU parallelism is achieved through running multiple event handlers in separate threads, however the event-driven systems have a more complex flow of execution and are harder to implement. One of the crucial characteristics is the limited number of threads required resulting in low resource consumption and potential for high scalability.

2.3 Reactor pattern

The reactor is an architectural design pattern that provides an event-driven way for an application to handle network requests from one or many simultaneously connected clients originally described by Schmidt (1995). It is as also known as Dispatcher or Notifier. The server listens for the incoming requests and dispatches the specific types of requests to the appropriate handlers responsible for servicing them. The Demultiplexer maintains information about all connected clients and is used by the Reactor to select a subset of clients that are ready for an event to occur without blocking on I/O. The typical events are ready to connect, ready for read, ready for write. This design pattern allows to avoid creating a new thread for each request or connection by using a synchronous demultiplexing of events and dispatching them to the handlers.

The following are the core components of the pattern.

Reactor

Handles registration and removal of the event handlers and initializes the Demultiplexer instance. Runs in a loop waiting for notifications from Demultiplexer on the occurred events and dispatches the processing of them to the corresponding handlers.

Handle

A resource that provides an interface for retrieving the input data from it or outputting the system response data to it. Most commonly represents network connections identifying the client socket endpoints, but can be other system resources.

Demultiplexer

Sends a notification to the reactor, when servicing operation can be performed on the handle without blocking the IO operation. For example, calling read() on a resource that at the moment of the call doesn't possess any data for reading would result in a blocking operation. Demultiplexer guarantees that the I/O system call related to the Handle wouldn't block by performing a select on the handles and returning the ones ready for the event to be processed or waiting until an event occurs in case none are present. Under the hood commonly uses the select() system call or the analog provided by the operating system.

Event handler

Each event handler is responsible for a specific type of an event and performs the actual processing of it in a non-blocking manner.

Reactor pattern can be implemented using the java.nio package. The mapping from the pattern components to the associated building blocks in Java is as follows.

Demultiplexer represented by the java.nio.channels.Selector class.

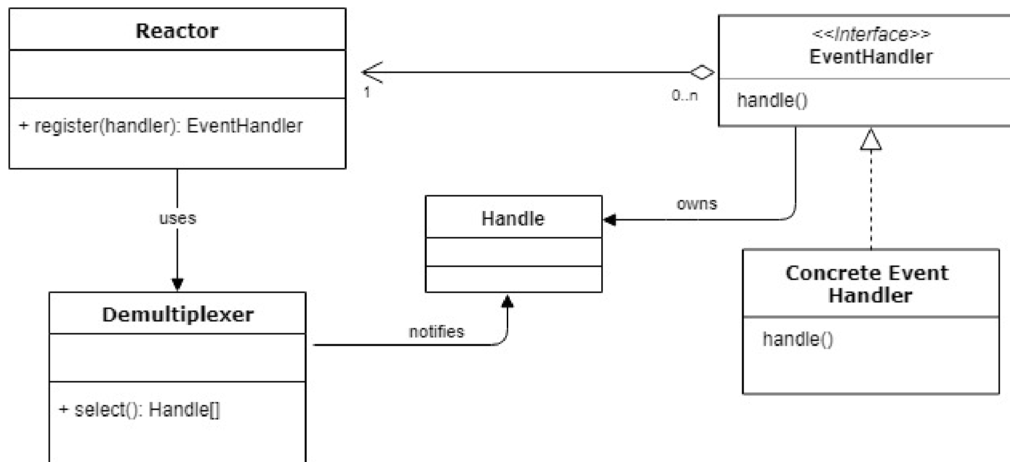


FIGURE 2.3: Reactor pattern UML

Handle represented by the instance of java.nio.channels.SelectionKey class retrieved as a result of Selector.select() call.

Event handler a class that provides certain computation logic related to the specific event type and executes it. The event types are SelectionKey.OPACCEPT, SelectionKey.OPCONNECT, SelectionKey.OPREAD, and SelectionKey.OPWRITE.

Reactor represents an instance of a class that opens a server endpoint for connecting to clients, maintains a reference to the Selector instance and runs in a loop calling Selector.select() and dispatching the events to the handlers as long as the server is alive.

There are many possible variations of Reactor pattern implementation that leave opportunities to tune the application for best performance. The basic Reactor design can be implemented in a single-threaded version, where accepting new clients, dispatching and handling the events are all done in one thread. However, in the modern day, this is typically used as a case study, but not in practice as a single thread is unlikely to be able to keep up with large bursts of new incoming clients. To achieve the desired degree of scalability additional threads are introduced. The multithreaded design variations with worker thread per each handler, the dedicated thread pool for all handlers, and multiple reactor threads are possible (Lea, 2003). The worker per handler approach is simpler to implement and allows to delegate the event processing from the reactor to handlers speeding it up. The thread pool is the further enhancement saving all of the advantages of the worker approach, but offering better tuning and control over resources. Multiple reactor threads is an advanced variation that can be used alongside the multithreaded handlers. It involves introducing additional reactors into the system, where the main reactor accepts new connections and distributes them to the sub-reactors each running in a separate thread and having their own Selector (Demultiplexer).

Reactor pattern is at the core of Node.js and Vert.x project, used by Nginx and Jetty servers, and is an inevitable part of applications exploiting the advantages of asynchronous IO. It helps to achieve the goal of scalable and robust servers and played its role in overcoming the C10K problem.

2.4 The C10K Problem

In 1999 the busiest file transfer websites were servicing 10 000 simultaneous clients with a Gigabit Ethernet pipe, and the web was only about to burst in popularity around the world starting to reach more ordinary everyday users. Murphy and Roser (2019) show how the global number of internet users grew exponentially from 413 million in 2000 to almost 2 billion in 2010. The term of the C10k problem was born when Kegel (2006) first published a cognominal article in 1999 on this problem. He announced that "it is time for web servers to handle ten thousand clients simultaneously." The original article was revised multiple times and became an influential resource on web server scalability.

He motivates his considerations by showing that hardware is continually evolving and getting more affordable, which in turn creates more opportunities to manage high connection concurrency. Based on a reasonable cost of a machine with a 1000 MHz CPU, 2 GB of RAM, and a 1000Mbit/sec network card, Kegel argues that 10 000 parallel clients are feasible, yielding 100KHz, 200Kbytes, and 100Kbits/sec per request - quite enough for 8kb of payload data. In practice, the vast majority of servers were far away from those numbers at that time. He then examined OS internals and shared notes on how to implement systems supporting thousands of clients.

The following I/O strategies are proposed:

1. Serve many clients with each thread, and use nonblocking I/O and level-triggered readiness notification.
2. Serve many clients with each thread, and use nonblocking I/O and readiness change notification
3. Serve many clients with each thread, and use asynchronous I/O and completion notification.
4. Serve one client with each server thread
5. Build the server code into the kernel
6. Bring the TCP stack into userspace

The C10k term has been reinforced ten years later when the Urban Airship company driven by the business needs to handle extraordinary high numbers of idle connections in parallel attempted to serve 500.000 connections on a single node (*Urban Airship: C500k in Action* 2010). Then the practically achieved upper limits were raised by WhatsApp (Reed, 2012) in 2011 entering into an era of C1M.

Chapter 3

Related works

3.1 Duality argument

The origin of the thread-oriented vs. event-driven design debate can be tracked down to the late seventies. One of the first published works in the area that laid the groundworks for the future researchers was the article *On the Duality of Operating System Structures* (Lauer and Needham, 1979). It compared the characteristics of message-oriented systems and the procedure-oriented systems as the authors refer them. The described procedure-oriented and message-oriented systems are synonymous to the thread-based and event-driven systems correspondingly.

3.1.1 Message-oriented system

The message-oriented system can be characterized as having a small, static number of processes, and efficient facilities for passing messages between the processes. Also, a mechanism for queuing messages at destination processes until they can be acted upon has to be present. There is no concurrent access to data structures. Peripheral devices are treated as processes. The priority of a process is statically determined, and no global naming scheme is useful.

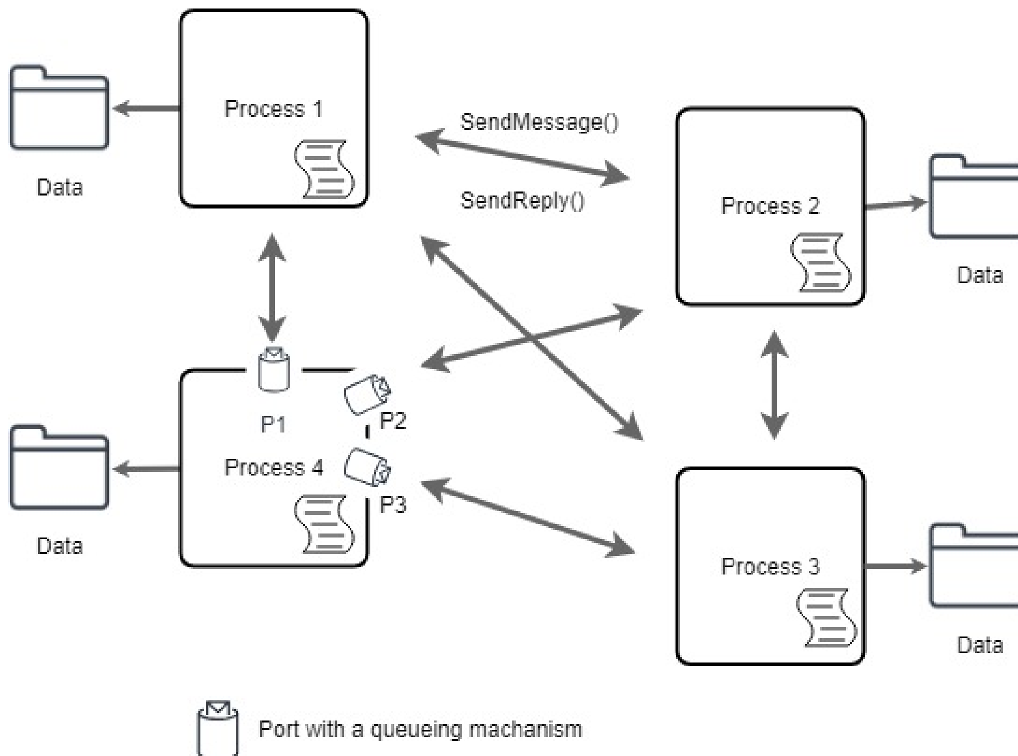


FIGURE 3.1: Conceptual illustration of a message-oriented system. Processes have message ports to receive messages in queue and their own data. A process operates data contained by another process by means of explicit message passing mechanism

Processes support four message transmission operations:

- **SendMessage**
- **AwaitReply**
- **WaitForMessage**

- **SendReply**

In a nutshell, a process template of a message-oriented system consists of local data and algorithms, certain message ports, and sends messages to specific message channels referring to other processes. Message channel is a destination of a message, and a port is a type-specific queue leading to the process.

3.1.2 Procedure-oriented system

The procedure-oriented system can be characterized as having a large number of small processes, which unlike in the message-oriented system is not static and can rapidly change, communication is achieved by using a direct sharing and interlocking of data, context of execution is identified with the function being executed. Proper synchronization and congestion control are at the core of such system and associates with waiting for locks. Data is shared directly, and locks should last for short period of time. Control of peripheral devices is in form of manipulating the locks. Priority is dynamically determined by the execution context, and global naming and context play an important role.

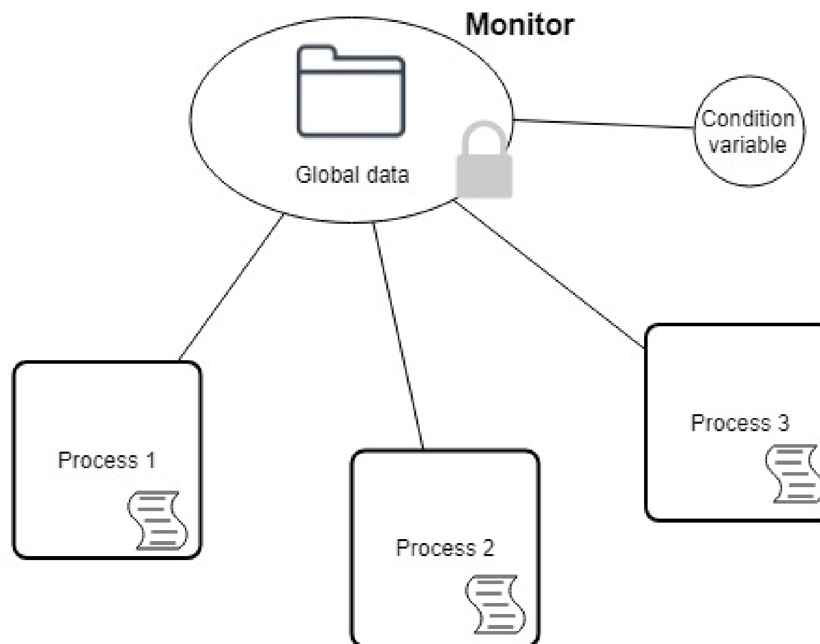


FIGURE 3.2: Conceptual illustration of a procedure-oriented system. Global data is shared across the processes. Monitor encapsulates the data and provides methods for synchronized access. In order to manipulate the global data, a process has to first acquire the mutex. Any other process can't change the state of the global data until the mutex is released. This way the shared data is manipulated one process at a time and sychronization is achieved.

Procedures implement algorithms and access global data. The procedure has a straightforward call facility:

- Synchronous call - regular call
- Asynchronous call - Fork (creation) and Join (termination)

Two more important components that make up the procedure-oriented system are modules and monitors. The former are collections of procedure and data, and

the latter are modules with locking facilities. New and Start calls instantiate and run modules correspondingly.

3.1.3 Differences and a modern look

The differences between the two systems as described in the original article can be summed up in the following table.

Message-oriented system	**Procedure-oriented system**
Processes: **CreateProcess**	Monitors: **NEW/START**
Message channel	External procedure id
Message port	Entry procedure id
Send msg (immediate); **AwaitReply**	Simple procedure call
Send msg (delayed); **AwaitReply**	**FORK**, **JOIN**
WaitForMessage statement	Monitor lock
Selective waiting	Condition variables, **WAIT**, **SIGNAL**

TABLE 3.1: Duality mapping.

As already mentioned POS's resemble the thread-based model and MOS's correspond to the event-driven model. Therefore, a refreshed modern mapping is proposed.

Event-driven	**Thread-based**
event handler	thread continuation
event loop	scheduling
event	exported function
dispatching a reply	returning from a procedure
dispatching a message, awaiting a reply	executing a blocking call
awaiting messages	waiting on condition variables

TABLE 3.2: Duality mapping: modern look. Adapted to thread-based and event-driven concepts (Li and Zdancewic, 2007)

3.1.4 Conclusions

The argument is made that a procedure-oriented system can be decomposed into a message-oriented system and vice versa; they are duals of each other and similar in performance. The authors argue that by mapping a program from one model into the other, the logic is not affected and the semantic content of the code remains equivalent.

Three main observations made are:

1. Both models are duals of each other. A program implemented in one model can be transformed equivalently into a program based on the other model.

2. They are logically equivalent, even though the syntax and concepts differ between the two.

3. The resulting performance should be virtually identical, provided that the same scheduling strategies are used.

Even though the arguments made are sound, there is often a mismatch between the theory and practice. Modern-day implementations of either of the models are unlikely to yield similar results. The performance of any system would largely depend on the hardware and the OS it runs on. Moreover, since the time the article was published both the hardware used by servers and the operating systems have evolved drastically. Multicore CPUs are the standard, many programming languages have native support of threads, and threading libraries like NPTL drastically improved thread performance and reduced the cost of idle threads and context switches.

3.2 SEDA

Staged event-driven architecture is a term that appeared as a result of Matt Welsh's PhD-thesis titled "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services" (2001). Even though, this notion started spreading in the early 2000s, some of the ideas at its core were applied in practice years earlier. It is a compromise between the thread-per-request and the asynchronous event-driven models. SEDA is a middle ground solution that in many cases allows to combine the advantages of both models and get the best of the two worlds.

3.2.1 High-level overview

Let's break down the internals of SEDA and introduce the key terms.

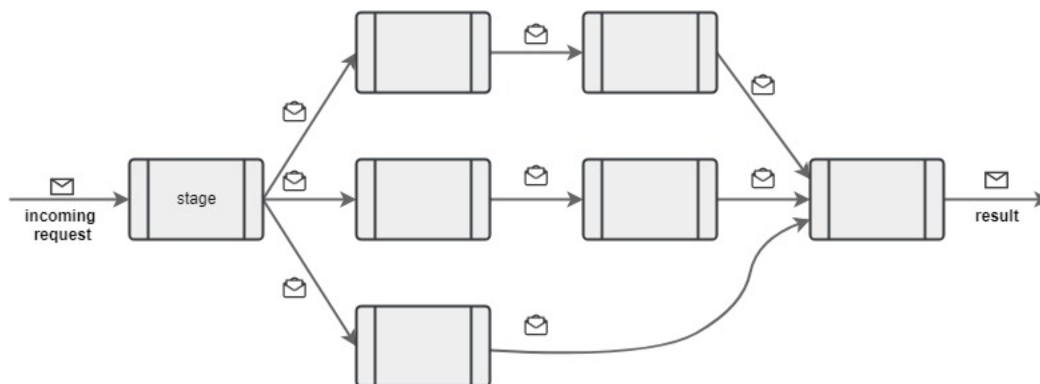


FIGURE 3.3: A high-level overview of the SEDA processing scheme.

Each rectangle is a stage with its unique logic. Events travel through queues between multiple stages. A request is put into an incoming queue initiating the execution of a set of commands at a stage. Then new events are generated that are processed at the latter stages until the processing is finished at the final one. The illustrated sequence of actions starting from the point when an incoming request is at the initial stage up to a result at the last stage can be called a SEDA transaction. The execution of the incoming request must be asynchronous.

Stage is a procedure processing an event. It is a so-called black-box where random messages (events) go in, processing takes place, and another events go out as a result.

Queue is a buffer of incoming events at an underlying stage. The events are passed between the stages and are queued at the beginning of each stage before getting processed.

Event is an arbitrary message. Typically the context of a request that is processed in a SEDA system is encapsulated into these events, and those, in turn, are passed between the stages.

This kind of architecture is very flexible. The number of stages applied is arbitrary. This way, large systems with complex logic can take advantage of more stages, while for simpler applications up to 5 stages may be enough. Also, an application can be tuned for best efficiency by adjusting the number of stages implemented or by independent self-regulated tuning at each stage, which is discussed later. Each event doesn't have to pass through the same sequence of stages; there can be specific paths for different types of events. Another advantage is the modularity achieved through breaking down an otherwise monolithic system into a sequence of stages.

3.2.2 Types of stages

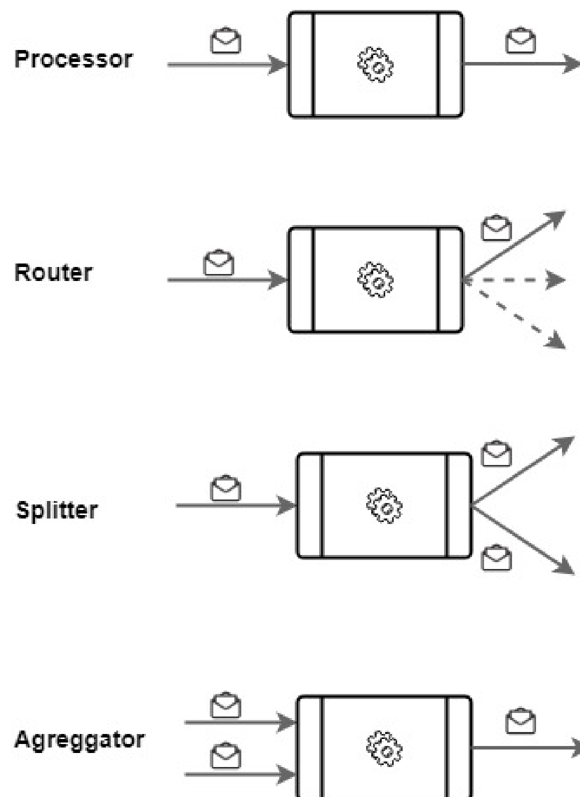


FIGURE 3.4: SEDA types of stages

Four main types of stages constitute a SEDA system:

1. **Processor**. Receives one event and outputs one event. The relationship is one to one. Passes a result to a specific subsequent stage. Any processing can take place inside, or any external resources like a database or a file read can be used.

2. **Router**. Receives one event and outputs one event. The relationship is one to one. However, based on the logic can choose out of multiple subsequent stages which one the event should be routed to.

3. **Splitter**. Receives one event and outputs multiple events. The relationship is one to many. Consequently, the total number of events in the system becomes larger.

4. **Aggregator**. Receives multiple events and outputs a single event. The relationship is many to one. May wait for a certain amount of context-related events to come in and then aggregate them into one or chose one out of many which is sent as an output.

These stage types provide a simple and straightforward way to architect a system with the required functionality.

3.2.3 Anatomy of a stage

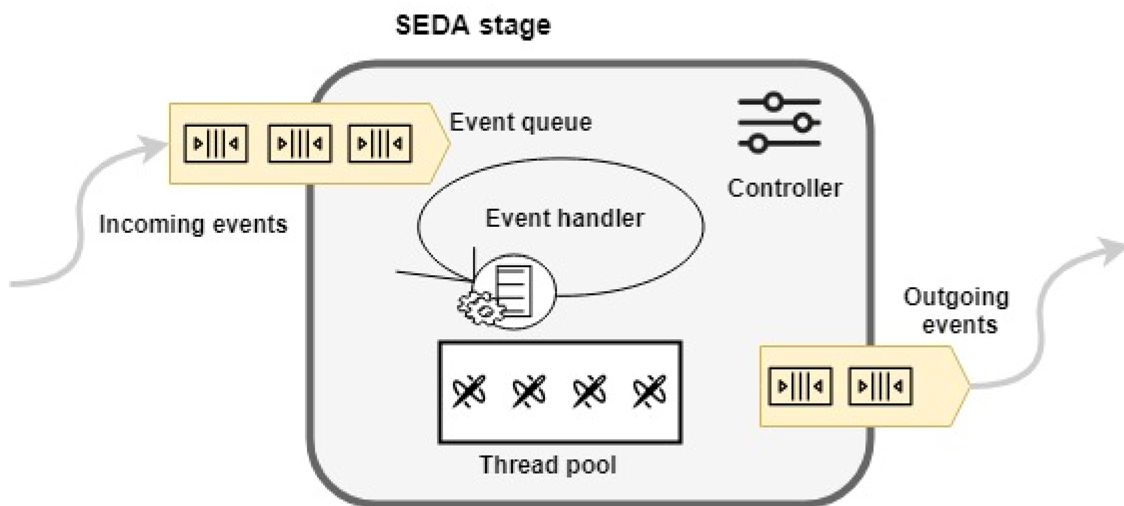


FIGURE 3.5: Anatomy of a SEDA stage. Consists of an incoming queue for the events, an event handler, a dynamically sized thread pool, and a controller responsible for scheduling, monitoring the load, and adjusting the parameters at runtime.

Incoming event queue is simply a data structure for temporarily storing the events until they are processed. However, an important aspect is that the queues may be bounded. This way when the queue is full, an incoming event may be rejected and further handled according to the rejection policy, which may be sending an error back or performing a specific function. Bounded queues are optional, but they enable the use of the backpressure and load shedding mechanisms where needed. Besides holding the events the queues between the stages play a role of an explicit control boundary and a certain level of isolation between the stages. The downside is the increased overhead of enqueuing and dequeuing the events instead of heaving the modules invoke each other directly.

Event handler is the brains of the stage. It contains logic for the processing of the events of an expected type.

The thread pool and threads it encapsulates are the muscles of the stage. Typically a limited small number of threads is used within a stage.

The most interesting part is the controller. The purpose of it is to manage the performance of a stage by dynamically adjusting scheduling and resource allocation. The goal aimed to achieve is to prevent the developer from having to manually

tune the performance iteratively looking for the trade-offs, offer the best ratio between the performance and the allocated resources on each stage at runtime, and keep the application running within its limits in a sense turning it into an alive creature constantly changing its shape best suited for conditions it's in at the current moment. For the sake of simplicity, there is only one controller illustrated, but it should logically be differentiated into two kinds. The thread pool controller monitors the incoming queue length and the idle period of threads and adjusts the thread pool size. When a queue length goes up, a limited amount of new threads can be spawned. When the threads start idling exceeding the threshold, a thread pool shrinks its size killing some of the threads. The batching controller, in turn, adjusts the number of events that the event handler processes on a single iteration. For certain types of operation, it's way more efficient to aggregate them into one batch and then process them together instead of one at a time. When the output rate decreases, the batching factor is bumped up. The beauty of a batching controller is that the changes on one particular stage don't entail any changes on the other one, i.e. the system os locally optimized.

The described dynamic tuning is independent of the functional purpose of a stage. For example, one stage can be doing data encryption, and the other one can be responsible for database access. In the first case we are dealing with a CPU-bound task, so there is no reason to have a thread pool of a size larger than the number of physical cores on a machine. In the second one, the criteria may be entirely different. The point is that each stage is optimized individually.

3.2.4 Conclusions

SEDA was an attempt to combine the best of the two worlds from the threaded and event-driven server designs. The advantages of the proposed architecture are numerous, but the most import ones are scalability under conditions of large numbers of concurrent requests, explicit control of overload, and a design that advocates for the separation of concerns and better modularity of the system. Among the disadvantages of SEDA are increased latencies in low load environments because of the queuing overheads and communication between the stages. Also, an overly large amount of stages would lead to excessive context switching, since each stage has its own not shared thread pool and the event gets executed in different threads while passing through the stages, which the author later admitted in his reflections on SEDA (Welsh, 2010). Multiple studies have been conducted since the appearance of SEDA comparing server performance of the thread-based, event-driven, and hybrid approaches, where SEDA-based servers performed with various success (Pariag et al., 2007; Behren et al., 2003a; K. Park, 2006).

It can be stated without exaggeration that work on SEDA became the golden standard in server design soon after its publication at least for the next couple of years. While it may not be the most performant server ever benchmarked by academics anymore, presented concepts still remain to be highly relevant nowadays. Not only did Matt Welsh come up with a new architecture for high concurrency servers, but also, he implemented a non-blocking IO library as part of the SEDA prototype called Sandstorm written in pure Java. One of the outcomes was a contribution to the development of Java NIO, where he participated as a member of JSR-51 expert group. It's hard to overestimate the importance of Welsh's work as he started a small revolution in server architecture and left the footprints in the Java

ecosystem as well. The ideas presented in SEDA laid the groundworks for the internal architecture of The Apache Cassandra Project and found their continuation in server frameworks such as Apache MINA and Netty.

3.3 Thousands of Threads and Blocking I/O

At the tech conference in 2008, Paul Tyma, Ph.D. in Computer Engineering and a senior engineer at Google at the time questioned whether the asynchronous server models built with Java NIO actually perform and potentially scale better than the ones built in a synchronous multithreaded manner using the standard IO and presented the empirical results of the experiment (Tyma, 2008).

The author picks a list of not necessarily valid, but common reasons to choose a thread-per-connection IO over NIO or vice versa and debates whether they are accurate. To some extent contrary to the popular opinion, a number of statements listed below are revealed to be not valid.

1. Asynchronous I/O is faster.
2. Thread context switching is slow.
3. Synchronization among threads will kill you.
4. Thread per connection doesn't scale.

A data transfer speed test of NIO server against the IO server running on Linux 2.6 and Windows XP operating systems revealed that the IO server has over 25% advantage in throughput, dismantling the myth that the asynchronous IO is faster. Another couple of tests show that the thread context switching is relatively cheap as with the development of NPTL in Linux it became much more efficient and even though the synchronization does have an overhead, the uncontended synchronization is cheap and non-blocking data structures perform very well in a concurrent environment. Also, the synchronous multithreaded server models can scale reasonably well when done right with the example of the Mailinator server.

Presented findings may not be as surprising nowadays as they were at the time, but chances are it is partially due to the experiments conducted by researchers such as Paul Tyma that the common knowledge about IO and NIO based systems is now more accurate.

Chapter 4

Scalable I/O in Java

4.1 Java IO and NIO

IO is an inevitable building block of any server-side application. Whether a write to a file or a data transfer over a socket has to be performed, the corresponding tools need to be used to achieve that. Java language has two independent sets of IO packages to offer. The standard Java IO package introduced since JDK 1.0 and Java NIO initially implemented as part of JSR-51 introduced in JDK 1.4 and then extended in JDK 1.7. NIO came to fill in a missing spot for the asynchronous non-blocking IO as an alternative to the original package.

The ideas of event-driven I/O are similar to the ones used when building GUIs. Some parallels can be drawn regarding the components of the java.awt package. Nonetheless, besides user interface programming, it is less common to see an event-driven approach at the core of a server-side Java application than the more usual thread-oriented one with blocking I/O. However, if you have to handle tens of thousands connections, the system has to be real-time, you are after low resource consumption, or any of listed is true, and you just want to take advantage of the asynchronous non-blocking way to perform IO, Java NIO is a powerful tool for such job.

In this section, we will briefly examine the internals of Java IO and NIO, and fundamental differences between the two, explaining them one by one. We won't go into the implementation details and overview all the interfaces of each package since this is out of the scope of this thesis and the main focus is the performance and scalability perspective on the I/O in Java. Let's dive right in.

4.1.1 IO streams vs. NIO channels and buffers

The first fundamental difference is when you want to transfer data from the Java application to an external destination or the other way around in case of the standard IO you achieve that through streams and in NIO you put/retrieve data to/from buffers which then gets transported through the channel.

With streams, you read bytes of data sequentially one at a time. You are not in control of what part of data present in a stream to read, and you can't move back and forth inside a stream. So, for example, you can't read just a chunk of data in the middle or at the end of the stream ignoring all the rest. To get to the last bytes in the stream you have first to retrieve all the preceding data, and when you have read the data once, it's not cached anywhere and you can't read it again. So, if that data has to be accessed multiple times after retrieval it is job of a programmer to cache it in a data structure of choice.

The buffer oriented approach in NIO is a bit different. The data is first read into the buffer from a channel before it can be retrieved for processing. The buffer interface allows you to move through the data inside the buffer and offers more flexibility. On the other hand, the programmer has to keep track of the positioning and marks of a buffer and make sure the data inside isn't accidentally overridden before getting processed. The data put into the buffer is getting transported through the channel. Channel is like a pipe that connects the byte buffer and the entity on the other end, which is typically a file or socket. Also, unlike when working with streams, you never put data directly into the channel, but always interface the buffer first that works with the channel. It may seem like buffer is just a wrapper for the primitive type array. It's partially true, but nio buffer is more than just a wrapper. Under the hood, buffers encapsulate arrays of primitive data with the positioning attributes and helper methods that allow flexibility in the way data can be retrieved, also native

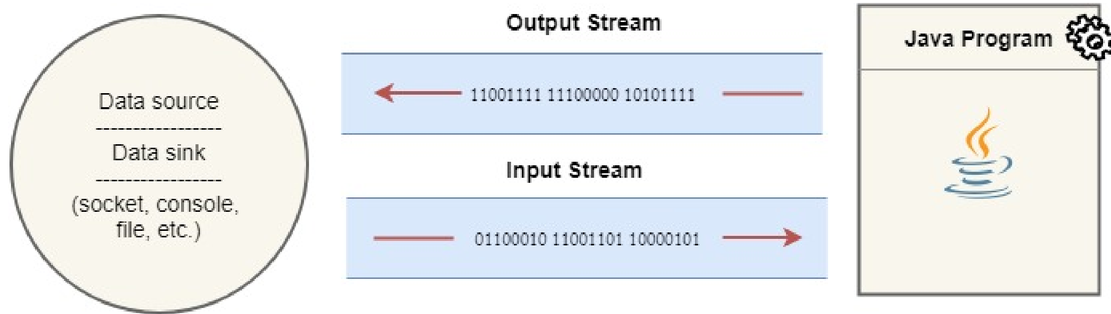


FIGURE 4.1: Stream oriented IO

1. Read data from input stream
2. Write data to output stream

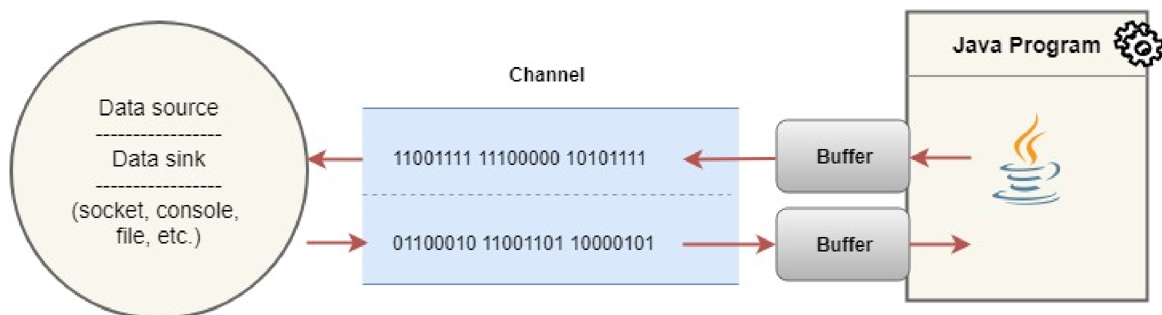


FIGURE 4.2: Buffer oriented IO

1. Channel writes data into buffer
2. Program reads data from a buffer
3. Program writes data into a buffer
4. Channel reads data from a buffer

implementations can allocate the space for the buffer directly in the kernel memory, or use another efficient implementation. Channel implementations as well are very close to the operating system, often use native code and vary significantly between the platforms. One more key difference is that channels, unlike streams, are bidirectional. In the case with the standard IO, you have to open one input stream and one output stream per connection, with channels you can read and write data through the same channel.

4.1.2 Blocking vs. non-blocking IO

Streams are blocking. Period. When you call a read or write operation on the stream, the thread executing the call gets blocked until the data is read or written. The thread cannot perform any other computation until the call returns. If read was called and data is never sent through the underlying stream, the thread is blocked indefinitely.

Channels offer a non-blocking mode of working with the I/O. A thread can execute a read on the channel asking for data and immediately get what's available or

nothing if data is not there. This prevents a lot of overhead, and a thread can carry on with the next tasks even when the read from a channel didn't result in any data retrieved.

4.1.3 Selectors and design implications

Selector is a third key concept in NIO of the same level of importance as buffers and channels. Selector provides I/O multiplexing functionality that allows examining events on multiple connected channels registered with the selector with a single thread. To make use of the select functionality on the application serving as a server, you have to:

1. Create an instance of *ServerSocketChannel*, open it, and set to non-blocking mode.
2. Open a *Selector*
3. Register the instance of *ServerSocketChannel* with the selector and specify the operation of interest (*OP_ACCEPT*)
4. Perform a selection (*Selector.select()*)
5. Accept the new connection to the server, which returns an instance of *SocketChannel* and register the corresponding socket channel with the selector specifying the operation of interest (*OP_READ/OP_WRITE*)

The subsequent selections will return a set of *SelectionKey* objects that contain a reference to the channels registered with the selector that are ready for the operation of interest. This way the selector serves as an indirect pointer through the *SelectionKey* object to the connected channels and provides a simple mechanism to manage a large number of channels with a minimal number or even a single thread In contrast, working with the standard IO package requires to have a dedicated thread per each connection to retrieve that data sent by clients when its there.

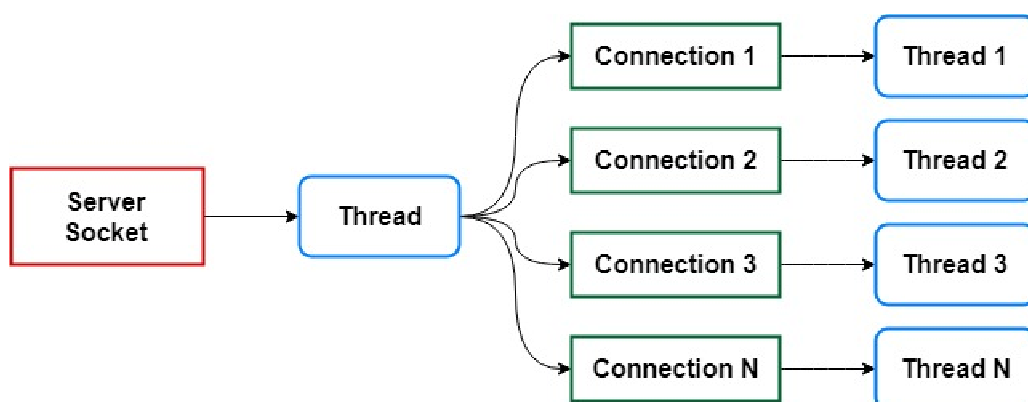


FIGURE 4.3: Thread-per connection design schema of standard IO-based application

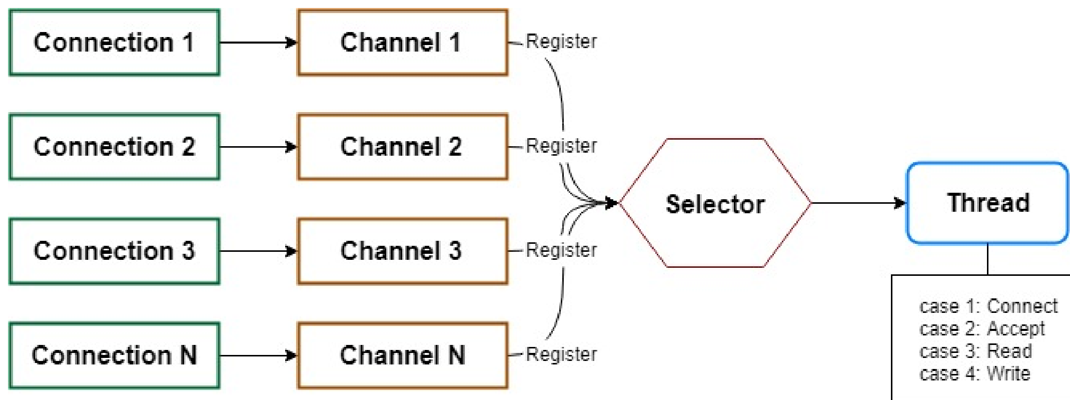


FIGURE 4.4: Design schema of a NIO application with a Selector

4.1.4 Summary

	IO	NIO
Data transfer	Stream oriented	Buffer oriented
I/O operations	Blocking	Non-blocking/blocking
Execution order	Synchronous	Asynchronous/synchronous
Programming model	Thread-oriented	Event-driven (when using Selectors)

TABLE 4.1: Differences between Java IO and NIO

Java NIO is often referred to as "non-blocking IO", however, this is not precisely accurate. The abbreviation stands for "New I/O" (Niemeyer and Leuck, 2013) and the Java Specification Request was titled as "New I/O APIs for the Java Platform". Fascinatingly, the official JSR page doesn't even mention a word about being blocking or non-blocking but emphasizes that it has to be scalable, buffered and make use of either asynchronous requests or polling to achieve the underlying requirement. Indeed, NIO can be used in a similarly to Java IO, configured in either non-blocking or blocking mode, and have either asynchronous or synchronous order of execution correspondingly. Nonetheless, being asynchronous and non-blocking is what the package is best known and suited for. This is also how it is most commonly applied in practice. Calling NIO event-driven to highlight the differences with the IO is generally valid, however; it's worth mentioning, that a program written with NIO doesn't have to be single-threaded, but can combine the advantages of the event-driven design with an appropriate threading model to handle computation-heavy tasks and utilize the multi-core CPUs. Speaking of the programming model used to build around IO or NIO, it doesn't have to be exclusively thread-oriented or event-driven, in this aspect the lines between the two are not clearly drawn, but worth distinguishing.

NIO offers better flexibility of working with data through buffers and means to efficiently handle thousands of simultaneous connections. Standard IO, on the other hand, is simpler to work with, still offers great performance on a moderate amount of simultaneous connections and might be better suited for dealing with massive amounts of byte data that doesn't need to be processed in chunks. The bottom line is, neither of the tools is better than the other in overall performance, NIO extended

the spectrum of functionality available in Java for managing input and output, and is promising from the application scalability point of view. When choosing one or the other, all the pros and cons have to be carefully considered.

4.2 I/O concurrency strategies

Before examining concurrency strategies, it is important to understand the IO operation flow. Java IO methods are native, so code is written specifically for each operating system and basically what it does is use an operating system specific IO API (system calls). IO system calls are executed in the kernel space, so after the process sends an IO request, an execution switches to a kernel mode and talks to a physical device driver (for example requests to read data into buffer). The device controller is responsible for filling the buffer. Reading is performed using pages and may load additional data. After it is done the OS copies the kernel space buffer into a buffer provided by a user or the user may share the same buffer with the OS and no coping is done (direct buffers).

4.2.1 Multithreaded Blocking I/O

A separate worker thread is spawned for each I/O request. The worker parses the request, performs a blocking I/O system call to retrieve the data, once the system call returns, the worker finishes its work by processing the retrieved data. Each thread (worker) performs system calls and computation independently from other threads within the process. Multiple threads utilize the modern multi-core hardware.

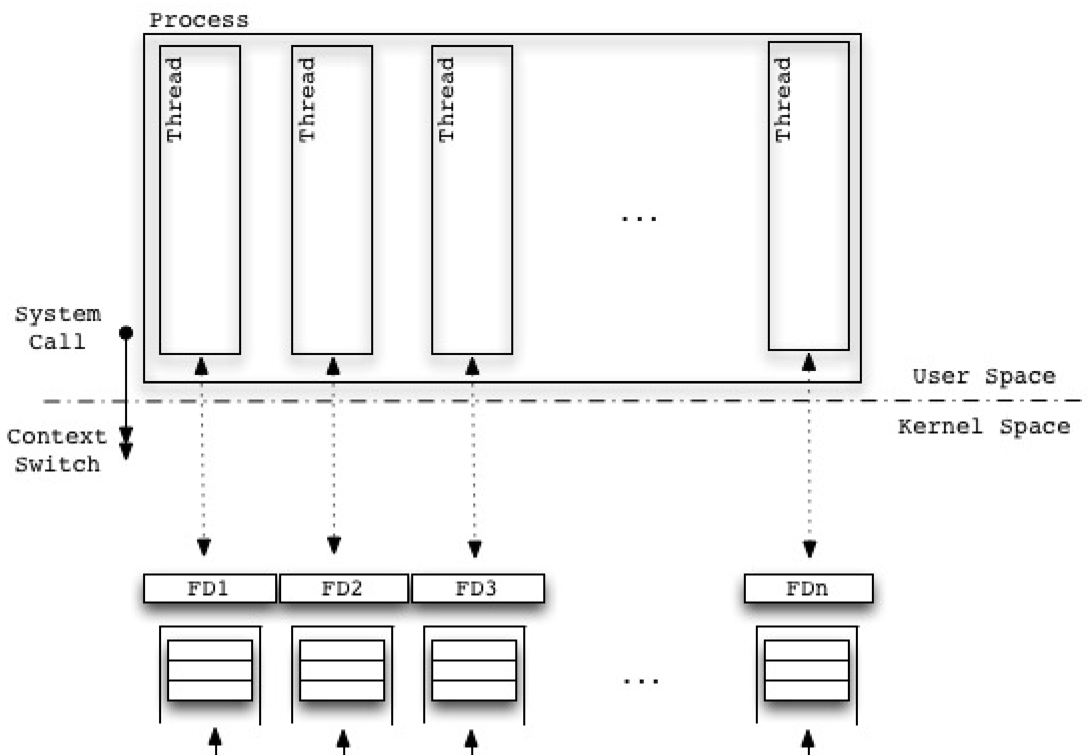


FIGURE 4.5: Multithreaded Blocking I/O

Algorithm

1. accept connection (perform system call, thread is blocked until a new connection arrives)
2. retrieve data (perform system call, thread is blocked until there is data sent
3. repeat (2)

Advantages

- Multiple threads help to better utilize the CPU
- Simple synchronous flow of execution
- No apparent bottleneck
- Easy to implement

Disadvantages

- Excessive context switching
- High memory usage (each thread stack has to be stored in memory, number of threads grows linearly)
- Not easy to debug due to large number of threads

4.2.2 Single Threaded Non-Blocking I/O

One single thread is created inside as part of the application that does all of the processing. Instead of making a direct blocking call to retrieve data from a specific connection and the underlying descriptor, an OS specific I/O multiplexing system call is performed that returns a set of ready connections which processing is guaranteed to be non-blocking. From the Java application side, in case of using NIO, a call to the *select()* method of *java.nio.channels.Selector* class is made, which in turn makes a system call on the corresponding system. *Selector.select()* implementation is platform dependent and Java provides different implementation for different OSs. In Linux the most efficient I/O multiplexing mechanism is *epoll()* which has a complexity of O(1), the equivalents of *epoll* are *kqueue* in FreeBSD and macOS, */dev/poll* in Solaris, and *IOCP* in Windows. The events returned from the *select()* call are then sequentially processed. Worth noting that, in case when dealing with CPU-bound events, having only one processing thread would have a drastically negative impact on the performance. Also, unlike in the blocking I/O model, this one has also a slight overhead of event management on the application side.

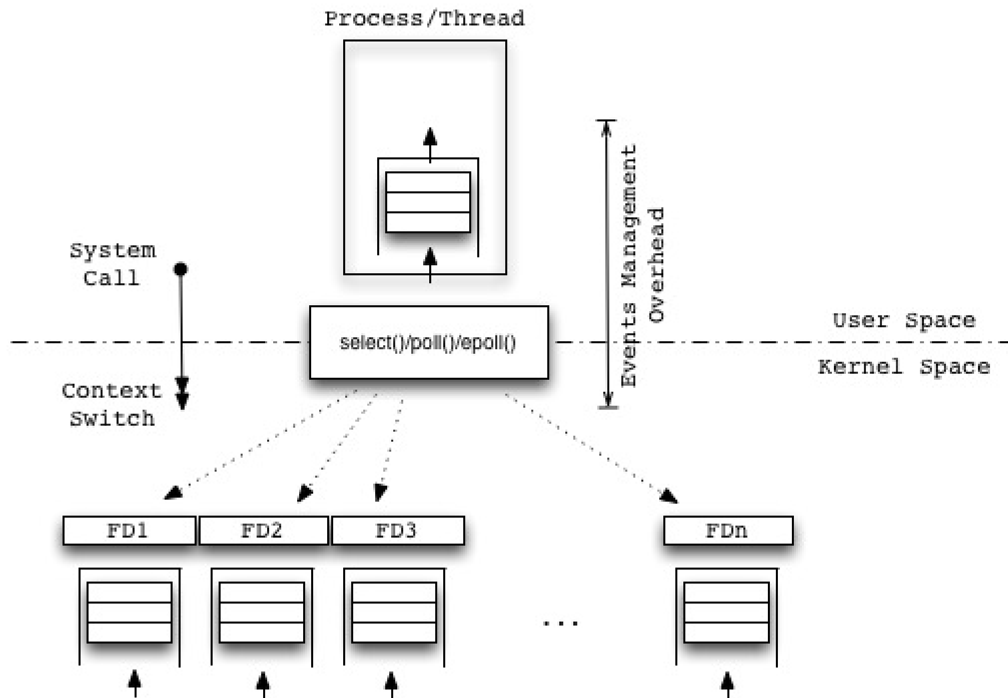


FIGURE 4.6: Single Threaded Non-Blocking I/O

Algorithm

1. Call *Selector.select()* (may block until an event ready to be processed arrives; performs platform specific system call)
2. Iterate through the selected events
3. If *OP_ACCEPT*, accept the connection, save the key (perform system call)
4. If *OP_READ*, retrieve the data (perform system call)
5. If there are any events left to be processed, perform next iteration (3-4)
6. Return to 1

Advantages

- Thread is never blocked as long as there are events to be processed
- Minimal memory usage
- No context switching between threads at application level, light context switches from user to kernel space

Disadvantages

- Doesn't utilize CPU sufficiently
- Inefficient for short-lived connections (too much overhead for event selection and management)
- Event loop may become a bottleneck

- Performance would degrade drastically in case of CPU bound tasks
- Moderately difficult to implement
- Difficult to debug

4.2.3 Multithreaded Non-Blocking I/O

Much like single-threaded non-blocking model in terms of a sequence of calls, except a pool of threads is used to process the vents in the event loop, eliminating the problem with CPU-bound tasks and utilizing the multi-core CPUs.

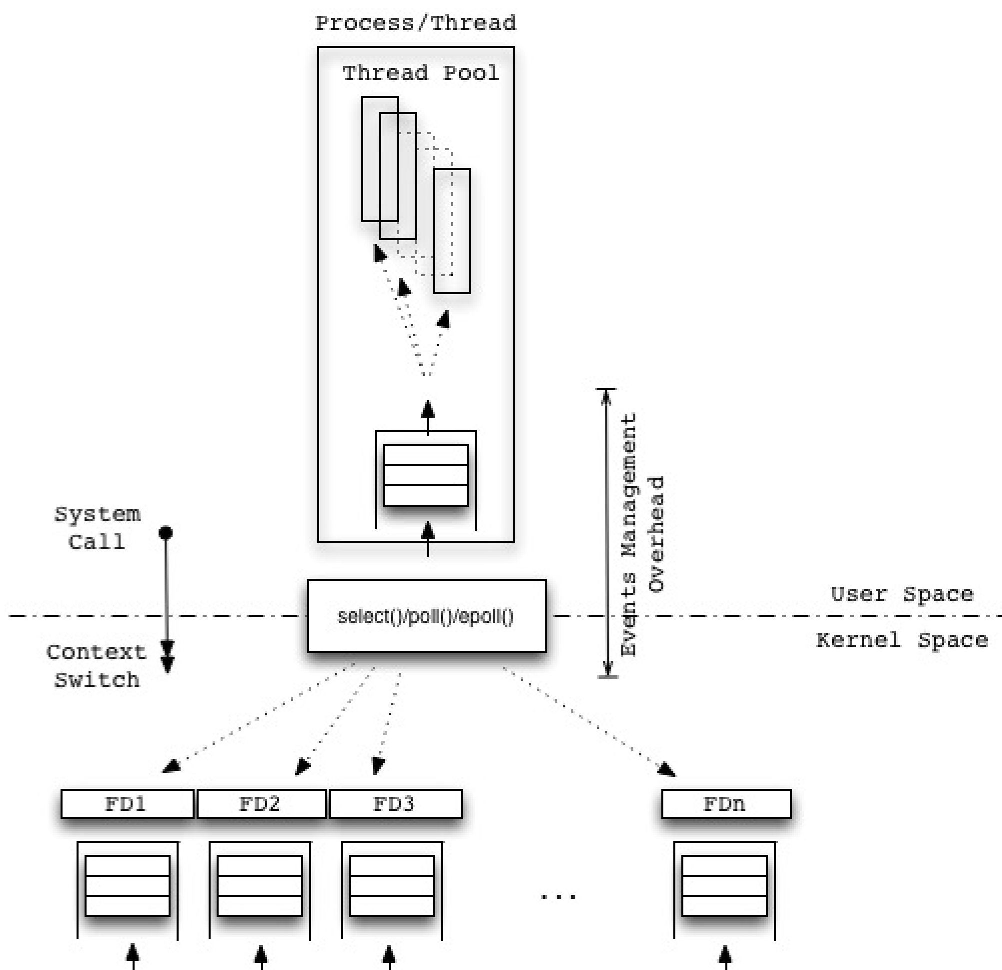


FIGURE 4.7: Multithreaded Non-Blocking I/O

Algorithm
Main loop:

1. Call *Selector.select()* (may block until an event ready to be processed arrives; performs platform specific system call)
2. Iterate through the selected events
3. Dispatch the event-associated task to a worker thread
4. If there are any events left to be processed, perform next iteration (3-4)

5. Return to 1

Worker thread loop:

1. If *OP_ACCEPT*, accept the connection, save the key (perform system call)
2. If *OP_READ*, retrieve the data (perform system call)

Advantages

- No thread is ever blocked as long as there are events to be processed
- Low memory usage
- Little context switching
- Utilize the multi-core CPU
- High scalability and performance in case of both IO-bound and CPU-bound tasks

Disadvantages

- Event loop may become a bottleneck
- Processing is delayed in time (before the task is processed it had to be dispatched to the worker thread and the thread had to be woken up)
- Requires thorough synchronization
- Difficult to implement
- Extremely difficult to debug

4.3 I/O trends in enterprise software

It is common to implement large enterprise Java applications in a thread-per-request or thread-per-connection manner, where threads perform complex computations on the backend side. The communication between the clients and the server is often hidden behind the layers of abstractions that frameworks provide, which often use the synchronous blocking I/O approach under the hood. It is unusual to see a developer working with barebones standard IO package offered in Java to implement an efficient way of network communication. The developer focuses on the functionality of a system and frameworks handle all the rest, including the "dirty" work of network communication, with little interference and tweaking if needed. When more and more users start using the system, the resource usage also increases, typically in a linear fashion. A single server reaches its limits for handling the load. The most common solution to the problem is to scale by adding new nodes, hence more overall processing power, which allows to handle more load, but at a price of more expensive bills for the hardware. Such way of developing enterprise software is acceptable as long as it achieves the business needs and satisfies the requirements. However, it neither means that it is the best way of doing things nor it is the most efficient one. Moreover, I would argue that in the nearest future this approach will shift from being a default choice to being used very sparingly, where it can actually be justified, becoming an extinct breed of legacy software. There are more performant and

resource-friendly approaches to meet the needs of the modern web and being a Java developer you don't to look far to see that the shift is already under way. One of the most popular frameworks in Java ecosystem, Spring first released Project Reactor in 2015 that is now featuring a reactive stack with non-blocking I/O and Reactor Netty, slowly drifting away from the older MVC servlet stack with synchronous blocking I/O. Recent performance test shows that the reactive stack is superior to the servlet stack and the difference becomes more evident as the number of concurrent users increases (Saxena, 2018).

Another example pointing out that there is a demand for the non-blocking I/O is the popularity of such frameworks as Play and Vert.x. Play is built on Akka and non-blocking I/O is one of its the core features that is achieved through the internal use of Netty and Akka HTTP, which in combination with the supported Actor concurrency model provide the modern tools for building highly-scalable application with low resource consumption. Vert.x in turn is event-driven and non-blocking at its core, which allows to handle many concurrent users with minimal amount threads. *Web Framework Benchmarks* 2018 show that, applications built with Vert.x outperform the competitors in most of the common types of tests.

In the world of web servers, nginx is continually conquering the trust of businesses and tech professionals. The work on this web server was started in 2002 by its founder and architect Igor Sysoev to solve the C10K problem. First public release was out in 2004 and the goal was reached in virtue of asynchronous event-driven architecture. Nginx quickly gained popularity due to being lightweight and offering high scalability opportunities under minimal hardware. Under the hood it actively uses multiplexing and notification of events, assigning specific tasks to individual processes. Incoming connections are handled through an efficient execution loop using a specific number of single-threaded processes called workers. Inside of each worker, nginx can handle many thousands of simultaneous connections and requests per second. Tests show its domination over Apache web server in memory usage as with an increase in concurrent connections nginx continues to consumes a fairly static and minimal amount of resources and an advantage in the number of requests handled per second (*Web server performance comparison* 2019).

Chapter 5

Experiments

The purpose of this part of work is to apply the earlier described concepts in practice, gain practical experience of implementing Java applications with different concurrency strategies based on the standard IO and NIO and run a series of experiments to get quantitative measurements of performance and scalability characteristics of the systems under test.

The main question of interest was to compare identical implementations of IO and NIO-based servers in the case of long-lived connections. For that reason, it was chosen to implement a chat server.

5.1 Implementation

Three types of engines of equivalent functionality were implemented:

1. Thread-per-connection IO engine (referred in the metrics section as **io**)
2. Single-threaded NIO engine (referred as **nio**)
3. Multi-threaded NIO engine (referred as **nio2**)

The thread-per-connection IO engine follows the multi-threaded blocking I/O concurrency strategy described in 4.2.1.

The single-threaded NIO engine follows the single-threaded non-blocking I/O concurrency strategy (4.2.2).

The multi-threaded NIO engine was designed to use one selector thread and a pool of handler threads of a fixed size and follows the multi-threaded non-blocking I/O concurrency strategy (4.2.3). Multiple configurations of thread-pools were attempted before running the final tests, and despite the simplicity, the fixed thread pool of size *number of processors * 4* offered the best trade-off between the context-switching overhead and the CPU utilization in a system with I/O-intensive tasks.

The system was designed to easily switch between the engine type being used based on the configuration file. All of the engines accept socket connections and operate at the TCP level.

The simplified flow of the execution is the following.

- IO engine

  Main thread:

  1. Open a *ServerSocket* at port specified from the config file
  2. Accept new connections
  3. When a new connection arrives create a *UserHandler* instance and spawn a new thread for it.
  4. repeat (2-3)

  UserHandler threads:

  1. Open input and output streams to the client socket
  2. Read user message
  3. Parse the message and add the associated username prefix to it
  4. Broadcast the message to all currently connected users
  5. repeat (2-4)

- NIO single-threaded engine:
  1. Open a *ServerSocketChannel* and configure it in a non-blocking mode
  2. Open a *Selector* and register it with the instance of *ServerSocketChannel*
  3. Perform a selection by executing *selector.select()*
  4. Iterate through the selected keys and for each key
     (a) if key is acceptable (incoming connection)
        i. Accept the connection and configure it in a non-blocking mode
        ii. Register the client socket channel with the selector
     (b) if key is readable (incoming message)
        i. Read user message
        ii. Parse the message and add the associated username prefix to it
        iii. Broadcast the message to all currently connected users
  5. repeat (3-4)
- The flow of NIO multi-threaded engine is similar to the single-threaded version, except that the thread pool of handlers was used, every event in the selection was dispatched to be executed by a handler thread from the pool and synchronization techniques were used for the methods executed in handler threads.

For purposes of measuring the time it takes to process an incoming message, the broadcast functionality was extended to send the *acknowledgement* to the sender after the received message was sent to all of the connected users.

5.2 Test setup

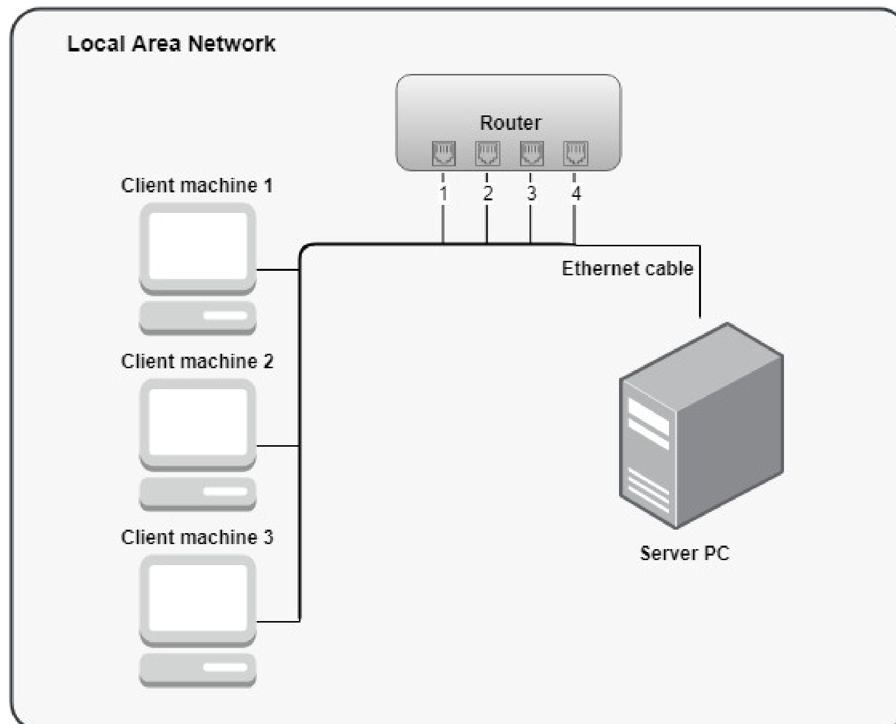


FIGURE 5.1: Test setup

Characteristics of components:

- Server PC:
  - CPU: Intel Core i7-3630QM CPU 2.40 GHz;
    cores: 4; logical processors: 8;
    L1 cache: 256 KB; L2 cache: 1.0 MB; L3 cache: 6.0 MB.
  - RAM: 12.0 GB DDR3
  - Network Card: 10/100/1000 Gigabit Ethernet LAN
  - OS: Microsoft Windows 10
  - Java version: 1.8.0_171
- Router: D-Link DIR-615; LAN 10/100BASE-TX

Note, the limitation of the router's LAN interface. For that reason, network throughput was continuously monitored, and no test was performed exceeding the limitation.

5.3 Real world simulation test

One of the ways to performance test an application is to simulate similar behavior as to when it is used in the real world. This test is performed using two separate client machines connected to the same local area network as the server. The first client machine is the *load generator* and the second client machine is the *metrics observer*.

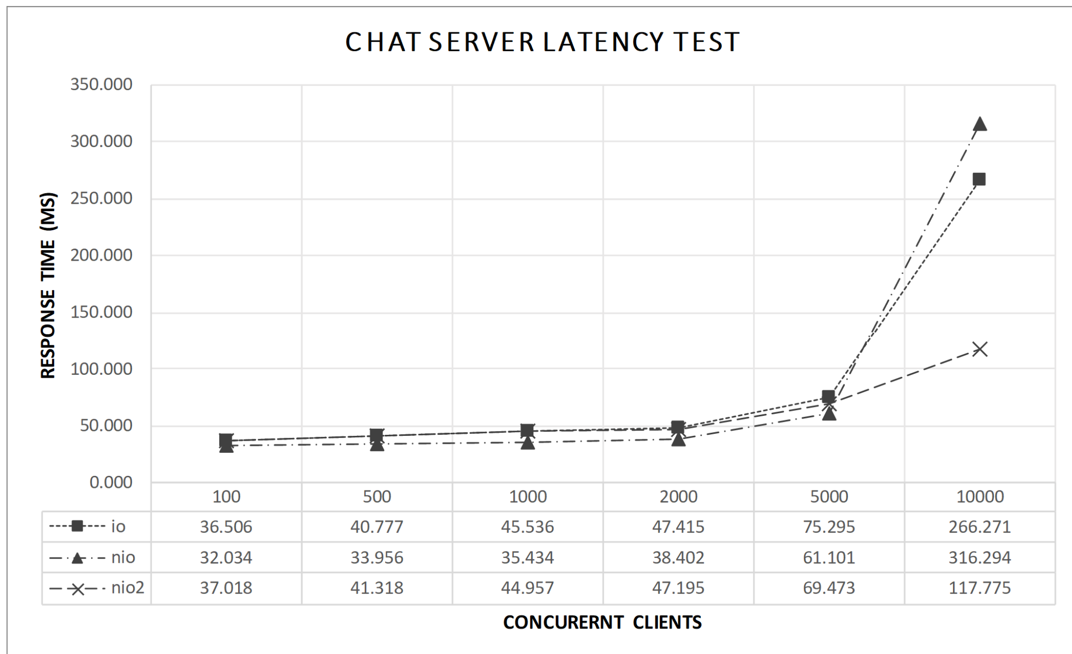
The external load generator connects a configured amount of clients to the server and starts sending messages to the server one by one from different connections at a specified rate. The load generator doesn't wait for the *ackowledgement* associated with the previous request to be received before sending the next request from the same connection. This ensures that the rate at which messages are being sent from the load generator is maintained at the same level as configured at the beginning of the test. The number of messages sent is proportional to the number of concurrent clients and is set to be (*number of concurrent clients / 100*) messages per second. Load generator's only responsibility is to connect a configured amount of clients to the server and generate the load.

The metrics observer in this test is used to gather the response time metrics. A full round trip of a request is measured, where the start time is right before the message is sent to the server, and the stop time is right after the *acknowledgement* is received. One additional client is connected to the server from the observer machine, and that client sends a message to the server, waits for the *acknowledgement* to be sent back, and proceeds with the next request. Each time a different random message is generated.

A warm-up was performed at the beginning of each measured test, and it was ensured that the results received are consistent across multiple iterations of tests. The CPU utilization, heap, and thread count metrics were gathered by using VisualVM profiler. Note, that the heap metrics represent not the overall heap size, but the used amount of memory.

5.3.1 Metrics

Latency



	100	500	1000	2000	5000	10000
io	36.506	40.777	45.536	47.415	75.295	266.271
nio	32.034	33.956	35.434	38.402	61.101	316.294
nio2	37.018	41.318	44.957	47.195	69.473	117.775

FIGURE 5.2: Chat server latency

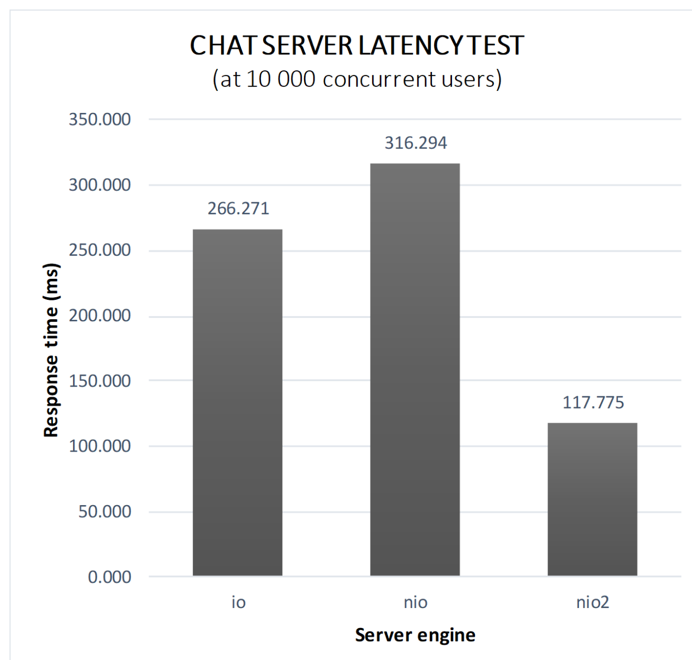
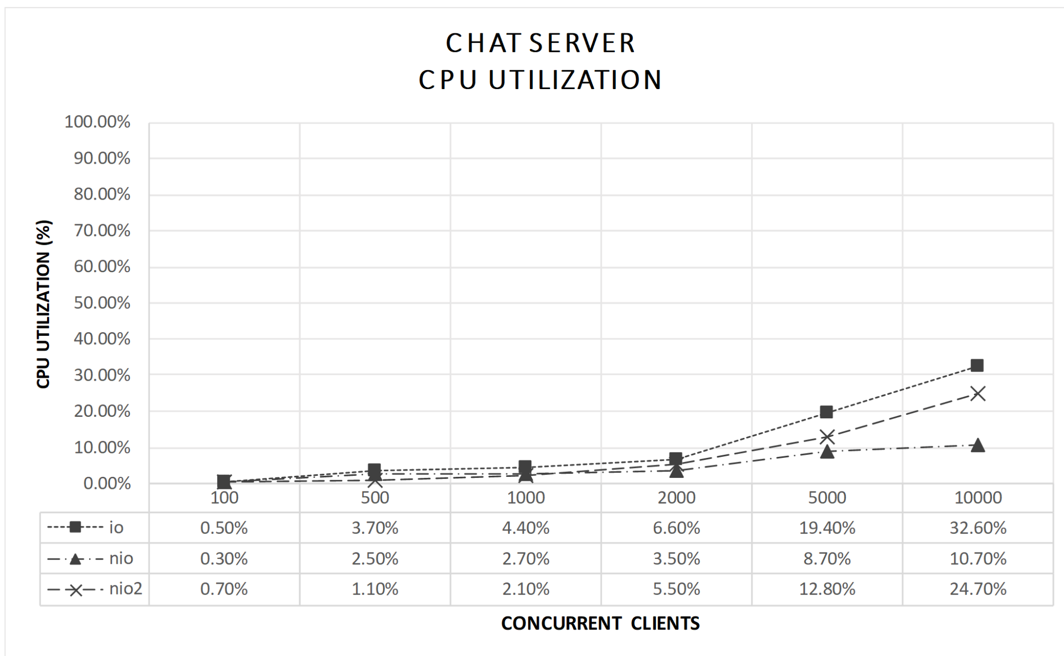


FIGURE 5.3: Comparison of chat server latency at 10 000 concurrent clients

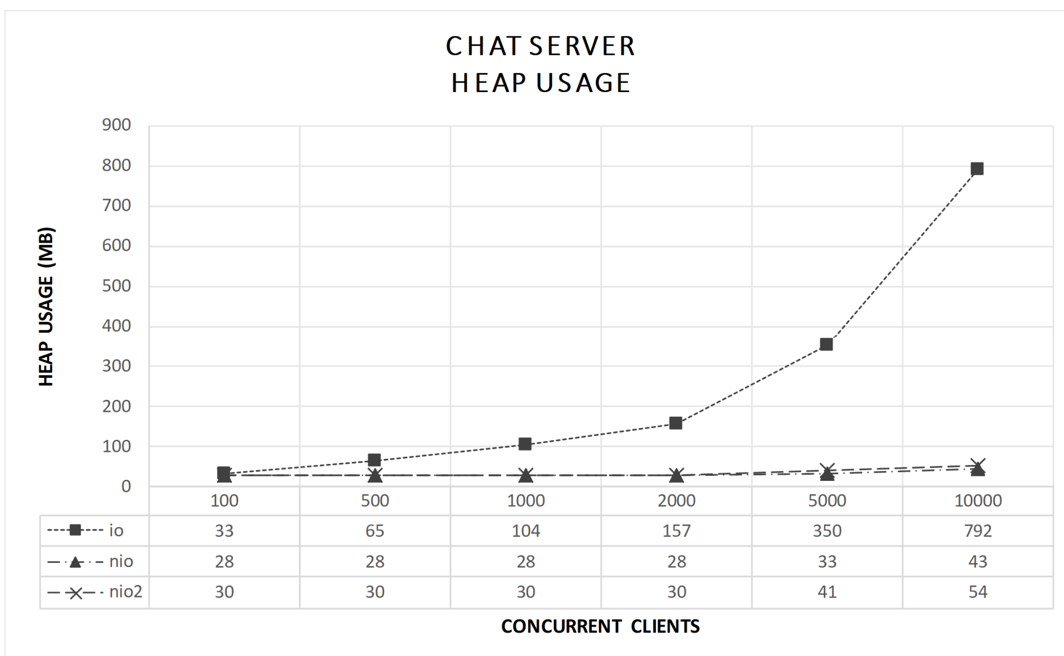
CPU utilization



	100	500	1000	2000	5000	10000
io	0.50%	3.70%	4.40%	6.60%	19.40%	32.60%
nio	0.30%	2.50%	2.70%	3.50%	8.70%	10.70%
nio2	0.70%	1.10%	2.10%	5.50%	12.80%	24.70%

FIGURE 5.4: Chat server CPU utilization

Heap



	100	500	1000	2000	5000	10000
io	33	65	104	157	350	792
nio	28	28	28	28	33	43
nio2	30	30	30	30	41	54

FIGURE 5.5: Chat server heap usage

Threads

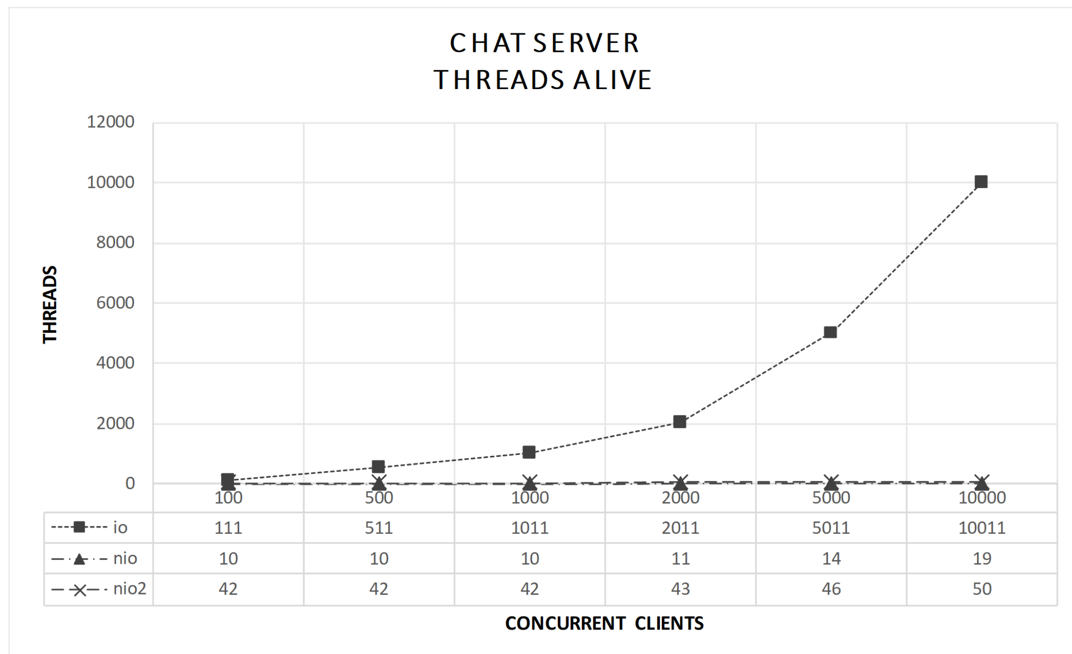


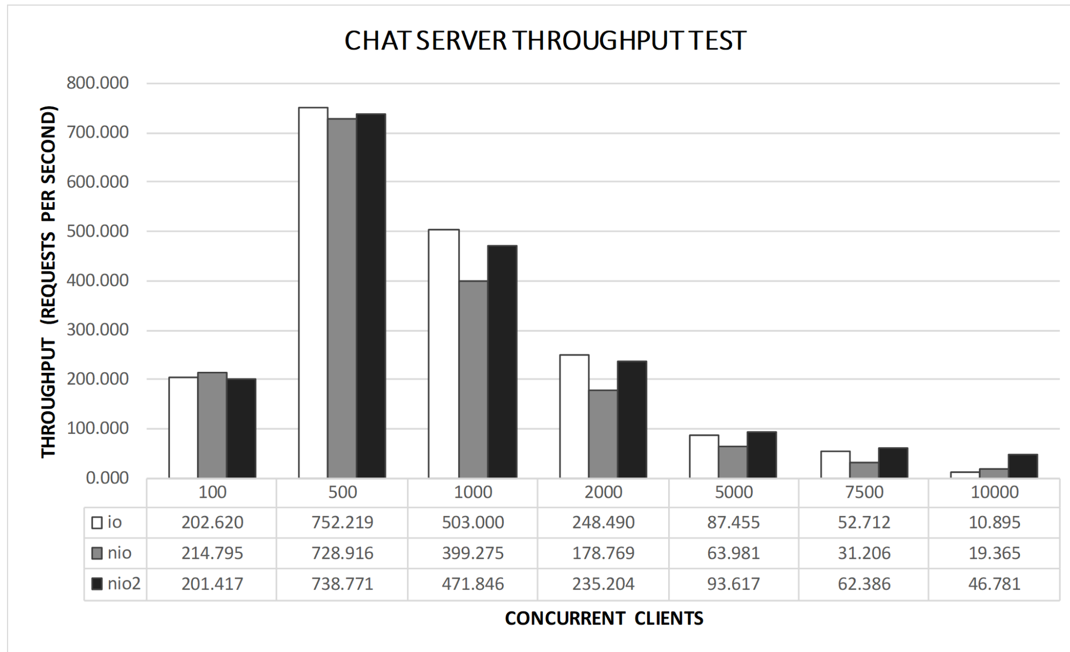
FIGURE 5.6: Chat server threads

5.4 High load test

The previous type of test applied a moderate amount of load to the server, not sufficient enough to fully utilize the CPU and make a conclusion about the system's throughput. The high load test was performed to determine the maximum quantity of requests each of the engines can handle within a unit of time. To achieve this, the client machines establish the set amount of connections to the server and start blasting messages. Unlike, in the previous case where the load generators were sending messages at a specific rate to simulate a particular scenario, this time the number of requests made from each connection depended on the speed at which server processes them. Each connection proceeds to send the next message only when it receives the *acknowledgement* response for the previous request. Throughput is calculated as the number of requests the server processed and sent an *acknowledgement* response back to the client per second.

5.4.1 Metrics

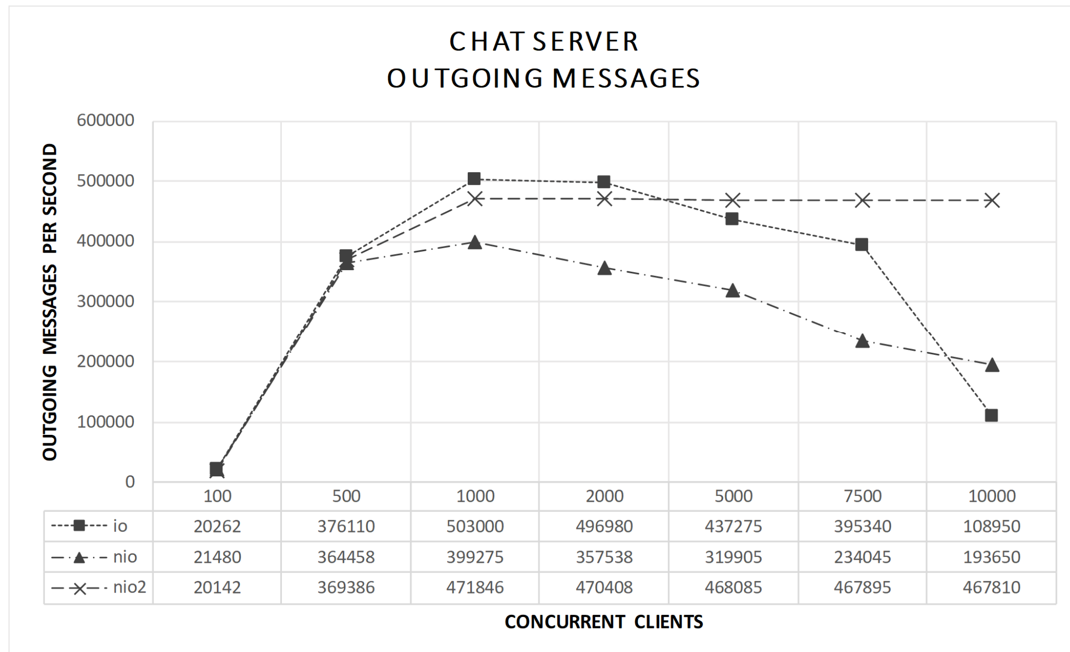
Throughput



	100	500	1000	2000	5000	7500	10000
io	202.620	752.219	503.000	248.490	87.455	52.712	10.895
nio	214.795	728.916	399.275	178.769	63.981	31.206	19.365
nio2	201.417	738.771	471.846	235.204	93.617	62.386	46.781

FIGURE 5.7: Chat server throughput

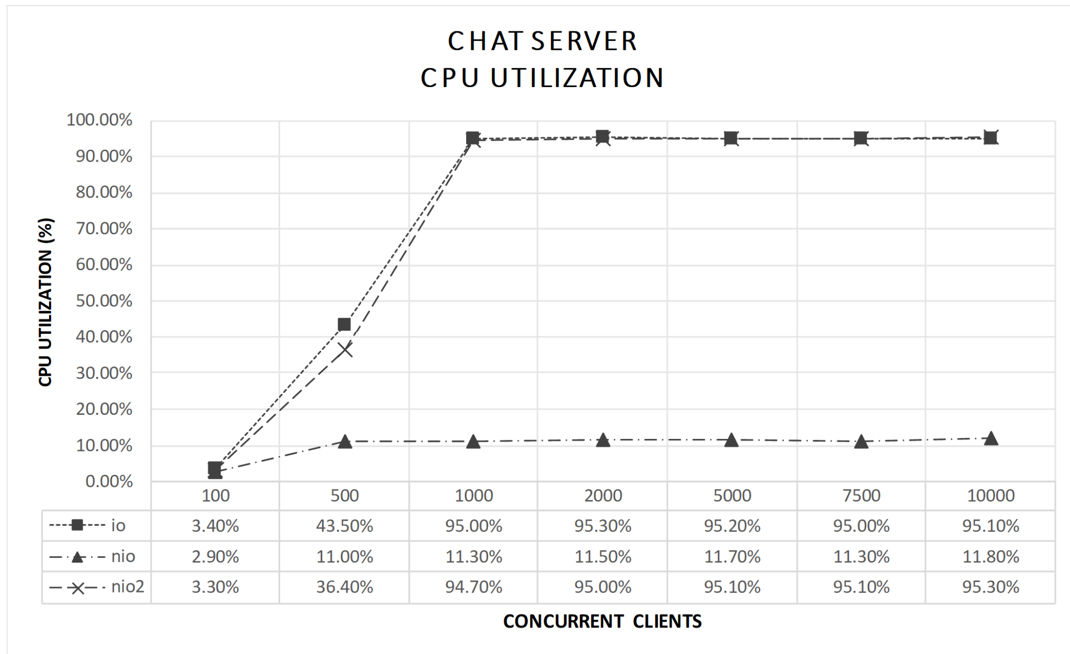
Outgoing messages



	100	500	1000	2000	5000	7500	10000
io	20262	376110	503000	496980	437275	395340	108950
nio	21480	364458	399275	357538	319905	234045	193650
nio2	20142	369386	471846	470408	468085	467895	467810

FIGURE 5.8: Chat server outgoing messages

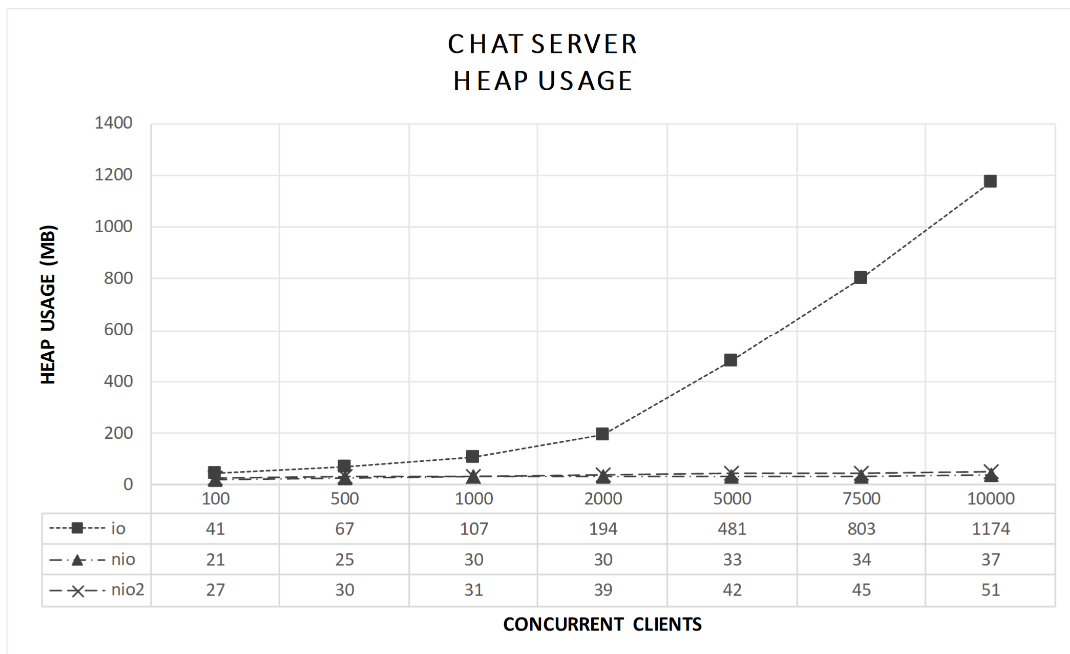
CPU utilization



	100	500	1000	2000	5000	7500	10000
io	3.40%	43.50%	95.00%	95.30%	95.20%	95.00%	95.10%
nio	2.90%	11.00%	11.30%	11.50%	11.70%	11.30%	11.80%
nio2	3.30%	36.40%	94.70%	95.00%	95.10%	95.10%	95.30%

FIGURE 5.9: Chat server CPU utilization at high load

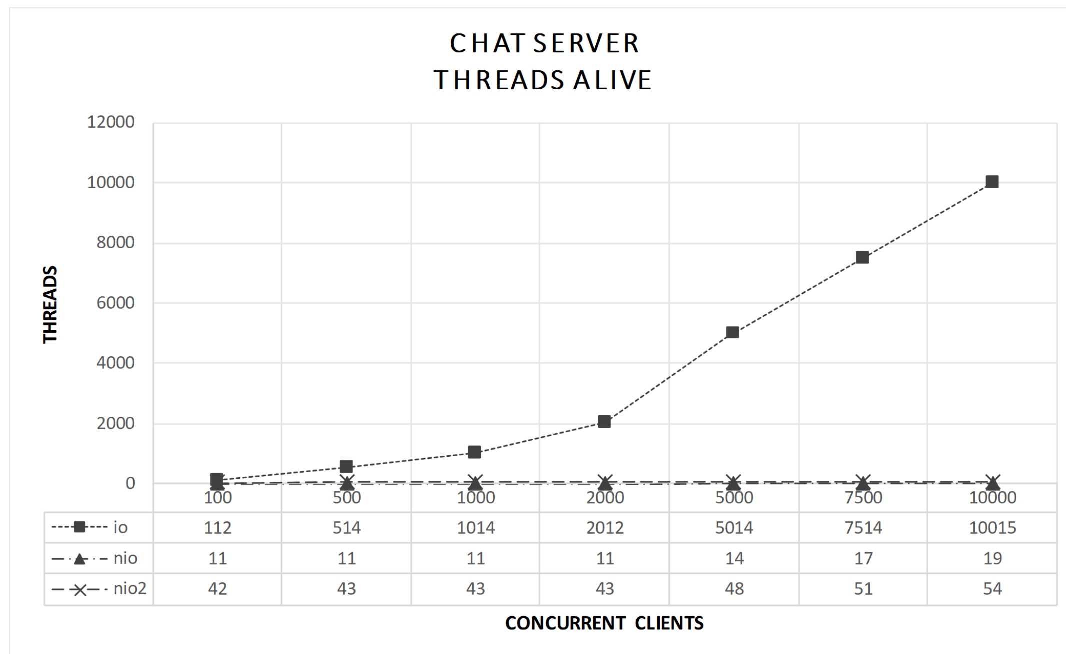
Heap



	100	500	1000	2000	5000	7500	10000
io	41	67	107	194	481	803	1174
nio	21	25	30	30	33	34	37
nio2	27	30	31	39	42	45	51

FIGURE 5.10: Chat server heap usage at high load

Threads



	100	500	1000	2000	5000	7500	10000
io	112	514	1014	2012	5014	7514	10015
nio	11	11	11	11	14	17	19
nio2	42	43	43	43	48	51	54

FIGURE 5.11: Chat server threads at high load

5.5 Analysis

Let's first break down the results of the real world simulation test.

It may not be noticeable from the visual representation of the latency metrics (fig. 5.2) at first sight unless the data table is examined, but a single-threaded *NIO* engine has a slight advantage (5-10 ms) in response times over the multi-threaded *NIO2* engine at the lower quantities of concurrent clients. This can be explained by the absence of the additional overheads in *NIO* that are present in *NIO2*. First of all, in the multi-threaded variation events are being dispatched to the handlers. Therefore no event in the selection is getting processed from the beginning to the end by the same thread, which in turn causes additional context switches. Besides that, an intermediary in this dispatching mechanism is an unbounded *LinkedBlockingQueue* to which the events are enqueued by the dispatcher and dequeued by the workers. Even though these operations are relatively cheap, they are performed repeatedly and still introduce additional overhead. Surprisingly, the single-threaded *NIO* engine performs very well, even up to 5000 concurrent clients.

The difference in latencies between the *IO* and *NIO2* engines up to 2000 clients is marginal, then *NIO2* starts gaining advantage due to the linear growth of the threads managed in the *IO* application.

At 10000 clients *NIO2* has a serious advantage over the counterparts, as both the *IO* and the *NIO* suffer from dramatic performance degradation. Such behavior of the *IO* engine is mainly due to the already mentioned drawback of the increasing amount of threads. *NIO*, in turn, fails to keep up with the increasing message rate not utilizing the CPU efficiently. One has to admit, that *NIO2* performance also declines significantly. One of the reasons for this is that each incoming message is broadcasted to every connected client forming a 1:N relationship. Sending out so

many messages is not a constant time operation, and while the amounts of connected clients grow twice in size, the response time has increased *1.69* times. Also, a small note is that the test performed increases the size of the step going from 5000 directly to 10000 clients, and the chart is not adjusted to proportionally display these distances. If an additional step at 7500 clients was added, the line visualizing the response times would seem to be less steep.

At a moderate load, neither of the engines have high CPU usage. However, the heap memory occupied by the *IO* is excessive in comparison to the NIO-based variations.

The high load test takes the engines to their limits. Similarly to the first test, the single-threaded engine has an edge in throughput at 100 concurrent clients (fig. 5.7), but this time due to an increased frequency of incoming messages the *IO* and *NIO* engines benefit from the multi-threading at earlier stages, and *NIO* starts lagging behind at 500+ concurrent users.

It may look like the peak performance in throughput is reached early. The answer to this lays in the outgoing messages (fig. 5.8) and the CPU utilization (fig. 5.9) charts. Due to the nature of the chat application, the outgoing traffic far exceeds the incoming one. Chart (fig. 5.8) visualizes the number of messages the server sends out each second. At 1000 clients that number already reaches 503 thousand messages every second for the *IO* engine and 471 thousand for the *NIO2*. This is also a point at which the CPU utilization reaches over 95%, and the upper threshold is met. It would have been useful to have a deeper granularity of a test between the 100 and 1000 concurrent clients. Since at 500 clients the CPU utilization was still below 50% for both *IO* and *NIO*, the highest throughput rate was likely to be reached at around 700 concurrent clients. Also, notice that the *NIO2* engine continues to send the same quantity of outgoing messages per second as the number of connections grows, while the *IO* starts to decline in that aspect.

A metric that puzzled me for a moment was a growing number of threads in both single-threaded and multi-threaded variations of the NIO engine since that was expected to remain static. As you can see from the figures 5.6 and 5.11 in both tests the number grew by 8-12 threads alive in both *NIO* and *NIO2*. Profiling the application has shown an interesting insight into the internals of the *Selector* implementation for the Windows OS. The thread dump contained the following information.

```
"Thread-17" #37 daemon prio=5 os_prio=0 tid=0x000000001c2b5000 nid=0x41b4
   java.lang.Thread.State: RUNNABLE
        at sun.nio.ch.WindowsSelectorImpl$SubSelector.poll0(Native Method)
        at sun.nio.ch.WindowsSelectorImpl$SubSelector.poll(Unknown Source)
        at sun.nio.ch.WindowsSelectorImpl$SubSelector.access$2600(Unknown
            Source)
        at sun.nio.ch.WindowsSelectorImpl$SelectThread.run(Unknown Source)

   Locked ownable synchronizers:
        - None
```

A more in-depth investigation of the source code revealed that the *Selector* implementation for Windows is multi-threaded and the number of helper threads is adjusted dynamically depending on the number of channels registered with the selector.

Summing up, both tests show that at the lowest amount of concurrent clients

single-threaded *NIO* engine slightly outperforms the counterparts, which is due to being lightweight and having the least overhead, but as the load grows it fails to keep up with it due to an inability to utilize the multi-core CPU. At moderate and high levels of load up to 5000 clients, both the *IO* and *NIO2* engines perform similarly in terms of throughput and response times, but *NIO2* better withstands the high numbers of concurrent connections. Unfortunately, the true potential of a multi-threaded non-blocking model was not fully revealed due to reaching the limits of CPU at early stages of the high load test.

Taking everything of the above into consideration, when choosing between the Java IO and NIO one has to be aware of the requirements of the system and the expected usage. The question of NIO vs. IO is not about which one is better. It's about trade-offs and finding the right tool for the job. When dealing with large numbers of connections that interact with the server infrequently multi-threaded approach with NIO would help to scale the system better, while thread-per-connection approach with IO would be wasting lots of resources and may hit a memory limit. However, when the connections are busy, it is likely that the CPU resources would be exhausted first, which was demonstrated by the high load test, and the primary concern shifts from scaling I/O to increasing the processing power.

Chapter 6

Conclusion

The main objective of this thesis was to analyze the Java IO and NIO from the perspective of building performant and scalable applications. This was achieved by conducting research in the area of thread-based and event-driven models, examining the implications of the blocking and non-blocking I/O on the overall performance of a server, and then applying the gained theoretical knowledge in practice and presenting the findings.

Thorough research on the threads vs. events debate in academia introduced a broader context of the problem. Next, this work examined the technical details and the key differences between the standard IO and NIO packages. The choice between the two has an impact on the design of the application and imposes constraints on how network communication is going to be handled. Lastly, before proceeding to the practical part, three types of concurrency strategies for dealing with the blocking and non-blocking I/O were proposed, which became the basis for the implementation of a chat server.

One of the most challenging and time-consuming tasks was running the experiments and gathering the metrics with many problems resolved along the way. The presented results show the behavior and characteristics of a thread-per-connection IO, single-threaded NIO, and multi-threaded NIO chat servers under moderate and high load with various numbers of concurrent clients.

The limitation of the experiments conducted is that they focus on a specific case of long-lived persistent connections. In a scenario, when a new connection is opened for each request, the scalability advantages of NIO over the IO would have been less evident. Nevertheless, one would still benefit from significantly lower memory consumption.

The future work may include a similar comparison in the case of short-lived connections. The classic example of a system with low overhead suitable for a benchmark is the JSON serialization or an echo server. Also, a question of interest would be measuring the performance in more sophisticated scenarios including database access and file I/O.

One of the unique contributions of this work is the comparative analysis of the IO and NIO-based systems with different concurrency strategies based on the metrics gathered. The experiments show that at lower numbers of concurrent clients, the difference between the IO and NIO in terms of throughput and response times is marginal. However, IO-based implementations suffer from a linear growth in memory usage, making NIO a more suitable option for handling large amounts of simultaneous connections.

Bibliography

Behren, Rob von et al. (2003a). "Capriccio: Scalable Threads for Internet Services". In: *SIGOPS Oper. Syst. Rev.* 37.5, pp. 268–281. ISSN: 0163-5980. DOI: 10.1145/1165389.945471. URL: http://doi.acm.org/10.1145/1165389.945471.

Behren, Rob von et al. (2003b). "Why Events Are a Bad Idea (for High-concurrency Servers)". In: *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*. HOTOS'03. Lihue, Hawaii: USENIX Association, pp. 4–4. URL: http://dl.acm.org/citation.cfm?id=1251054.1251058.

Elmeleegy, Khaled et al. (2004). "Lazy Asynchronous I/O for Event-driven Servers". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '04. Boston, MA: USENIX Association, pp. 21–21. URL: http://dl.acm.org/citation.cfm?id=1247415.1247436.

K. Park, V. S. Pai (2006). "Connection conditioning: Architecture-independent support for simple, robust servers". In: *Network Systems Design and Implementation*.

Kegel, Dan (2006). *The C10K problem*. URL: http://www.kegel.com/c10k.html.

Lauer, Hugh C. and Roger M. Needham (1979). "On the Duality of Operating System Structures". In: *SIGOPS Oper. Syst. Rev.* 13.2, pp. 3–19. ISSN: 0163-5980. DOI: 10.1145/850657.850658. URL: http://doi.acm.org/10.1145/850657.850658.

Lea, Douglas (2003). "Scalable IO in Java". In: URL: http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf.

Lee, Edward (2006). "The Problem with Threads". In: *Computer* 39, pp. 33 –42. DOI: 10.1109/MC.2006.180.

Li, Peng and Steve Zdancewic (2007). "Combining Events and Threads for Scalable Network Services Implementation and Evaluation of Monadic, Application-level Concurrency Primitives". In: *SIGPLAN Not.* 42.6, pp. 189–199. ISSN: 0362-1340. DOI: 10.1145/1273442.1250756. URL: http://doi.acm.org/10.1145/1273442.1250756.

Murphy, Julia and Max Roser (2019). "Internet". In: *Our World in Data*. URL: https://ourworldindata.org/internet.

Netcraft Web Server survey (2019). URL: https://news.netcraft.com/archives/2019/01/24/january-2019-web-server-survey.html.

Niemeyer, Patrick and Daniel Leuck (2013). *Learning Java, 4th Edition*. O'Reilly Media.

Ousterhout, John (1996). "Why threads are A bad idea (for most purposes)". Invited talk at USENIX.

Pariag, David et al. (2007). "Comparing the Performance of Web Server Architectures". In: *SIGOPS Oper. Syst. Rev.* 41.3, pp. 231–243. ISSN: 0163-5980. DOI: 10.1145/1272998.1273021. URL: http://doi.acm.org/10.1145/1272998.1273021.

Reactive Manifesto. Version 2.0. URL: https://www.reactivemanifesto.org.

Reed, Rick (2012). *Scaling to Millions of Simultaneous Connections*. URL: http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf.

Saxena, Raj (2018). *SpringBoot 2 performance — servlet stack vs WebFlux reactive stack*.

Schmidt, Douglas C. (1995). “Pattern Languages of Program Design”. In: ed. by James O. Coplien and Douglas C. Schmidt. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. Chap. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching, pp. 529–545. ISBN: 0-201-60734-4. URL: http://dl.acm.org/citation.cfm?id=218662.218705.

Tyma, Paul (2008). *Thousands of Threads and Blocking I/O. The old way to write Java Servers in New again*. Santa Clara, CA, USA.

Urban Airship: C500k in Action (2010). URL: http://urbanairship.com/blog/2010/08/24/c500k-in-action-at-urban-airship/.

Web Framework Benchmarks (2018). URL: https://www.techempower.com/benchmarks/.

Web server performance comparison (2019). https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison.

Welsh, Matt (2010). *A Retrospective on SEDA*. URL: http://matt-welsh.blogspot.com/2010/07/retrospective-on-seda.html.

Welsh, Matt, David Culler, and Eric Brewer (2001). “SEDA: An Architecture for Well-conditioned, Scalable Internet Services”. In: *SIGOPS Oper. Syst. Rev.* 35.5, pp. 230–243. ISSN: 0163-5980. DOI: 10.1145/502059.502057. URL: http://doi.acm.org/10.1145/502059.502057.

Zhu, Yuhao et al. (2015). “Microarchitectural implications of event-driven server-side web applications”. English (US). In: *Proceedings - 48th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2015*. Vol. 05-09-December-2015. United States: IEEE Computer Society, pp. 762–774. DOI: 10.1145/2830772.2830792.