MASTER THESIS'S THESIS 2024

Comparison of Concurrency Technologies in Java

Elias Gustafsson, Oliver Nederlund Persson

ISSN 1650-2884
LU-CS-EX: 2024-31

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

MASTER THESIS
Computer Science

LU-CS-EX: 2024-31

Comparison of Concurrency Technologies in Java

Elias Gustafsson, Oliver Nederlund Persson

Comparison of Concurrency Technologies in Java

(Structured Testing in a High Load Environment)

Elias Gustafsson
elias@gustafsson.at

Oliver Nederlund Persson
oliver.nederlund.persson@gmail.com

June 17, 2024

Master Thesis's thesis work carried out at Sinch AB.

Supervisors: Jonas Skeppstedt, jonas.skeppstedt@cs.lth.se
Samuel Alberius
Thomas Lundström

Examiner: Flavius Gruian, flavius.gruian@cs.lth.se

Abstract

This study was conducted in order to test the performance of Java's virtual threads against platform threads and reactive systems in the context of a high load server / client system. As virtual threads were only introduced in Java 21 (2023) they have yet to be thoroughly tested. We took great care to ensure that the constraints of the testing environment did not unduly influence the results. The experiments were designed to test performance in several ways, entailing methods with various mixes of simulated IO operations and computational tasks.

In regards to pure computational operations, neither virtual threads nor reactive streams outperformed regular platform threads. However, when it came to IO-heavy operations both of the high-concurrency technologies performed significantly better than platform threads. Reactive streams utilized significantly less memory and had fewer fluctuations, while virtual threads had much lower latencies. For instance, latencies for virtual threads where up to 44% shorter than for reactive in the 99th percentile for the IO tests, and had the best performance in three out of four benchmarks. Reactive had about half the memory usage of virtual threads in three out of four benchmarks, and had the best memory performance in all four benchmarks.

Because of their different strengths and performance that varies with the benchmark composition, we cannot conclude that either one of the technologies is generally better. Rather it is a matter of what metrics to prioritize. In our report, we provide all of the data from the benchmarks and explain the testing methodology in detail. In order to provide some guidelines, we can state that virtual threads and reactive performed better than platform threads in the benchmarks containing blocking operations, and that virtual threads were generally faster than reactive. However reactive was more stable and memory efficient than both virtual- and platform threads.

Keywords: Reactive, Virtual threads, Java, Load testing, Concurrency

Acknowledgements

Firstly, we would like to extend a special thank you to our supervisor at LTH, Jonas Skeppstedt, who in addition to clear and concise guidance on this thesis has held truly inspiring lectures during our time here. Secondly, we would like to thank Sinch and our supervisors Samuel Alberius and Thomas Lundström, who have given us a warm welcome and an interesting look inside an IT company. Thirdly, we thank Dora Beronić for responding to our questions about her previous studies in this field. We also extend thank yous to Alexander Svarvare for proof-reading our report and to Richard Lundberg et al for giving us an introduction to stack trace analyzis.

Contents

1 Introduction **9**
- 1.1 Research Questions . . . 10
- 1.2 Justification of our Study . . . 10
- 1.3 Previous Research . . . 11
  - 1.3.1 Structured Concurrency Constructs (2022) . . . 11
  - 1.3.2 On Analyzing Virtual Threads (2021) . . . 12
  - 1.3.3 Efficient Structured Concurrency through Lightweight Fibers (2020) . . . 13
  - 1.3.4 Integrating virtual threads in a Java framework (2023) . . . 13
- 1.4 Distribution of Work . . . 14

2 Theoretical Background **17**
- 2.1 Central Concepts . . . 17
  - 2.1.1 Virtual Threads . . . 17
  - 2.1.2 Reactive Systems . . . 18
- 2.2 Spring Framework . . . 20
  - 2.2.1 Spring Boot . . . 21
  - 2.2.2 Spring WebFlux . . . 21
  - 2.2.3 Spring web MVC framework . . . 21
  - 2.2.4 Summary spring . . . 21
- 2.3 Testing . . . 22
  - 2.3.1 On Performance Testing . . . 22
  - 2.3.2 Java Testing . . . 23
  - 2.3.3 Analysing the performance and costs of reactive programming libraries in Java . . . 24
  - 2.3.4 Examples of test implementation . . . 25
  - 2.3.5 Overhead Analysis . . . 25
  - 2.3.6 Hardware metrics . . . 26
- 2.4 Tools . . . 26
  - 2.4.1 Vegeta . . . 26
  - 2.4.2 VisualVM . . . 27
  - 2.4.3 Wireshark . . . 27

2.4.4 Flame Graphs . . . 27

3 Method **29**
3.1 Overview . . . 29
3.2 Test Setup . . . 30
3.2.1 Benchmark Design . . . 30
3.3 Testing within the JVM . . . 31
3.3.1 Hardware Setup . . . 33
3.3.2 Metrics Collected . . . 33
3.3.3 Handling Errors . . . 33
3.4 Parameters . . . 33
3.5 Load Ramping Experiment . . . 34
3.6 Constant Load Test and Additional Profiling . . . 35
3.6.1 Constant Load Test . . . 35
3.6.2 Stack Profiling . . . 36
3.7 Validation . . . 36
3.7.1 Network Connection . . . 37
3.7.2 Impact of Profiling Software . . . 37

4 Results **39**
4.1 Ramping Load Experiment . . . 39
4.1.1 CPU Intense . . . 40
4.1.2 I/O intense . . . 43
4.1.3 Matmul . . . 46
4.1.4 Mixed method . . . 48
4.1.5 Thread Creation . . . 51
4.1.6 Stability . . . 52
4.2 Constant Load Experiment . . . 53
4.2.1 CPU Measurements . . . 53
4.2.2 Memory Measurements . . . 54
4.2.3 Safepoint Measurements . . . 55
4.2.4 Sync Measurements . . . 56
4.2.5 Call Stack Analysis . . . 57
4.2.6 Network Connection Analysis . . . 58
4.3 Summary of the Results . . . 58

5 Conclusions **61**
5.1 Addressing Research Questions . . . 61
5.2 Contribution and Future Research . . . 62
5.3 Limitations and Considerations . . . 63
5.4 Conclusions in Summary . . . 63

Appendix A A **71**
A.1 Scripts . . . 71
A.1.1 Script for sending HTTP requests . . . 71

Appendix B Data **73**
B.1 Iteration tests . . . 73

Appendix C Benchmark Methods **75**
C.1 Imperative 75
C.1.1 IO 75
C.1.2 Compute 75
C.1.3 Matmul and Mixed 75
C.2 Reactive 76
C.2.1 IO 76
C.2.2 Compute 76
C.2.3 Matmul and Mixed 77

Chapter 1
Introduction

As the world and all its inhabitants and actors become ever more connected, efficient systems for communication and data transfer are more important than ever. In many applications, such as messaging servers, efficiency entails handling blocking I/O calls and a high level of concurrency in a good way. One well established option for scaleable and non-blocking code is reactive systems, which entails asynchronous code based on data streams with publishers and subscribers. For instance, several backends at Sinch AB where this study was conducted are implemented as reactive systems. Although efficient, reactive systems are considered difficult, and therefore expensive to program and maintain as developers are often uncomfortable with abandoning imperative programming (Brian Olson 2019).

Virtual threads, a new feature in Java 21 (Ben Weidig 2023), are intended to solve the same problems as the various reactive frameworks without the need for difficult syntax and new coding paradigms. As they are an implementation of the Java.lang.Thread interface, they can be used interchangeably with regular platform threads (traditional Java threads, implemented as a thin wrappers on a kernel threads) although it is of course not always a good option as we will explain in the background section. The virtual threads are decoupled from the kernel threads of the operating system, instead being handled by the Java virtual machine which mounts and unmounts virtual threads from platform threads. In this context, mounting a virtual thread means moving its states and local variables to a platform thread, and unmounting means moving the states and local variables of the virtual thread to the heap so that a different virtual thread can be mounted to the platform thread. When a blocking method is called from a virtual thread, it is unmounted from its kernel thread and saved on the heap for the duration of the waiting period, all without interference from the programmer.

This study aims to compare the performance of reactive systems, virtual threads and platform threads, specifically for a system with a high and dynamic load which is the primary use case for both of these methodologies. As Java 21 was only released in 2023 the body of research on its virtual threads is still very thin, which is the raison d'etre for this study. There is some limited research on virtual threads which serves as part of the theoretical foundation of this study, and it is presented under section 1.3. In order to make the tests as fair as possible, the reactive and virtual thread-based servers both utilize the Spring Boot

framework, are called over HTTP (which is how Sinch AB tests its servers) and implement the same functions. They are run in a controlled environment, and system statistics are recorded with VisualVM (Oracle Corporation 2024*b*). There is a number of parameters that are of interest for measuring and comparing the performance of these technologies. In this study, we look thoroughly at latency, throughput, memory usage and CPU utilization. These metrics are looked at in much of the previous research cited in this report. We also consider them intuitively relevant, as latency and throughput are important for the user experience and memory usage and CPU utilization are important for dimensioning server hardware. Special care is also taken to ensure that testing software and the server-client connection have minimal impact on the measurements.

1.1 Research Questions

Small gains in efficiency for software handling high loads (e.g. servers) can lead to an increased customer satisfaction and loyalty (Kim et al. 2007), and we also believe that more efficient software is generally desirable. To optimize high load software systems, Java 21 introduced virtual threads in 2023. This is a promising project that is intended for "high-throughput concurrent applications, especially those that consist of a great number of concurrent tasks that spend much of their time waiting" (Oracle Corporation 2023). If virtual threads are deemed to have similar or better performance than the previous state-of-the-art approach of asynchronous programming, it would be very interesting. Even if they perform similarly to other technologies, they may also offer a lot of other advantages such as software that is easy to read, write and debug.

As of now there are not many studies comparing virtual threads to other concurrency approaches in Java due to their recent introduction. Furthermore, we believe that many of these comparative studies are performed by corporations and are not published. There are studies that compare beta versions of virtual threads in Java with platform threads (Beronić et al. 2022) (Beronić et al. 2021) (Pufek et al. 2020) and one study that compares virtual threads with a reactive approach in Quarkus using Java 21 (Navarro et al. 2023). However, we have found these studies to have a somewhat limited scope. For example, some limit the number of maximum platform threads, and most do not test a wide array of methods (e.g. mixes of computational and IO tests that may disadvantage some of the technologies unduly). Another aspect we could add to this research topic are tests performed on a wide variety of loads. In short, **the aim of this study is to compare virtual threads, platform threads and reactive systems for high-concurrency applications**

The research questions of this thesis are:

- **RQ1:** What are the differences between these different concurrency techniques within the JVM on high load systems?
- **RQ2:** Are virtual threads a viable alternative to replace platform threads and asynchronous data streams on high load applications?

1.2 Justification of our Study

As virtual threads are a very recent addition to the JVM, the body of research on them is relatively slim. It is even slimmer on comparisons between virtual threads and asynchronous

programming, which we consider particularly relevant as asynchronous programming has very similar use cases to virtual threads. We also feel that we can contribute a lot on performance testing methodology for virtual thread performance testing, seeing as the studies we cited have generally not publicized the details of their test setups and have not performed a particularly large and varied set of tests.

We have gathered a large amount of data by using scripts that run our test cases many times with different parameters. A large amount of data is absolutely necessary for our tests to be conclusive and our research to be relevant. This comprehensive set of data is an important contribution to this field.

1.3 Previous Research

The body of previous research on comparisons between reactive systems and virtual thread-based system is still relatively thin, however we have benefited greatly from research that has studied each of these technologies separately. Especially methodologies of studying and measuring the performance of high-concurrency systems have been very applicable to this study.

1.3.1 Structured Concurrency Constructs (2022)

In the study "Comparison of Structured Concurrency Constructs in Java and Kotlin - Virtual Threads and Coroutines" (Beronić et al. 2022), the authors compare concurrency constructs in languages implemented in the JVM, including Java. The testing was done on a machine running Ubuntu with 16 GBs of memory and an Intel i7 (3,4 GHz) processor using OpenJDK's JDK-19, an early access version with support for virtual threads. The test case was a HTTP server waiting for incoming standard requests. The study tested four different concepts, two of which were implemented in Java including virtual threads and platform threads. The primary measurements used were heap memory, latency and number of kernel threads started. The tools used for testing were Vegeta (used for application testing that can perform attacks on a HTTP server as described in the background section) and VisualVM. The testing procedure was to use Vegeta to do attacks with two different rates (2000 and 8000 requests per second) for 10 seconds. They also perform pre-runs to warm up the testing rig. The latencies were calculated by Vegeta and VisualVM was used to collect the remaining measurements.

The first result they presented was the total number of threads started. For 2000 attacks per second the average amount of started kernel threads were 101 017 for regular Java platform threads and 36 for virtual Java threads. For an attack rate of 8000 the average amount of started threads was 388 749 for regular Java threads and 268 for virtual Java threads.

The second result presented was heap usage. With an attack rate of 2000 requests per second, regular java threads used 128.28 MB while virtual threads used 16.22 MB. With a rate of 8000 requests per second regular java threads used 386.08 MB while virtual threads used 64.66 MB.

The third and last test concerned latency measured in milliseconds. Here the researchers used mean value, 50th percentile and 95th percentiles. For 8000 request per second the latency results for platform threads were 0.99, 0,16 and 0.83 respectively. For virtual threads they were 0,16, 0,10 and 0,13. This indicates that there is a notable difference between the percentiles.

The authors concluded that virtual threads are promising but more extensive testing need to be done on the official version of virtual threads. Some elements from this study that we decided to incorporate were the inclusion of memory usage and latency metrics, as well as the Vegeta testing tool. However, we collect a larger set of metrics over a larger spectrum of loads as well as perform several verifications of our testing environment.

1.3.2 On Analyzing Virtual Threads (2021)

The study "On Analyzing Virtual Threads - a Structured Concurrency Model for Scaleable Applications on the JVM" (Beronić et al. 2021) aims to explore structured concurrency in Java with a focus on virtual threads. The authors argue that blocking operations such as I/O cause problems regarding efficiency due their 'single access policy'. These threads rely on kernel threads which require a lot of resources, leading to an inefficient use of hardware. They tested how virtual threads compared to platform threads.

The testing environment was hosted on a machine running Ubuntu with an 8-core processor and 13 GBs of memory. They used an early version of JDK loom-17ea.2 (project loom) that supports virtual threads. To compare the results they used creation speed, efficiency and concurrency level. The tests consisted of a merge sort algorithm, as well as generic parallel and sequential run algorithms.

To measure the creation speed, they used latency measured in milliseconds. The test simply created multiple threads with different type of scales. The four different scales was 0.1, 0.5, 1 and 1.5 million. The result was that the creation of regular threads was 1.7 , 2.2 and 2.7 times slower then virtual threads for 0.1, 1 and 1.5 million created threads. Furthermore, the creation of threads for both regular and virtual threads seemed to scale linearly.

Regular threads were also tested against virtual threads in terms of performance. They tested this using a merge sort algorithm, sorting arrays with 1000 through 512 000 elements. They measured how long it would take regular threads and virtual threads to execute the algorithm, with both approaches having access to 16 kernel threads. The result was that for smaller scale (up to 60 thousand elements) virtual threads outperformed platform threads with a factor of 80. The difference became smaller when the array became bigger but on average virtual threads had 30% better performance.

To test the concurrency level, they measured the latency for performing an unspecified task, where they gradually increased the number of threads symbolizing requests. The number of threads started at a hundred thousand and was gradually increased to ten million due to an increasing number of requests. For both threading types, the latencies increased with the increased number of requests. With 2500 requests virtual threads outperformed platform threads by 38%, but as the requests increased to ten thousand, they outperformed platform threads with 591%. From this the authors concluded that virtual threads should be able to handle more load given the same amount of CPU resources.

Some elements of this study that influenced our own were the merge sort benchmark and concurrency data. The merge sort benchmark served as the inspiration of a matrix multiplication benchmark we used, and the superior performance of virtual threads as a reason to investigate them further.

1.3.3 Efficient Structured Concurrency through Lightweight Fibers (2020)

The study "Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine" (Pufek et al. 2020) explores Java fiber threads (an early working name for virtual threads). The authors argue that systems that handle incoming requests tend to use kernel threads poorly, wasting a lot of resources. The study investigates structured concurrency techniques trying to find more efficient methods to handle high load systems.

The tests were conducted on a virtual machine running Ubuntu with 9GBs of memory. For the fiber threads they used an early access version of OpenJdk. Like the previous study, they used Vegeta to simulate load. The server side of the test setup was an HTTP server implementation provided in the JDK. The number of threads were limited to 64.

In the test, each thread was assigned a task where they added a delay (Thread.sleep) to artificially simulate load. After that, latency was measured for different delay lengths. While under heavy load, the result was that the latency for virtual threads were 16% of that for the platform threads. The authors concluded the report with stating that virtual threads will have a prominent role in the future, but that further investigation is needed due to their experimental status.

The study's simulation of I/O operations as simple delays as well as varying delay duration is something that we decided to also include in our experiments. Their arguments for how a high level of concurrency might require more specialized techniques also helped serve as a justification of our study.

1.3.4 Integrating virtual threads in a Java framework (2023)

A recent study on the matter of virtual threads in Java was recently published (Navarro et al. 2023). The study aims to investigate reactive and virtual threads in the JVM and integrate it to the Quarkus framework, thus offering an approach for integrating virtual threads into Java reactive frameworks. Quarkus was used due to it being optimized for cloud environments by having low resource usage compared to for example Spring Boot, due to it's native compilation capability and optimized runtime. The authors also defended their choose of Quarkus over the Spring framework since the Spring implementation of virtual thread servers do not allow for virtual threads to easily run in the same application as reactive systems (implemented in Spring Webflux). Quarkus allow for all three technologies to be used in the same application.

The experimental part of the study was conducted on a machine with a twelve core i7-10850H (2.7 GHz) chip and 32 GBs of memory running Ubuntu. They split the application and had the servers and clients run in Docker containers with constraints on CPU and memory. The study implemented load testing for the three different technologies where latency, throughput, CPU usage and heap usage where measured for varying request rates.

The server functions used in the study were inspired by the Fortune test from the Techempower benchmark suite, where an object is fetched from a database and added to a list, after which the list is sorted and a JSON object is returned. In this study, an additional delay was added in order to make the heap grow. The study's authors also outlined different working conditions for the different server functions that they called: normal (throughput equal to request rate, CPU usage less then 80%), critical (throughput equal to rate but latency increase and

memory usage increase) and overload (throughput less than rate and high resource utilization).

The result was that for no added delay the reactive approach had the lowest latency, highest throughout and lowest resource utilization. The platform threads also outperformed the virtual threads. With a constant 200 ms delay the reactive approach still had the best performance. The comparison between blocking and virtual threads, however, was more difficult. Virtual threads had a higher overall CPU and memory usage, and they had higher throughput than blocking threads until it reached a critical level and crashed. The conclusions they drew from this were:

- **i)** That virtual threads in Quarkus reached a plateau in regard to memory usage quickly.
- **ii)** For shorter blocking operations virtual threads are more computationally expensive than platform threads.
- **iii)** Reactive threads have the best resource utilization of these technologies.

After performing comparisons between the different technologies, the authors used a fixed request rate and compared virtual threads and the reactive approach. They speculated that the garbage collector was the major reason for the difference between the technologies. In this experiment they collected metrics from the garbage collector including count, average pause, collection pause and sum of pauses. They concluded that virtual threads had a much higher garbage collection usage which was the root cause of it performing worse then the Quarkus reactive approach.

One thing from this study that is readily applicable to ours is procedures to prevent introducing noise to the test data causing an unfair comparison between the technologies. The authors used warm-up runs before every test, to make sure that the appropriate classes had been loaded and that the just in time (JIT) compiler had optimized hot methods. Furthermore, they ran each test three times. Even though they took these precautions the tests were still somewhat unstable and unpredictable. They attributed this instability to 'chaos' introduced by the garbage collector and JIT compiler. Due to this they opted to choose the best of the three tests each test run instead of aggregating an average of the test runs. The use of a database might also be a pitfall according to the authors. The relatively complicated benchmarks used by the authors may have introduced additional noise to the results which can be seen in their presented result, where the performance of the servers sometimes have unexpected dips. Furthermore, the benchmark test used in this study is somewhat unfair since it forces the server to use a lot of memory which is not the intended use case for virtual threads (clearly stated in project loom, the project name for Java's implementation of virtual threads).

1.4 Distribution of Work

Distribution of work between the authors has been practically equal. Every step of both the experimental methodology and the writing of the report has been subject to fervent discussion and thorough revisions. Good results are generally the result of engaged discussions and different perspectives.

There are however some areas where one person contributed more than the other. For instance, Oliver Nederlund Persson contributed more to the implementation of the client and automated test cases while Elias Gustafsson contributed more to the implementation of

the server and tests on the network connection. Oliver is also more responsible for finding and compiling previous research while Elias looked more closely at the JVM.

Chapter 2
Background

In this chapter we present some theoretical background on the central concepts of this study. We also summarize a few select studies on similar topics.

2.1 Central Concepts

Here we introduce the important concepts of this study, including everything the reader needs to know about them to capitalize on our results and conclusions. We introduce virtual threads and reactive, which the reader might not be familiar with, and also the Spring framework in which the servers were implemented.

2.1.1 Virtual Threads

The concept of virtual threads has been around for a long time. (Vahalia 1996, p 53 - 55) describe a *lightweight process* (LWP) as a process that is more decoupled from the kernel threads than what for instance a platform thread is. LWPs do not block kernel threads when the LWP is blocked due to I/O. The limitation of LWPs is that they require expensive system calls for synchronization, creation and destruction. However, multiple *user Threads* (Vahalia 1996, p 55 - 58) can later be mounted on LWPs or regular processes. *User Threads* being a high level thread abstraction on the user level managed by libraries without the kernel knowing about the threads. For User threads the Library manage schedules and context switches furthermore it also save the *User Threads* individual stack. However, the kernel still manages scheduling between the processes or the LWPs. If the *User Thread* is mounted on a LWP, a user level block (e.g. I/O) will not block the kernel thread (only the LWP). *User Threads* that not block the underlying process, Virtual threads, have been implemented in the *Go* language as Goroutines and now in Java 21 as virtual threads.

Virtual threads in java were proposed in JDK Enhancement proposal 425 (JEP) and were implemented as a preview feature in java 19 and later java 20. They were finalized in Java 21. Virtual threads are not intended to replace existing threads nor to replace existing asyn-

chronous styles. Instead, they are intended as a tool for implementing regular 'thread-per-task' programming while operating in an unblocking manner.

In JEP 444, the authors brought up how server applications tend to handle multiple users that are independent of each other. Dedicating one thread per user is a simple way to handle concurrency, but this approach is not scaleable with regular platform threads. That is because these traditional java threads are implemented by the JVM as wrappers around kernel threads. Therefore, the underlying system (hardware and OS) dictates how many parallel requests that can be handled at once which generally puts a low ceiling on concurrency as they are stuck waiting for blocking operations. An asynchronous style of programming can be used to handle this. However, as stated in JEP 444, this style of programming introduces some problems and makes debugging difficult.

The virtual threads in Java are instances of java.lang.thread just like platform threads. A platform thread runs on a kernel thread and captures it for the entire duration of the code, while virtual threads do not. As virtual threads are implementations of the java.lang.thread interface, they could be considered a user-friendly way of enabling high concurrency as most Java developers would probably be familiar with the interface. All of the context switching in regards to virtual- and platform threads are handled automatically, and the programmer is not intended to handle virtual threads differently than platform threads. Just like virtual memory simulates abundant resources by mapping a large number of addresses to a smaller memory, a large number of virtual threads are mapped to a smaller number of kernel threads.

Java's platform threads rely on the OS scheduler while virtual threads are managed by the JVM assigning the virtual threads to platform threads. Virtual threads are not restricted to one specific platform thread during their lifetime, and can be assigned to different platform threads by the scheduler after each context switch. A virtual thread gets unmounted from its platform threads when it is blocked (for example I/O operations), which ensures that the underlying platform threads are not being blocked. However, the JVM does not always unmount virtual threads. Two situations where the virtual threads are not unmounted are certain blocking situations and pinning. The first situation is due to JVM and OS limitations were the JVM compensates by temporarily increasing parallelism, i.e. creating more platform threads. The second scenario, called pinning, happens when blocking operations are performed inside of a synchronized block. In both these situations the underlying platform thread is also being blocked (Oracle Corporation 2023).

The stack frames of virtual threads are stored in the garbage-collected heap in between context switches. The heap grows and shrinks dynamically. This allows the program to run many virtual threads. There is also support for thread-local variables, but it is recommended to avoid this since the number of virtual threads can grow a lot which may cause memory related errors as heap usage increases. According to the documentation of Java 21, a single JVM can potentially handle millions of virtual threads (Oracle Corporation 2024*a*) and the frameworks used in this study do not impose limits on the number of virtual threads created.

2.1.2 Reactive Systems

There are a number of implementations of reactive systems, but a common denominator is that they rely on asynchronous data streams. When describing reactive systems, the Reactive Manifesto (Jonas Bonér 2014) is usually cited. It is written and signed by a number of IT professionals and says reactive systems should be flexible, modular and easily scale-able. The manifesto enumerates four distinct properties that reactive systems should have:

- **Message driven:** Asynchronous message passing enables compartmentalization and monitoring the message streams makes it easy to apply e.g. load management and back pressure. Components can be publishers or subscribers in these systems, and subscribers react on data from the publishers.

- **Responsive:** The responsiveness of a system is the key to make it user friendly and thus use-able, so a reactive system should be focused on responsiveness. As an added benefit, this also simplifies error handling.

- **Elastic:** A reactive system should be elastic, meaning responsiveness and efficiency should not degrade with a varying workload. Various methods are employed to achieve this, for example backpressure.

- **Resilient:** A reactive system should stay responsive even in the case of failures. Components should be compartmentalized and there should be a well defined way of handling recovery without delegating error handling to the clients of the failed components.

Asynchronous, event driven programming is often considered to be a difficult process. To implement and maintain, and the resulting code is often difficult to debug and understand (Madsen et al. 2017), (Kambona et al. 2013). Non-imperative code is harder to debug not only for the obvious reasons such as difficulty inserting break points, but also because most automated debugging features have trouble handling asynchronous code. The phenomenon of asynchronous programming has been described as 'callback hell' (Edwards 2009), (Belson et al. 2019), (Brodu et al. 2015). The resulting code has been compared to 'asynchronous spaghetti' and modern 'goto' (Kambona et al. 2013).

In listing 2.2 and 2.1, the implementation differences between reactive services and synchronous services (same code for virtual threads and platform threads) are illustrated. In these examples the services perform a blocking call fetching an array from a database, convert the values to integers, sort them and remove all entries with a certain ID. For the reactive service to work asynchronously the implementation must be in the form of callbacks usually resulting in long lines of code, like functional programming. Meanwhile the implementation of virtual and platform threads follows a more imperative approach that can make it significantly easier to implement and read.

```
1  import reactor.core.publisher.Flux;
2  import reactor.core.publisher.Mono;
3  @Service
4  public class ReactiveService {
5      @Autowired
6      private ReactiveDataRepository reactiveDataRepository;
7      public Flux<Data> fetchDataConvertSortAndFilter() {
8          Flux<Data> dataFlux = reactiveDataRepository.findAll();
9          return dataFlux.flatMap(data -> {
10                             try {
11                                 int value = Integer.parseInt(data.
       getValue());
12                                 data.setValue(String.valueOf(value)
       );
13                                 return Mono.just(data);
14                             } catch (NumberFormatException e) {
15                                 return Mono.empty();
16                             }
17                         })
```

```
                        .filter(data -> data.getId() != 31)
                        .collectSortedList(Comparator.comparingInt(
    data -> Integer.parseInt(data.getValue())))
                        .flatMapMany(Flux::fromIterable);
    }
}
```

Listing 2.1: Reactive service

```
@Service
public class RegularThreadService {
    @Autowired
    private DataRepository dataRepository;
    public List<Data> fetchDataConvertSortAndFilter() {
        List<Data> dataList = dataRepository.findAll();
        for (Data data : dataList) {
            try {
                int value = Integer.parseInt(data.getValue());
                data.setValue(String.valueOf(value));
            } catch (NumberFormatException e) {
                return null;
            }
        }
        Collections.sort(dataList, Comparator.comparingInt(data ->
    Integer.parseInt(data.getValue())));
        dataList.removeIf(data -> data.getId() == 31);
        return dataList;
    }
}
```

Listing 2.2: Threads

2.2 Spring Framework

The spring framework (Spring 2024) is used to build enterprise-grade Java applications. Spring provides functionalities and features that facilitate development of robust, scalable and maintainable applications. The framework handles dependency injection that promotes a loose coupling for components of the application by externalizing their dependencies. Spring also offers functionalities such as data access (integration with databases), security, testing, the model view controller (MVC), WebFlux and Spring Boot. MVC is a framework that helps building web applications and provides components for managing session state and HTTP requests. Spring Boot is a framework that extends the spring framework by providing auto-configuration and some default functions which helps in building production-ready applications. WebFlux is a reactive programming framework allowing for non-blocking and asynchronous concurrency.

All in all the Spring framework allows for developing solid production-ready applications. It is used in the industry, including the host company, thus it will be used to implement the servers in this project. The reason for this is that the framework offers a good base for comparison between the different concurrency approaches, reducing potential differences between them compared to if said approaches would have been entirely created by the authors.

2.2.1 Spring Boot

Spring boot (Spring 2024) is used to simplify the process of configuring, deploying and building applications. It includes applications that can handle HTTP request. There are three main features to this: autoconfiguration, the ability to create standalone applications and an opinionated approach to configuration. Autoconfiguration reduces some of the need for the developer to perform certain configurations regarding dependencies, although it is possible to override these autoconfigurations. Opinionated approach means that Spring Boot adds starter dependencies and configurations based on the needs of the project. Standalone applications means applications that run without relying on an external web server.

2.2.2 Spring WebFlux

Spring WebFlux is a reactive framework that is non-blocking and asynchronous (Spring 2024). WebFlux is based on project Reactor and implements reactive streams. The goal of reactive streams is to handle 'live' data with an unknown volume. This is done through controlling the exchange of data across asynchronous boundaries (data between threads and thread pools) without having the receiver buffering an unknown amount of data. An example is backpressure, where the consumer of the reactive steam (data) controls the rate at which it receives data, thus preventing potential overload (Reactive Streams 2024).

Spring WebFlux provides a reactive web client that handle HTTP requests in a purely asynchronous manner without blocking I/O. Its client-side applications allows for outbound requests to external services or APIs without blocking through leveraging reactive streams and handling backpressure efficiently. It offers a multitude of HTTP operations (for example POST and GET) and support error handling and integration with other Spring applications.

2.2.3 Spring web MVC framework

This framework is designed to build web applications following the MVC pattern (Spring 2024). This includes three components: model, view and controller. In the context of this project the controller is the primary component. When a request has been received, the Spring dispatcher servlet intercepts the request and forwards it to the right controller. The controller handles incoming HTTP requests from clients. Within the controller there is a request mapping that map a certain request to a certain handler function for example POST '/path'.

This framework offers a simple method to create servers that can handle incoming HTTP request in an effective way. It is compatible with Java 21 and offer support for both regular platform threads and virtual threads through a simple configuration process.

2.2.4 Summary spring

To summarize, the Spring framework offers a simple way to create production-ready applications including web clients. The framework will be used in this project due to these main reasons:

- **i)** It is a commercial framework, so by using it instead of implementing something ourselves we reduce the risk of the implementations not being equally good for all technologies tested.

- **ii)** It is commonly used in enterprises including the host company.

2.3 Testing

Good methodologies for testing performance are absolutely central to this study. As we are interested in comparing virtual threads with reactive systems, care has to be taken so that other parts of our setup do not influence the results unduly. In the first subsection ("On Performance Testing"), we present some terminology and ideas that are useful to know for readers of our study.

2.3.1 On Performance Testing

To compare different approaches there must be a system of how to compare them. For performance there is a multitude of possible test data that measure different aspects of the application. Furthermore, the application has multiple areas of operation and thus measuring with different loads and load types is important. Steven Haines write about this in the book “Quantifying Performance” (Haines 2006) in the context of a Java enterprise. Haines outline three primary measurement categories which are response time, throughput and resource utilization. These three categories share a relation under increasing loads (measured by Haines as number of concurrent users). When the load increases the resource utilization increases alongside the throughput. Eventually the available resources become saturated due to either ineffective utilization or due to limited hardware. When the resources are saturated, the throughput will stagnate, and it may decline due to the system having to spend more resources managing itself (for example by conducting context switches). Also, the response time increases due to increased strain on the system.

Haines mentions both performance and scalability. Performance means an application’s ability to handle a high load and and scalability entails how the system handles an increase in load. Although performance and scalability are usually seen as similar, they are not. The performance of a system is measured for a certain load while the scalability measures the ability for a request to uphold the same performance during increased loads. A way to test scalability is to test the system while gradually increasing the load while measuring important metrics. When the load get too high the system will experience a decrease in throughput and increase in response time which marks the point in which the system has reached its maximum load potential.

Load testing is the act of testing a system under heavy loads such as having many concurrent users. The purpose is to detect load related problems (Jiang & Hassan 2015). It can also be extended to find load related problems regarding performance such as response time and throughput (Jiang et al. 2009). Recommendations for effective load testing is to perform it during a longer duration of time (Jiang et al. 2009) and to not only focus on the average result but also include factors such as 95th percentile (Jiang & Hassan 2015). The reasoning for longer test runs is due to small fluctuations in memory, CPU usage etc. looking at the more extreme cases such as the 95th percentile also provide certain insights as one can detect unacceptable deviations in performance.

2.3.2 Java Testing

To answer the research questions and fulfil the aim of the thesis, custom benchmarking was utilized. A benchmark can be viewed as a test to evaluate the performance of a tool or technique (Sim et al. 2003).

The aim of the benchmarks is to test the applications in different areas of operation, such as CPU and I/O intense operations. For benchmarking to be a fair, comparison certain requirements need to be fulfilled on the individual benchmark tests. According to (Bull et al. 1999), a few properties of a successful benchmark is:

- **Representative:** If the benchmark intends to test I/O these operations ought to be included.
- **Interpretable:** The result should give insights to why that particular result was achieved.
- **Robustness:** The test itself should not be an uncertainty when repeating tests.
- **Portability:** Be able to recreate it.
- **Standardized:** Performance metrics should mean the same.
- **Transparent:** Clear what is being tested.

In short, the benchmarks should test a specific area where the metrics and measurements should have the same meaning between the technologies. The test should also be interpretable to explain potential differences.

Doing performance testing on runtime environments can be very difficult and the performance of Java applications can sometimes seem unpredictable due to various reasons such as the garbage collector, JRE and JIT. The performance of a Java application can depend on the JVM it is running on or how long it runs for (Eeckhout et al. 2003). The complex and dynamic runtime of Java applications may require more well thought out tests while benchmarking compared to compiled languages such as C and C++ that have a predictable runtime environment (Blackburn et al. 2006). Performance of Java applications can be difficult to test with benchmarking due to high variance within the virtual machine, so therefore quantitative methods are recommended for ensuring a good benchmarking test (Gu et al. 2006). Prevalent methodologies for testing the performance of Java applications have been criticized for leading to incorrect and misleading results (Lion et al. 2016). Some reasons are that Java applications can give different results between runs due to different sources of non-determinism such as just-in-time optimization in the virtual machine, garbage collection and thread scheduling. There are multiple way to present the result from Java benchmarking tests such as:

- **i)** Taking the average.
- **ii)** Taking the median.
- **iii)** Only including the best run.
- **iv)** Only including the worst run.

Ways of mitigating the seemingly unpredictable effects in the JVM include forced garbage collections between different test iterations to avoid having the garbage collector starting at different times between the test runs. One can also perform back-to-back measurements ('cccddd' instead of 'cdcdcd', c and d being specific tests) on the same VM instance. Warm-up runs and being careful not to include initial class loading can also give better data (Lion et al. 2016). We undertook all of these measures in our study.

Previous studies that used similar methods as our project (load testing with Vegeta) did not specify eventual server crashes or a high variance between test iterations (Beronić et al. 2022), (Beronić et al. 2021), (Pufek et al. 2020). A member of these three studies was asked by us (Beronić 2024) regarding her test results. The consulted researcher stated that she did in fact experience server crashes, but it was due to 'Fibers', an early version of virtual threads. However, they did state that server crashes would happen if they would have tested with a higher load. Furthermore, the researchers stated that they did not experience notable differences between test iterations, however they did notice seemingly random spikes in latency after further analysis which they attributed to the garbage collector. Finally she stressed the importance of warm-up runs, where she stated that her group noticed notable difference between the warm-up iterations. This inspired us to look closely at latency and server behaviour at higher rates.

In general, handling the test data carefully is especially important when testing Java applications. The unpredictable nature of the Java runtime may lead to a high variance in repeated measurements, which should of course be analyzed. For instance it could be concatenated with some kind of statistical approach (Lion et al. 2016).

2.3.3 Analysing the performance and costs of reactive programming libraries in Java

Ponge et. al. compares the performance of different reactive libraries. The study (Ponge et al. 2021) may not be related to virtual threads but it provides insight in how to make comparisons of similar systems. The study compares three of the commonly used Java libraries for reactive programming including SmallRye Mutiny, RxJava and project Reactor.

The study split the tests into three domains: individual operations, I/O bound operations and multiple-operator pipelines. This is roughly the same domains as the tests we used in our study. They aimed to make these tests like what is regularly used in reactive programming. For individual and multiple operations, they compared the performance on variable transformation for the three reactive libraries. For the I/O bound operations they did two tests. The first test was file processing were read and write are blocking operations. The second test was based around network requests. Instead of using a manual delay operation they decided to create a realistic blocking operation. In this test, text from a book is fetched from a server and afterward operations are performed on the server response.

A study tested asynchronous versus thread-based HTTP servers (running Tomcat NIO and Tomcat BIO) using apache bench to simulate high loads with concurrent users (Fan & Wang 2015). The test setup was one server machine and one attacking machine both having 2.5 GHz Zeon six-core CPU and 16 GB of RAM, although only one CPU was active to reduce complexity. The use of two separate machines is another element we included in our experiments. The result was the both the asynchronous and thread-based servers had similar average response times on increasing workload. However, when looking at the tail respond times (for example 95th and 99th percentile) the asynchronous server had much

lower response time then the thread-based approaches with an increasing difference at higher loads. The authors concluded that the thread-based server did not scale well due to a limited queue size. We took note of this and decided to look closely at parameters in Spring and the JVM that we had the ability to adjust.

2.3.4 Examples of test implementation

The process of comparing different concurrent approaches in the context of I/O operations shares a lot of similarities with comparing different server applications. Both the server tests and the concurrent approaches utilize methods such as load and stress testing. Furthermore they share a lot of valuable metrics. Previous research related to comparing servers may be valuable for selecting metrics and test methods (for example how to create tests with I/O operations).

For the interested reader, here are some studies we have drawn inspiration from when designing our tests:

- **i)** Comparing .NET and Java by measuring latency and memory with different loads (Hamed & Kafri 2009).
- **ii)** Comparing different web server architectures (for example thread per client) with load testing and measuring throughput and response time (Pariag et al. 2007).
- **iii)** Comparing servers using throughput and response time for I/O heavy situations (using a MySQL database) and CPU intense situations (calculating Fibonacci) (Chitra & Satapathy 2017).
- **iv)** Comparing multi-core web-server architectures through load testing using throughput as the primary metric using a server with two hard drives and 21600 files to simulate I/O (Harji et al. 2012).
- **v)** Using load testing (large amount of HTTP requests) to compare performance of cloud-based services using throughput, response time and CPU utilization as the primary metrics(Salah et al. 2017).

2.3.5 Overhead Analysis

The three technologies under test are all part of the Java ecosystem. Despite their different implementations, each technology relies on the fundamental threading engine inherent to the Java Virtual Machine (JVM). Notably, the JVM assumes an important role in the management and scheduling of these technologies. Especially when managing reactive and virtual threads, which rely less on the operating system than platform threads. Consequently, this managerial function introduces an overhead. That is clear when examinating of the source code for these technologies. We investigated the code bases of these methods, and found indications that they may lead to extensive call stacks, which means looking at call stacks could be useful for explaining performance differences.

2.3.6 Hardware metrics

Hardware is usually a limiting factor of programs both in terms of computation speed and load capacity. From a developer's standpoint there is not much he or she can do to improve the physical hardware, but the developers do have control of making applications that utilize the hardware resources in an optimal way.

In concurrent applications with many blocking instances due to for example I/O there is a risk that the CPU resources remain idle due to non-optimal program structures where certain operations block kernel threads. This can cause queue-like conditions that have the potential to reduce throughput and increase latency. A way to detect these ineffective blocks is to investigate the CPU utilization and monitor how high it is and also how stable it is, for example going from a low utilization to a high utilization periodically during constant loads. This is something we look at in our report.

CPU utilization

Even though CPU utilization as a metric does not explicitly give insight about efficiency metrics such as latency and throughput it is still a metric that is valuable for assessing the overall performance and evaluating resource utilization. CPU utilization may provide valuable information on bottlenecks, since the metric includes the ratio of the time the CPU is idle and whether or not the system reaches saturation (full resource utilization). The optimal CPU usage for saturation level and ideal performance may vary for different architectures and machines but around 80 % seem to be a good threshold for a server (Liu & Ding 2010) (AWS) (Cloud 2024). CPU usage is also considered to be a valuable economical metric from a stakeholder perspective in cloud environment (Liu & Ding 2010). We measure CPU utilization for all our benchmarks using VisualVM.

2.4 Tools

The tools described below will be used throughout the study.

2.4.1 Vegeta

Vegeta is an open source load testing tool that generates and transmits HTTP requests and saves data such as latency and success rate to a file. Vegeta is used through the command line and is easy to integrate into e.g. python and bash scripts. The tool allows for load testing by simulating many concurrent users. There are many parameters that can be manually adjusted, such as duration, requests per second, number of concurrent users and more. The tool then returns a report containing important parameters such as latency (average, 50th, 90th, 95th and 99th percentile), throughput, the actual rate and server responses.

Vegeta is written in GO which is a relatively high performing language with built-in support for concurrency. This allows for testing with high load for the attacking machine (i.e. the client sending requests to the server). If Vegeta does not manage to uphold the requested specifications such as requests per second it will inform the users of this. This makes the tool reliable.

Vegeta was used in similar studies in this area (Beronić et al. 2022), (Pufek et al. 2020). Additionally Vegeta has been used in a lot of studies as a load testing tool. See (Schuler et al.

2021), (Park et al. 2021), (Nor Sobri et al. 2022), (Choi et al. 2021) and (Zhang et al. 2023). Other popular tools for HTTP load testing were also evaluated such as Apache JMeter and Apache benchmark. However, these tools were not as reliable in simulating a high level of concurrency and requests per second resulting in the client side tools suffering performance issues before the actual server (at least when we tested and evaluated them).

2.4.2 VisualVM

VisualVM was used for profiling the servers in our study. This tool allows for the real-time monitoring of a large number of parameters in Java applications. This includes parameters such as CPU usage, memory usage, thread metadata and garbage collector usage. VisualVM collects data about the JVM application from the JVM. This tool was chosen due to ease of use and its ability to trace multiple performance parameters and isolate resource usage to individual Java applications. Furthermore, the tool is deemed to be solid due to it having been managed by Oracle and having been used in previous studies such as (Beronić et al. 2022).

2.4.3 Wireshark

Wireshark was used by us to analyze server crashes that occurred during our experiments. When the server was running on high loads the sender was blocked and reported an error. This error is not correctly reported in Vegeta and additional tools had to be used to analysis errors and if they were due to the server or external sources.

Wireshark allows the user to analyze network traffic in a useful way. It can for instance list all of the network traffic over a connection, displaying URLs, protocols used, sender, receiver and timestamps. During a high server load the server may block additional connections, not be able to send acknowledgements or not capable of receiving acknowledgements. This can be seen in Wireshark as the server machine sends reset commands (closing the connection due to being overloaded) or through messy communication regarding DUP ACKs (packet may have been lost or received out of order) and re-transmissions. This occurs when the server's processing capacity is exceeded, and some packets might be processed out of order or dropped.

2.4.4 Flame Graphs

Flame Graphs Gregg (2016) is primarily used as a tool to visualize the stack trace outputs from profilers such as Perf, Dtrace, Jstack and Xtrace. Flame graph was implemented to get a better understanding of the stack traces that are generated by profilers. For multithreaded applications, multiple threads may perform the same procedure thus flame graph aggregates these threads to show the overall patterns of the applications related to function calls and relative frequency.

Flame graphs are relatively easy to use and they are presented in two dimensions. The y-axis represents the call stack while the x-axis represents the relative frequency and the percentage of the CPU time spend in a certain function. Figure 2.1 illustrates a sample flame graph, and a simple interpretation of it may be that functions e(), g(), i() and a() all run on the CPU after being called through the functions or procedures below them. In the example above the stack trace for 'g' is :a -> b -> c -> d -> f ->g.

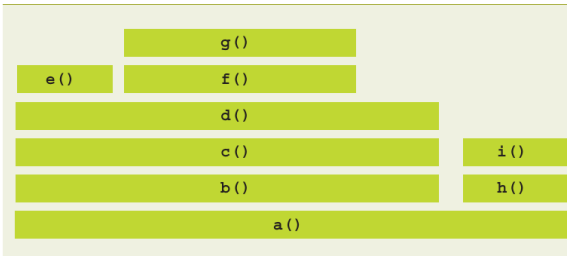


Figure 2.1: Enter Caption

In summary the flame graph is a visualization of the aggregated stack trace for an application. It can be used to investigate the number of calls required to reach the function that actually runs on the CPU and it can also be used to spot overhead. For example if function g in the figure 2.1 is the primary function, then function 'e' and 'i' may be functions that add overhead such as the garbage collector and thread scheduler.

Chapter 3
Method

3.1 Overview

An overview of the method can be described by the following steps:

1. **Design Benchmarks:** Design four benchmarks testing I/O, computation, and two mixes between computation and I/O.

2. **Create Servers:** Create three servers using platform threads, virtual threads, and reactive streams. All servers will implement the four benchmarks.

3. **Load Ramping Experiments:** In the load ramping experiments, a client machine will send concurrent requests to a server that is hosting one of the three technologies, for one of the four benchmarks. The concurrency level is decided by requests per second and will linearly increase until resource saturation occurs for the server. During these tests, the client and server metrics will be collected (such as CPU usage and latency).

4. **Constant Load Experiments:** Similar to the load ramping experiments, except now the request rate is kept constant at a level corresponding to about 70% resource utilization (of whatever the limiting factor is for the benchmark in question). Also we use one test series instead of five, as variance was much lower than for load ramping in our dry runs.

5. **Validation and Additional Experiments:** Perform validation and additional experiments to explain the results and validate the work. This includes stack profiling, examining the network connection, and assessing the impact of the profiling software.

In the following sections the method will be further elaborated regarding benchmark design, test environment and other considerations.

3.2 Test Setup

In this section we present our test setup, including the benchmark implementations.

3.2.1 Benchmark Design

In Java there are a lot of common benchmarks such as the Java Grande benchmark suite (Jils Matthew et al. 1999) that is a collection of benchmarks made to evaluate the performance of certain Java implementations. However, if these Java implementations cannot run the same or very similar source code, for example asynchronous concurrency vs virtual threads, these tests may introduce uncertainty. For example, the programmer's implementation of the benchmark of technology A and B in Java might lead to compiler optimizations for one of the technologies but not the other. That might in turn lead to unfair testing due to errors from the implementation of the benchmark rather than errors in the technology itself. To prevent possible differences in the implementation of benchmarks between different technologies (e.g. reactive vs virtual threads) a good approach is to make the benchmarks simple. The simplicity of the individual benchmarks reduces the risk that the potential differences between the technologies tested are due to implementational error. However, it is worth pointing out that these simple benchmarks may lead to benchmark specific optimizations. We have taken precautions have been taken to remove these, such as including loop variables to prevent optimizations within loops.

Since this thesis is focused on high-concurrency systems, we selected a server approach to implement the testing environment. Spring was used to run the server-side applications (benchmarks) since it is a commonly used library and thus increasing the relevance of potential results.

Four benchmarks were designed by us to test the technologies in different areas, see the previous chapter for guidelines on our benchmark design. The first benchmark tests a strictly I/O-bounded operation, the second benchmark test a strict computationally bounded operation, and the other two benchmarks test mixes between computational and I/O intense operations. Note that all I/O operations were simulated through different versions of 'thread.sleep' or 'delay' instead of using for example a database. The reason for this was to remove potential bottlenecks as the usage of a database as I/O was suspected to be a bottleneck in the study published by Navarro et al. (2023).

All four benchmarks had different parameters that regulate the behavior of said benchmark. For I/O it was sleep/delay duration and for computational tasks it was the number of floating-point operations. These parameters were manually tuned to make said benchmark test the intended area, for example CPU or I/O. The tests are available in Appendix C.

I/O Test

This method simulates a simple blocking I/O operation with minimal to zero computational operations. It is inspired by Beronić et al. (2022). Here we used a fixed delay of 100 ms. The delay might be somewhat high, but it is not unreasonable for blocking operations such as messages, fetching items from remote servers or databases. Despite the arbitrary nature of the delay length, it still does a good job of testing how well the different servers handle IO and how well they perform when the number of concurrent requests increases.

For platform threads and virtual threads, the delay is a simple thread sleep operation. For

the reactive approach the delay is set on the return object itself rather than the thread (see Appendix C), as this emulates how asynchronous applications handle blocking operations.

The fixed sleep/delay was set to 100 ms since it is a reasonable latency that can emulate realistic conditions (OpenSignal 2020) for a messaging server.

Computation Test

This method does not contain any I/O operation apart from actually receiving a request to perform a computation and returning the result. Instead, it contains a computation with squared time complexity that aims to load the different applications. The purpose with this is to investigate potential differences between the approaches in regard to simple computation.

Mixed Test - computation

As previously mentioned, one of the main concerns with virtual threads is that they can use a lot of space on the heap, as their stack frames are stored there when context switches occur. In order to test this (with a somewhat extreme case) our servers implement a matrix multiplication method. Matrix multiplication is a very common test case in performance measurements, and is also a common operation in practical applications (e.g. computer graphics). The matrix multiplication is on a 200x200 matrix, followed by a 25ms sleep operation. The dimensions of the matrix were chosen experimentally so that the load ramping test would run for a reasonable duration of time when ramping up the load.

The purpose of this benchmark was to test a mix between computational and I/O operations, with a focus on computation. Hence the matrix dimensions were tuned to put high constrains on the CPU while the sleep duration was kept short.

Mixed Test - I/O

This method is similar to above method, but the dimensions of the matrix was reduced to 150x150 and the delay increased to 100 milliseconds. This weights this test more toward I/O, while still containing significant elements of computation and memory requirements.

The purpose of this benchmark is like previous benchmark but with an additional focus on I/O and lower CPU utilization. The matrix dimensions were reduced to decrease floating point operations while the sleep/delay duration was increased.

3.3 Testing within the JVM

As mentioned in previous chapters, testing Java applications can lead to different results between iterations even though the test variables and test environment are seemingly the same. To mitigate this as much as possible the following precautions are taken. We deem them to be effective and within reasonable scope of this thesis.

Manual garbage collection between each test iteration. To allow all test iterations to have the same starting point (Lion et al. 2016). Before every iteration the attacking machine will send a HTTP request that forces the server to perform a garbage collection.

Warm up the JVM before tests (Lion et al. 2016), (Lion et al. 2016), (Navarro et al. 2023), (Pufek et al. 2020), (Beronić et al. 2021), (Beronić et al. 2022). This allows the application to load necessary classes and allow the JIT to do warm optimizations. This allows for a

more consistent results. The warm-up is performed by running a prolonged load test with the same testing method before each iteration. The warm-up also intend to force JIT compilation. The machine running the server has a threshold counter for JIT compilation of equal to 10 000 (accessed by the command (`java -XX:+UnlockDiagnosticVMOptions -XX:+PrintFlagsFinal -version`), meaning a method can be compiled after being called that many times. Thus the warm-up specifics (number of requests) will be designed to call methods more then 10 000 times with margin of safety. Another potential options is to disable the JIT compiler. However, the JIT compiler is an integral part of java and it is not recommended to disable it. Also, the JIT compiler will most likely always be running during similar conditions as this experiment test (servers), thus the study become more realistic with the JIT compiler enabled.

Back-to-back measurements will also be done, where a JVM running a certain approach will remain live between measurements (Lion et al. 2016).

Furthermore, the tests will be performed multiple times. As seen in previous research there are multiple ways to process the data. One way is to take the average of all runs (Pufek et al. 2020), (Beronić et al. 2022), but this may lead to high variance resulting in misguiding results. Another approach is to take the best performing result from each iteration (Navarro et al. 2023). The idea behind it is that the best run for all three technologies for a certain method will be compared thus leading to a fair comparison. The third approach is to take a statistical approach (Lion et al. 2016) using confidence intervals. All these options are viable, and they offer different pros and cons. Taking the average have a possibility to reduce some of the variance between the test runs. However, if one technology is very volatile, e.g. if it performs well on some runs and poorly on other runs, the results may be misleading. Taking the best iteration offers a fair comparison between the technologies considering they all are compared during their peak performance.

All these approaches were considered when designing the experiment. However it was decided to take the average of the tests, similar to previous studies (Pufek et al. 2020), (Beronić et al. 2022). This study considered the average of five tests instead of the three tests used in previous studies. Of course, taking the average of more tests could lead to a more accurate result. Due to time restrictions, five tests were the most reasonable and is still more than what previous studies did (Pufek et al. 2020), (Beronić et al. 2022), (Navarro et al. 2023).

To reduce bias the technologies will be tested with different types of benchmarks. For example, virtual threads are not recommended to be used for tasks requiring a lot of memory (Oracle Corporation 2023). Therefore, there ought to be some tests that also test them in other areas such as computation tasks, I/O tasks and of course more memory-intensive tasks.

In summary; previous research and critique seem to indicate that performance testing Java applications can lead to unpredictable results in some runs. To reduce this volatility, measures have been taken inspired by previous studies as stated in the Introduction and Background sections. Furthermore, other measurements have been taken such as monitoring the network between the machines, keeping a consistent and stable test environment and keeping the tests consistent. For example, if technology A is tested through an attack (a batch of HTTP requests) every two minutes, then technology B will also be tested that way. To ensure consistency between tests, they will be automated to ensure equal timings (e.g. time between request batches for the load ramping test), garbage collections and request increment.

3.3.1 Hardware Setup

Our experiments were run on two identical machines (MacBook Pro 2019) with the following specifications: 2.6 GHz-6 core Intel Core i7, Memory 16 GB 2667 MHz DDR4, MacOS 14.2.1. One machine hosted a server and collected profiling data while the other machine performed load testing through Vegeta. The machines were connected with an one-gigabit ethernet cable. OpenJDK version 21.0.2+13-58 was used both for the runtime environment and JVM (64 bit server VM). Spring Boot version 3.3.2 was used.

3.3.2 Metrics Collected

Both the client machine and the server machine collect metrics during the attacks. The client machine collects the following metrics: latency (minimum, maximum, mean, 50th 90th, 95th, 99th percentile), throughput, rate, server responses (for success rate), bytes in and bytes out. This is done through Vegeta.

The server machine collect metrics regarding CPU-utilization, number of threads (live and started), heap (total size and used size). The reasoning behind including these metrics is explained in previous sections. We also collect data on sync point inflations and deflations, as well as safepoints. Sync- and safepoints relate to concurrency, and will be explained quickly when they occur in the report. Studying this data might be useful for explaining our results. All data is then concatenated into files and processed.

3.3.3 Handling Errors

Both the attacking machine and server machine were monitored for errors (printed to the terminal or the output files), which only occurred under high loads or during moments where the resources (CPU, heap or threads) were very strained. Errors are not included during the comparison, but they still may be important for the study, especially when monitoring the technologies in critical zones with high loads.

On the receiving side, the errors mostly occurred due to not receiving a server response on the requests. These errors were either displayed through a token ‘EOF’ in Vegeta or through a message like ‘TCP connection reset’ in Wireshark. To confirm that the errors originated from the server side, Wireshark was used on the server machine to monitor its traffic during high loads. Note that Wireshark was not present during the actual test runs but rather incorporated after the actual tests. This confirmed that the server crashes were due to server overload and were displayed through either a clearly stated message ‘reset’ or through a chaotic dialog between the receiver and sender regarding re-transmissions and ACK-DUP. These scenarios are usually due to servers being overloaded, and they were not present during tests without errors. Of course, this does not with certainty prove that there may not be any network problems, although during the test runs the connection was monitored along side other variables related to server overload such as high resource utilization, high latency, and low throughput.

3.4 Parameters

We had to choose several parameters for our experiments, explained quickly below.

- A test run was terminated after three consecutive failed requests. After this had happened latencies were several minutes long, with falls outside the scope of this study as we are interested in practical use cases.
- We performed each test series five times and calculated averages. As each test series took several hours, doing more was not practical.
- The length and request rates used in the warm-up were calculated to call the server over ten thousand times with some margin of error, as up until that number the JVM might perform JIT-compilations (as explained in the background chapter).
- Delays of 60 seconds before garbage collections and 20 seconds after. This was decided empirically by studying time series from VisualVM and choosing numbers that allowed CPU utilization and memory usage to drop to nominal values between test iterations.

3.5 Load Ramping Experiment

The four different benchmarks described above (computation, I/O, mixed-I/O, mixed-Compute) were performed on our three different servers. They were automated through a script and consisted of staggered Vegeta attacks with a delay and a forced garbage collection in between each run. This test series was performed five times for each server-method configuration and all test series were preceded by warm-up attacks.

The tests are performed by an automated script to ensure equal timings (time for warmup, time to garbage collections etc) between the different approaches and methods. This script includes a warm-up attack between each iteration of test. After each individual test the client send a request to the server to manually perform garbage collection and after that the client machine has a constant delay before it starts the next HTTP attack.

The variable that changes between the individual attacks is requests per second (concurrency level) with increment of 50 request per second (50 chosen so that the experiments could be conducted within reasonable time while still being somewhat fine grained). This will continue until either the server fails three consecutive times or the throughput becomes too low compared to the attack rate, leading to a crash for the attacking machine due to time-outing while waiting for results (70 seconds). Only tests with a 100% success level will be considered during this experiment, however all test data will be included and discussed.

During this experiment several metrics were collected. The machine sending the HTTP requests collected latency (minimum, maximum, mean, 50/90/95/99 percentile), throughput, request per second rate (actual rate and the intended rate), total requests sent, total failed and successful requests and eventual error codes. The server machine sampled CPU usage, heap usage, number of threads, safe points, inflation points, contended lock attempts and deflation points.

The tests were performed five times each, meaning that a total of sixty tests were performed during this experiment (four test methods, three technologies tested, five test runs).

Each test iteration (for example virtual threads I/O test) was conducted like in the lists below.

Warm-up:

1. 60 seconds with 300 requests per second

2. Manual garbage collection
3. Sleep operation for 20 seconds
4. Repeat three times with rates adjusted for the specific method

The three later warm-up rates are adjusted to 70 % of the max rate for the method (as explained in the background chapter this is a good operating area for a messaging server). After the warm-up the actual tests are performed according to the list below.

Attacks:

1. Send HTTP requests with rate 'RATE' for 10 seconds. Save results.
2. Sleep operation for 60 seconds
3. Manual garbage collection
4. Sleep for 20 seconds
5. Increase 'RATE' by 25/50 repeat

The above procedure is repeated N times until three consecutive failures, e.g. the server crashes. See Appendix B for the automated script.

3.6 Constant Load Test and Additional Profiling

3.6.1 Constant Load Test

To verify the primary testing outcomes, longer test runs were conducted to investigate if the difference between the technologies were still present and that it scales with time. According to our supervisor at Sinch, servers are usually scaled up when around 80 percent of hardware resources are utilized. Thus these more specific tests were conducted at a request rate translating to about a 70 percent load of either the CPUs (computation tests) or the memory (IO tests). Since 80% resource usage means scaling up the server and that there is some variance in resource usage, a reasonable mode of operation would be a bit lower than 80% (e.g. 70%). The long runs were only performed with one specific rate that was selected to emulate realistic working conditions. The rates for the benchmarks were different but the rates were the same for all technologies, e.g. the compute method for virtual threads and reactive streams had the same request rate.

Each test run was performed according to the following:

1) 3 x 60 seconds warm-up runs with 300 requests per second each.
2) 2 minutes warm-up run with the intended request rate.
3) Manual garbage collection.
4) Delay for 60 seconds.

5) Start test. 10 minutes.

For these tests Vegeta and VisualVM were utilized to sample results.

3.6.2 Stack Profiling

To investigate the call stacks and get an overview of the overhead within these three technologies FlameGraphs were utilized. These graphs are an abstraction of the output from profilers, such as perf and Dtrace. We used Dtrace. The graphs visually show the extent of the call stack, and also approximate the CPU time spend in each method presented as a fraction of the sampling time.

The testing procedure followed the same protocols presented in previous section, including extensive warm-ups. The test itself was shorter however (at 240 seconds) as we did not see any difference in results from longer tests in our dry runs. Each test iteration was performed according to the following procedure:

1) Warm-up (3 X 60 seconds Vegeta attacks with the same rates that would be used on the actual test. Including manual garbage collections)

2) Manual garbage collection

3) Manual delay 60 seconds

4) Start of test. 240 seconds with a constant rate

This procedure was performed for all three technologies with all four methods. The rates were different between the methods but were kept the same for all three technologies. The rates were manually tuned with the goal to not crash the servers during testing while keeping the hardware utilization within realistic working conditions (i.e. 70% utilization of the limiting factor). This proved to be difficult due to some methods crashing before others. The final rates where:

- 1200 requests per second for Compute
- 1600 requests per second for I/O
- 300 requests per second for matmul compute
- 600 requests per second for matmul sleep

When the measurements were finished the profiling result was processed using Brendan Gregg's Flamegraph scripts to collapse the stacks and convert them into FlameGraphs. The current implementation of the Jstack (Oracle Help Center 2024) processing script was not updated to handle virtual threads, but a few adjustments of ours fixed that.

3.7 Validation

It would be poor form to perform these kinds of tests without proper validation to make sure that the constraints and configurations of our testing environment did not unduly influence the results. Below we present the major validations and tests of our setup.

3.7.1 Network Connection

When load testing was performed in this study, the tests ran until the server crashed or the network connection was severed. Of course, questions arose as to whether it really was server overload that led to the network shutting down. In order to test this, two experiments were performed. Firstly, an empty endpoint was set up at each of the servers (empty in that they simply returned an acknowledgement and did not force the server-side computer to take any other action). When calling this method, we should be able to reach very high rates if the network connection does not set a low limit on capacity. Secondly, a selection of our ordinary tests where performed again and monitored in WireShark (described in the previous chapter). Results of the measurements are presented in the next chapter.

3.7.2 Impact of Profiling Software

It is a fundamental law of nature that by observing something you affect it (Heisenberg 1927). It holds especially true for profiling software, as some kind of probes will have to be embedded into it and data has to be recorded by the same hardware that is supposed to be under test. To make sure that our profiling software had a negligible impact on our experiments, we ran our tests once without profiling. Through measurements on the client side it was verified that any deviation in performance was well within normal margins of error, i.e. performance as measured from the client (latency and throughput) were not noticeably affected by the absence of profiling software.

Chapter 4
Results

This study consists of two major experiments, a constant load and a ramping load benchmarking, as described in chapter 2. The results of these experiments will be presented here. When we measure latency, we present different percentiles as stated in the theory section. Here follows a quick recap of what a percentile is.

Percentile x of the probability distribution function P, henceforth denoted as P_x (or by x:th percentile) indicates the x:th percentile. Notably is the 50:th percentile P_{50} which from the probability distribution function is defined as the median value which is the same value the average (A) of N-samples converges to if N grows (in the case of symmetric P). The 50:th percentile can be defined as follows.

$$50\% = P(X \leq a) = \int_{-\infty}^{a} f(t)dt$$

More general, the P_x percentile is when the cumulative distribution function reaches $x/100$. Thus, percentiles are useful for describing worst case scenarios.

4.1 Ramping Load Experiment

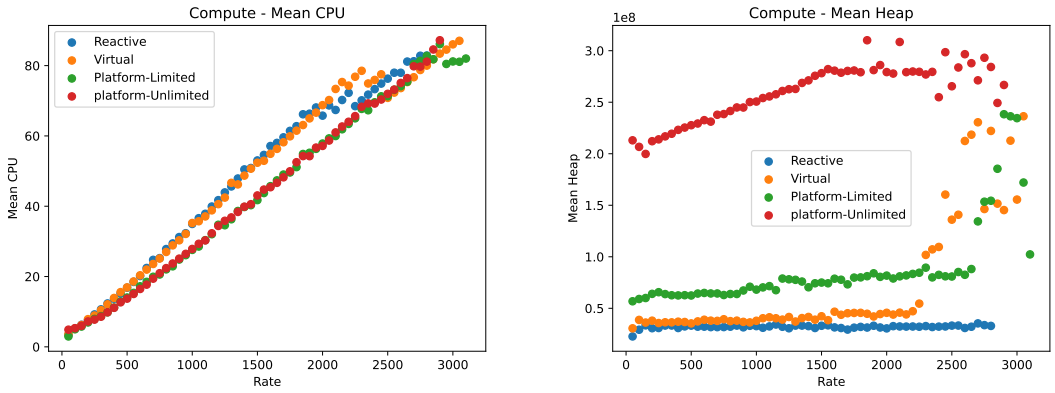
In this experiment, virtual threads, platform threads and a reactive system were tested on our custom benchmarking methods. Data from these tests will be presented below.

As stated in the theory section, the results between identical tests can vary a lot due to the inner workings of the JVM. Many precautions were taken to mitigate this, and they are explained in detail in chapter 2. Every individual test resulted in between twenty and eighty series of samples depending on the test (the samples are concatenated by VisualVM).

The rate described in the following sections is requests per second. In this section, different latency percentiles will also be displayed. The 99th percentile for latency is the latency (time) that 99% of requests are faster than. Meaning that it somewhat describes the worst-case scenarios. Due to instability of the tests. For the plots in the following section, each sample is the mean from five successful attempts.

4.1.1 CPU Intense

This method consists of computations vaguely inspired by the calculation of a Fibonacci sequence with some elements of randomness introduced (see Appendix). It contains no delays or server calls, and is thus not the intended use case for either virtual threads or reactive streams. It is however an interesting measure of performance and is thus included in this report.



(a) Mean CPU usage in percentage for reactive, platform, and virtual threads.

(b) Mean heap usage in bytes for reactive, platform, and virtual threads.

Figure 4.1: CPU and Heap usage

Figure 4.1a and 4.1b illustrates the hardware utilization attributable to the three technologies during a CPU intense operation. The CPU utilization for the methods differ significantly. The reactive approach has the highest CPU utilization overall, closely followed by virtual threads. The blocking platform threads have the lowest CPU utilization. The memory utilization for the three concurrency approaches are also different (as can be seen in figure 4.1b). At lower rates the platform threads have the highest memory utilization while virtual and reactive have lower memory utilization. However, as the request rate increases the virtual threads start to consume more memory.

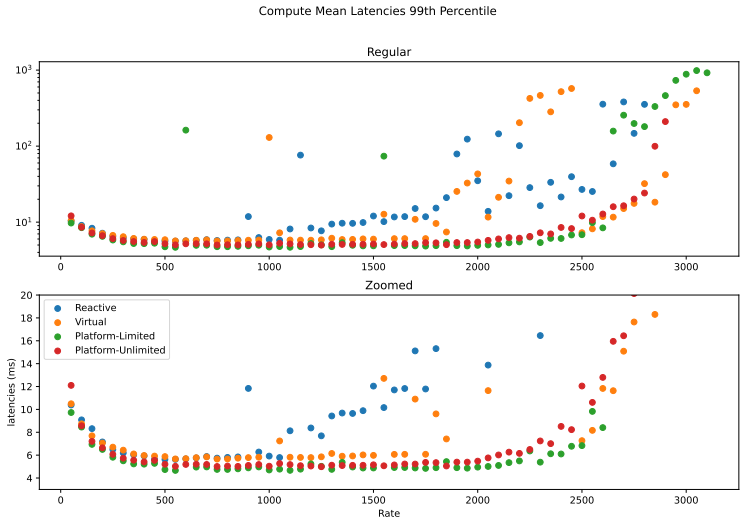


Figure 4.2: Mean of 99th percentile latency in milliseconds for reactive, platform, and virtual threads.

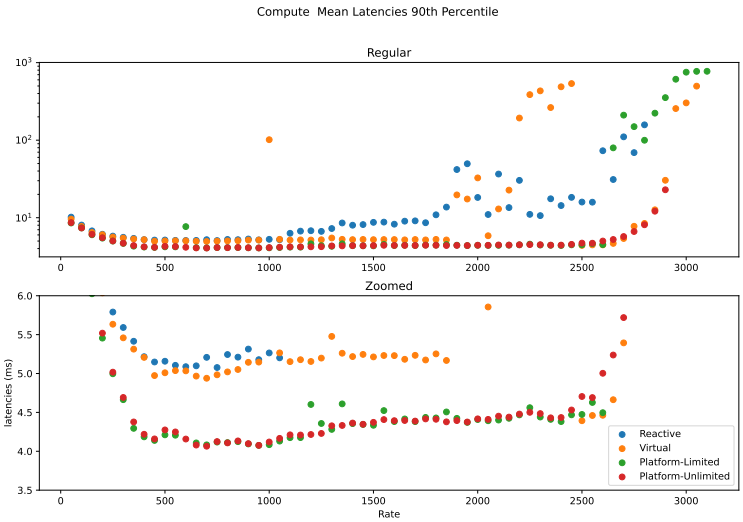


Figure 4.3: Mean 90th percentile latency in milliseconds for reactive, platform, and virtual threads.

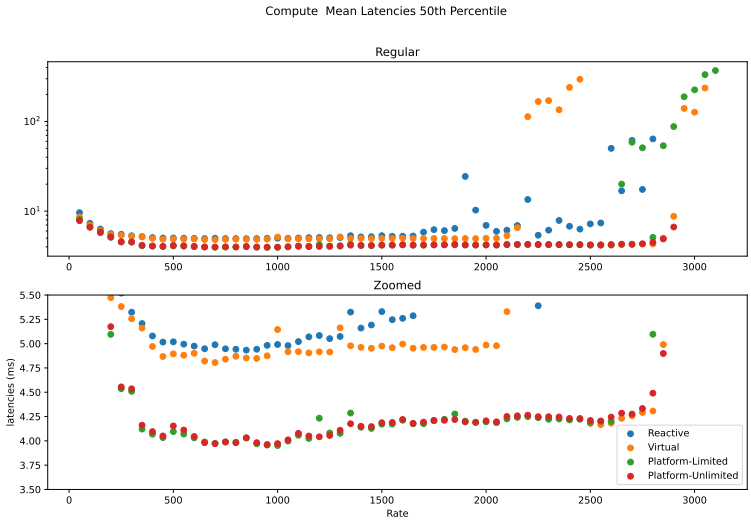


Figure 4.4: Mean 50th percentile latency in milliseconds for reactive, platform, and virtual threads.

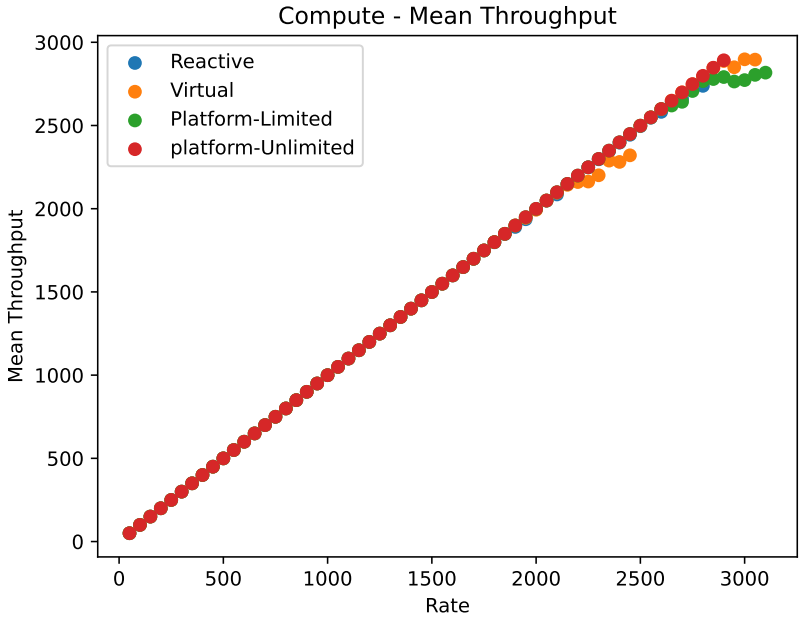
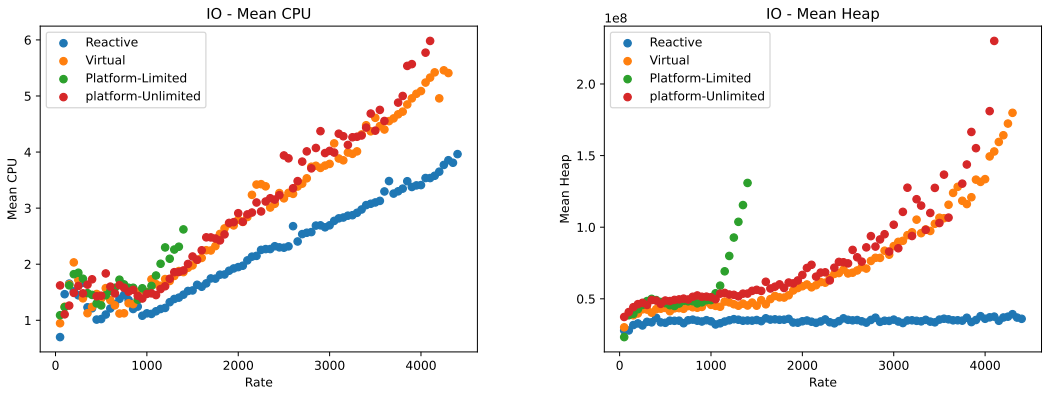


Figure 4.5: Mean throughput (messages per second) for reactive, platform, and virtual threads.

Figures 4.2, 4.3, 4.4 and 4.5 illustrate performance metrics for the different technologies. Platform threads have the best performance regarding latency in all percentiles having an almost 25 % lower latency than virtual threads and reactive streams in the 99th percentile. Virtual threads have a slightly lower latency than the reactive approach for rates lower than 2200 request per second. However, for request rates exceeding that, the latency for virtual threads increases and subsequently becomes very unstable leading to a uneven performance.

4.1.2 I/O intense

The result for all three technologies for the I/O intense operation will be presented here. First plots comparing the different approaches will be presented. After that individual plots for the three approaches will be illustrated. The purpose of the individual plots is to visualise the relation between utilization of hardware resources and performance metrics which may give insight into the technologies' performance.



(a) Mean CPU usage in percentage for reactive, platform, and virtual threads.

(b) Mean heap usage in bytes for reactive, platform, and virtual threads.

Figure 4.6: IO method CPU and Heap usage

Figures 4.6a and 4.6b illustrate the hardware utilization from the three technologies during an I/O operation. All three technologies have rather small CPU utilization (as can be seen in figure 4.6a), but in the comparison between virtual threads and the reactive approach the reactive threads clearly have a smaller CPU utilization. Note that the CPU utilization for this method is low meaning that a lot of different factors can affect it relatively much, such as a garbage collection. Both virtual and platform threads utilize a lot of heap memory compared to the reactive approach (as can be seen in figure 4.6b). When the servers are overloaded the heap usage increases drastically for platform and virtual threads.

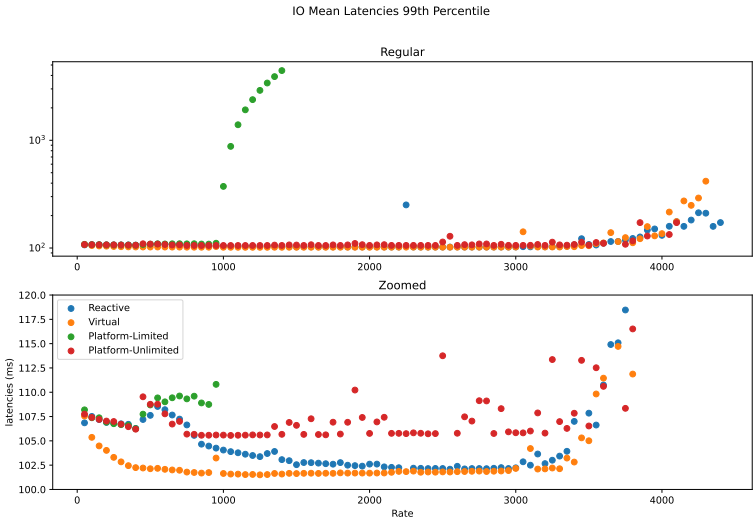


Figure 4.7: Mean of 99th percentile latency in milliseconds for reactive, platform, and virtual threads.

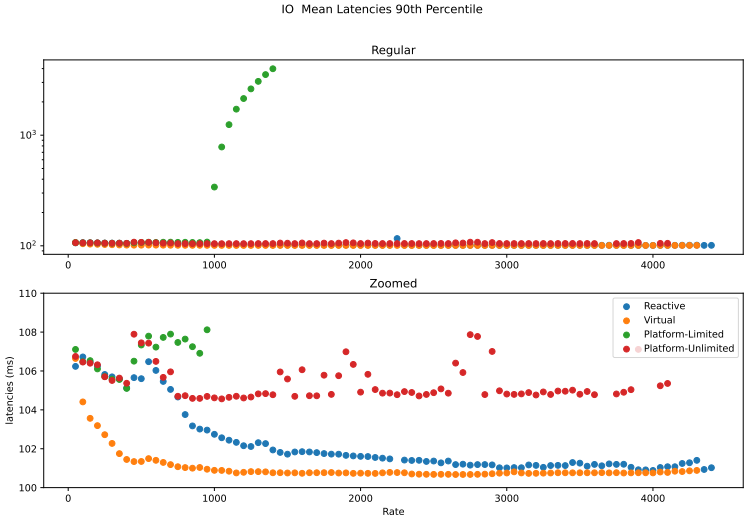


Figure 4.8: Mean of 90th percentile latency in milliseconds for reactive, platform, and virtual threads. Each sample is the mean from 5 successful attempts.

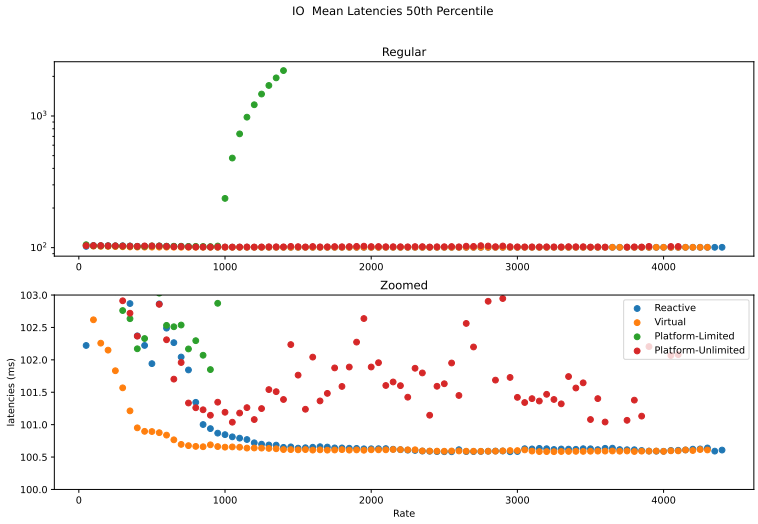


Figure 4.9: Mean 50th percentile latency in milliseconds for reactive, platform, and virtual threads.

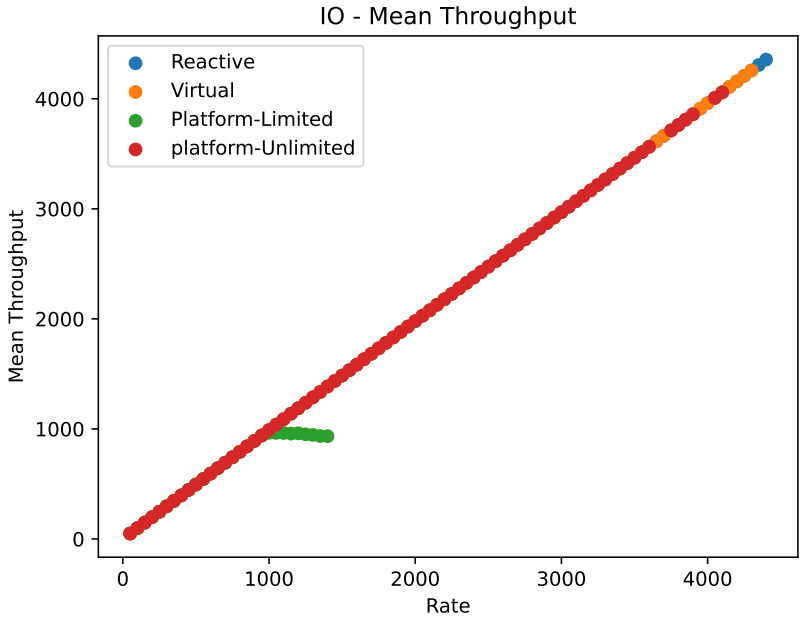
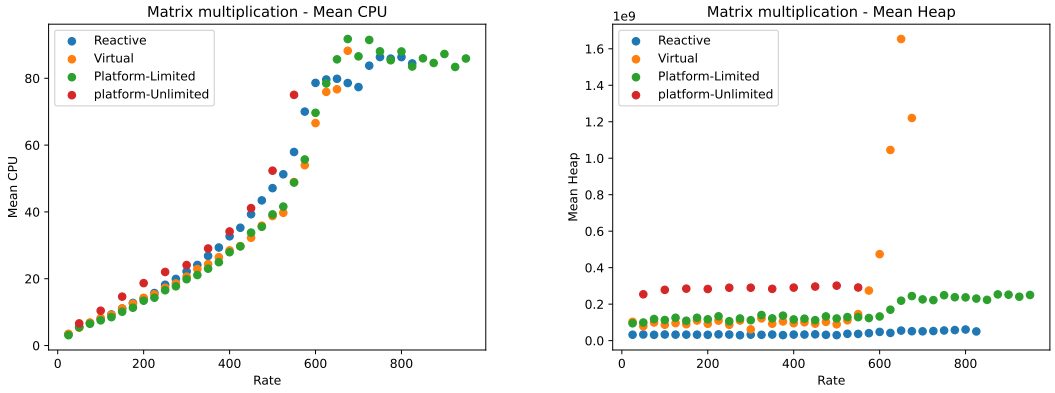


Figure 4.10: Mean throughput (messages per second) for reactive, platform, and virtual threads.

Figures 4.7, 4.8, 4.9 and 4.10 illustrate performance metrics for the three technologies. Virtual threads and reactive streams are relatively equal in terms of latency, but virtual threads has slightly better 90th percentile latency. Platform threads have a noticeably worse performance regarding latency compared to reactive and virtual threads in the median case and the 90th percentile. Reactive threads can handle higher concurrency loads performing better than virtual threads in the 99th percentile for loads higher than 4000 requests per second.

4.1.3 Matmul



(a) Mean CPU usage in percentage for reactive, platform, and virtual threads.

(b) Mean heap usage in bytes for reactive, platform, and virtual threads.

Figure 4.11: Matrix Multiplication CPU and Heap usage

Figures 4.11a and 4.11b illustrate the hardware utilization from the three technologies during a mixed operation. The CPU utilization for the methods is relatively similar (as can be seen in figure 4.11a). The reactive approach has the highest CPU utilization overall closely followed by platform threads (unlimited) and virtual threads. The memory utilization for the three technologies also differ (as can be seen in figure 4.11b). At lower rates the three approaches have similar memory utilization but as the rate increases the memory usage for both virtual- and platform threads increases exponentially.

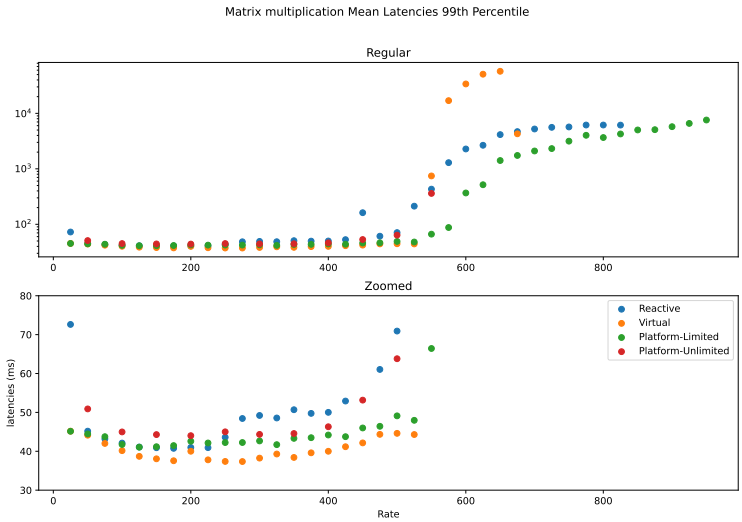


Figure 4.12: Mean of 99th percentile latency in milliseconds for reactive, platform, and virtual threads.

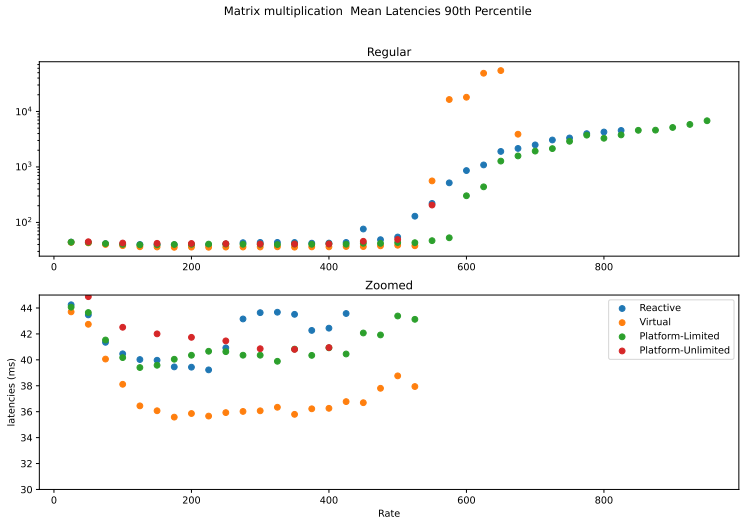


Figure 4.13: Mean of 90th percentile latency in milliseconds for reactive, platform, and virtual threads.

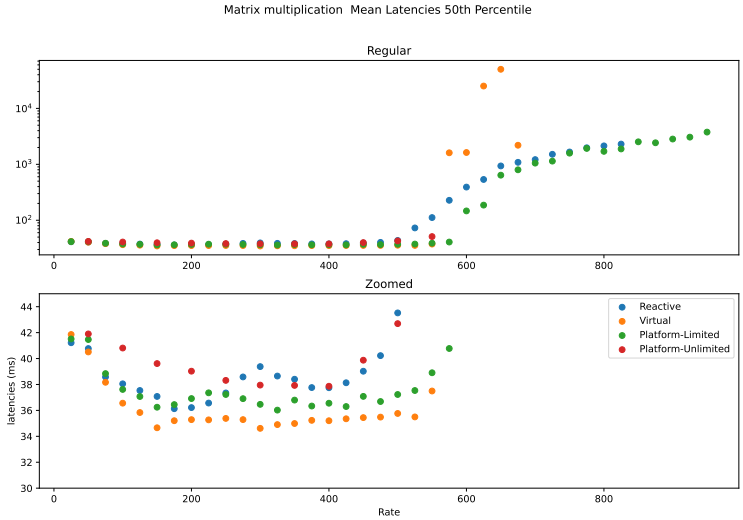


Figure 4.14: Mean 50th percentile latency in milliseconds for reactive, platform, and virtual threads.

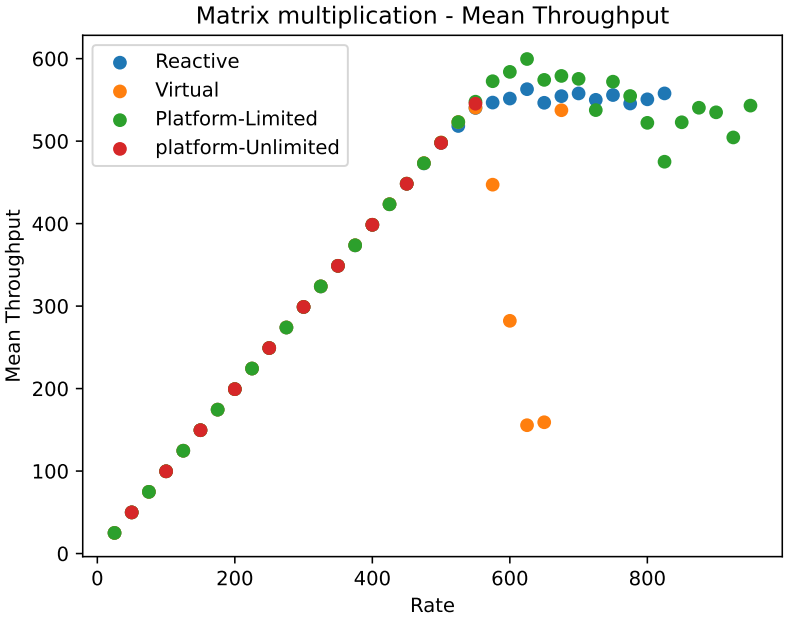
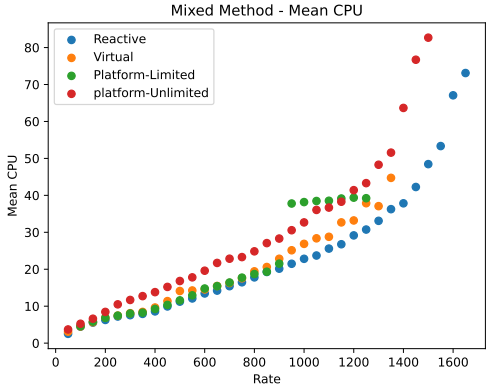


Figure 4.15: Mean throughput (messages per second) for reactive, platform, and virtual threads.

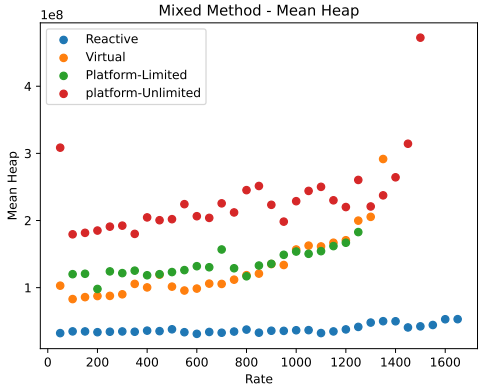
Figures 4.12, 4.13, 4.14 and 4.15 illustrate performance metrics for the three technologies. For lower rates corresponding to a CPU usage of less then 60 % Virtual threads perform the best regarding 99th, 90th and 50th percentile latencies and having similar throughput as the other two technologies. However, when the load increases to around 550 the latency for virtual threads increases drastically alongside a reduced throughput and an increasing memory utilization. This results in the server crashing. Platform threads and reactive streams also exhibit increases in latency decreases in throughput, however not drastically enough to cause a crash on the server-side. Platform threads were able to reach the highest rate.

4.1.4 Mixed method

The "mixed method" is the matrix multiplication with a 100 ms delay, meaning it entails significant loads in regards to computation, delay operations and memory usage.



(a) Mean CPU usage in percentage for reactive, platform, and virtual threads.



(b) Mean heap usage in bytes for reactive, platform, and virtual threads.

Figure 4.16: Mixed Method CPU and Heap usage

Figures 4.16a and figure 4.16a illustrate the resource utilization for the three technologies. The platform threads have the highest CPU and heap utilization while the reactive technology have the lowest.

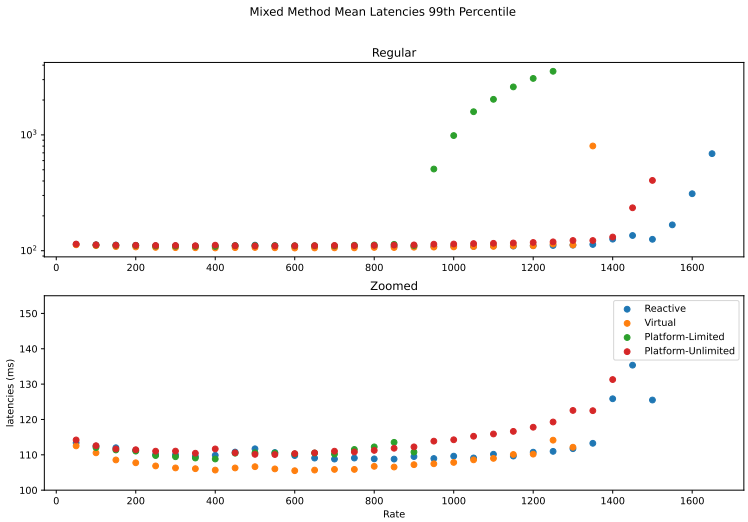


Figure 4.17: Mean of 99th percentile latency in milliseconds for reactive, platform, and virtual threads.

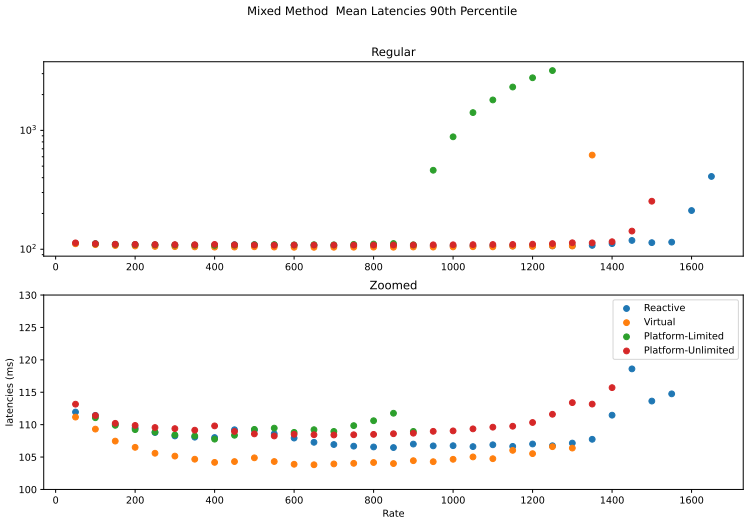


Figure 4.18: Mean of 90th percentile latency in milliseconds for reactive, platform, and virtual threads.

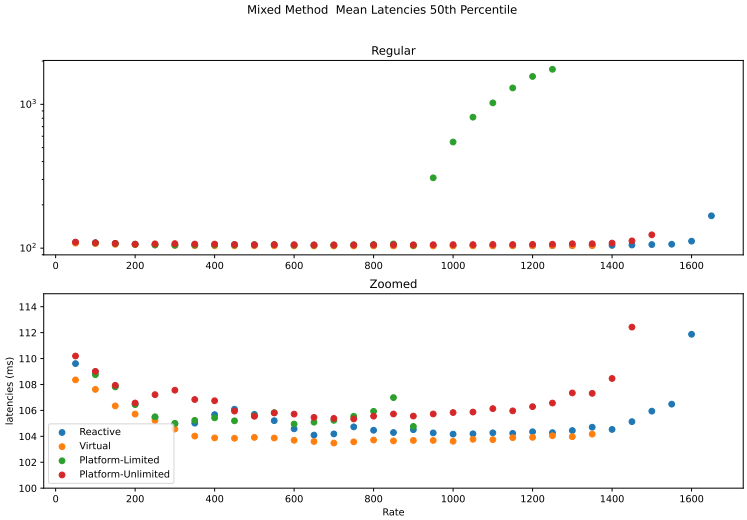


Figure 4.19: Median latency in milliseconds for reactive, platform, and virtual threads.

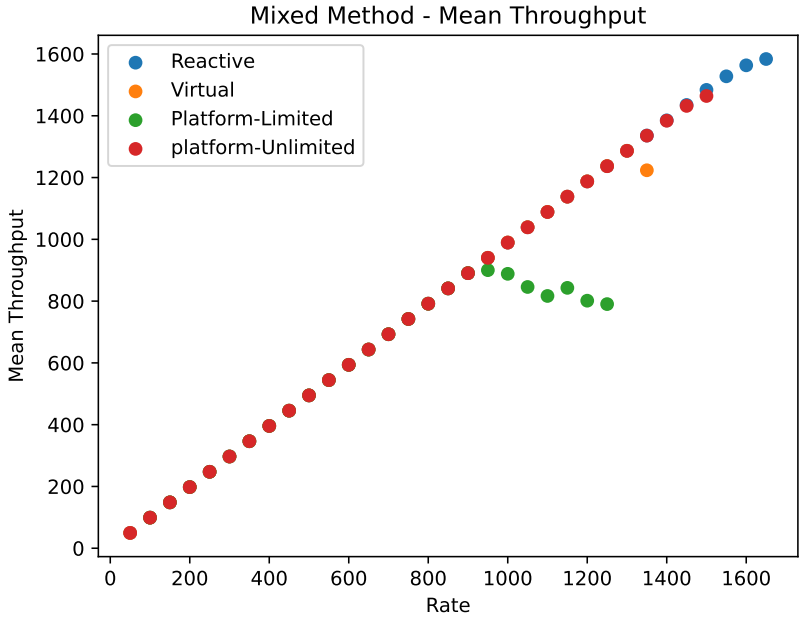


Figure 4.20: Mean throughput (messages per second) for reactive, platform, and virtual threads.

Figures 4.17, 4.18, 4.19 and 4.20 illustrate performance metrics. Virtual threads have the lowest latency in all percentiles closely followed by the reactive approach. However, the reactive approach manages to handle more requests per second. The throughput for the technologies is relatively similar although both unlimited and limited blocking threads reaches a plateau in throughput before crashing.

Overall, virtual threads have the best performance at rates less than 1300 requests per second. The reactive approach seems to be the most stable being able to handle more concurrency before failing. The platform threads have the highest hardware resource utilization where the unlimited approach utilizes more CPU and memory, due to having more threads.

4.1.5 Thread Creation

Table 4.2 and 4.1 display the number of created threads for the different methods at the start and end of the tests. Note that the thread count for limited platform is more than 100, but this is due to the thread limit being set on the threads handling incoming requests. For the reactive and virtual servers, the number of threads are not increased by much as the load increases. However, platform threads increased the thread count as the load increases.

Table 4.1: Mean live threads (start/end) for different methods

Method	Compute	IO	Matrix multiplication	Mixed method
Reactive	29 / 30	40 / 41	40 / 41	40 / 41
Virtual Thread	30 / 32	30 / 34	33 / 34	33 / 33
Platform - Limited	120 / 121	51 / 120	98 / 121	120 / 121
Platform - Unlimited	814 / 1049	98 / 716	875 / 1076	332 / 1170

Table 4.2: Mean Daemon threads (start/end) for different methods

Method	Compute	IO	Matrix multiplication	Mixed method
Reactive	27 / 28	38 / 39	38 / 39	38 / 39
Virtual Thread	27 / 28	25 / 30	29 / 30	28 / 29
Platform - Limited	115 / 117	47 / 116	94 / 117	115 / 117
Platform - Unlimited	810 / 1045	94 / 712	870 / 1072	328 / 1165

4.1.6 Stability

In figure 4.21 the percentage of failures for each benchmark is presented. Note that the failure percentage is not based on five tests for higher loads. This is due to the tests being stopped due to multiple errors in a row, some of the errors are not present in the above graph due the tests being stopped due to timeout from the attacking machine meaning these are server-side errors. From the results displayed in figure 4.21 one can deduct that the reactive servers are more stable. Furthermore, for validation purposes, another observation is that for higher rates the significance of the result for virtual and platform threads are reduced due to the averaged values containing less samples.

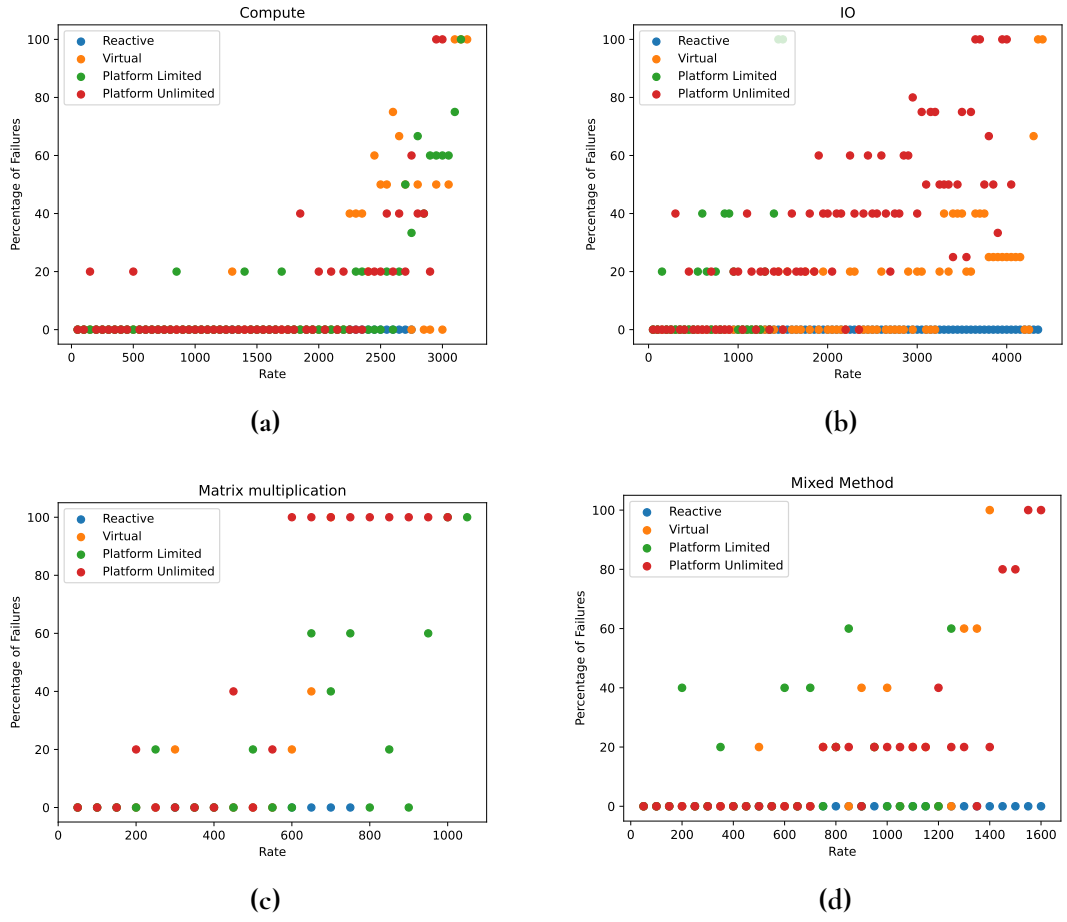


Figure 4.21: Percentage of failures for the different technologies on different benchmarks. A test is viewed as having failed if it has one or more failed requests.

4.2 Constant Load Experiment

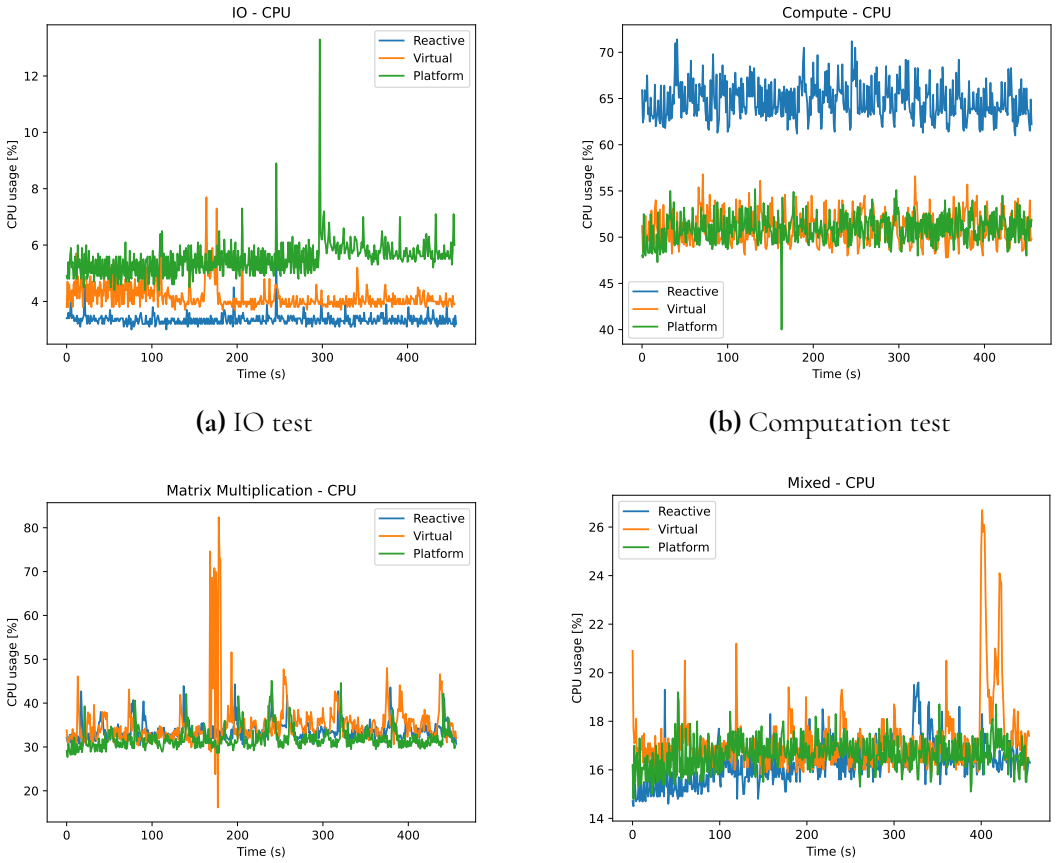
In addition to testing how well the different technologies scaled with the request rate, we also tested them under constant load. The results are presented as in the previous experiment, with the measurements of the different technologies presented alongside each other and divided by the methods tested and the measurement taken. In order to gain some deeper understanding of the difference between the technologies, the results also contain data on safepoints and contended lock attempts. Below in Table 4.3 are the latencies of the different technologies as measured in this experiment, with built in delays subtracted before normalizing:

Table 4.3: Average latencies between benchmarks for different percentiles

IO mean latencies (ms)					
Technology	Mean	50th	90th	99th	99th Normalized
Reactive	100.79	100.62	101.33	102.45	0.36
Virtual threads	100.61	100.58	100.71	101.08	0.16
Platform threads	102.28	101.26	104.93	106.80	1.00
Compute mean latencies (ms)					
Technology	Mean	50th	90th	99th	99th Normalized
Reactive	7.88	6.17	11.00	22.50	1.00
Virtual threads	4.87	4.85	5.15	5.54	0.25
Platform threads	4.96	4.80	5.01	6.08	0.27
Matmul mean latencies (ms)					
Technology	Mean	50th	90th	99th	99th Normalized
Reactive	40.26	38.83	45.32	52.13	0.05
Virtual threads	50.85	37.78	40.83	538.44	1.00
Platform threads	38.29	37.81	41.05	45.88	0.04
Mixed mean latencies (ms)					
Technology	Mean	50th	90th	99th	99th Normalized
Reactive	105.99	105.36	109.38	113.17	1.00
Virtual threads	104.17	104.00	105.13	106.61	0.50
Platform threads	105.05	104.14	107.52	108.89	0.68

4.2.1 CPU Measurements

CPU utilization was similar for the three concurrency approaches for the two mixed tests, with virtual threads being slightly more unstable than the other two approaches. However, for the IO test there were clear differences and for the computation test reactive had significantly worse performance while virtual- and platform threads performed similarly.



(a) IO test

(b) Computation test

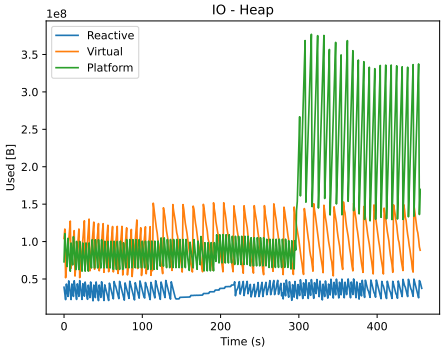
(c) Matmul test

(d) Mixed test

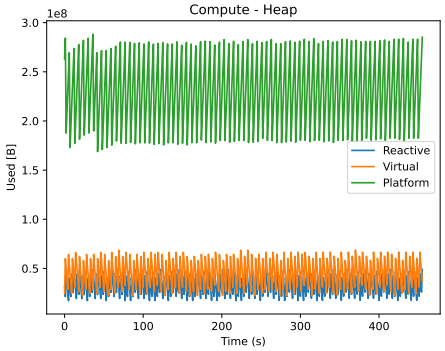
Figure 4.22: CPU usage (%) vs time (s) for different tests.

4.2.2 Memory Measurements

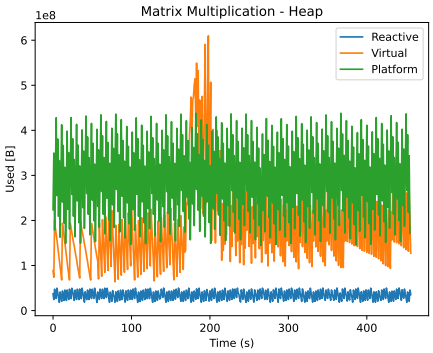
The memory measurements were similar for all of the four tests, in that performance was tiered in the same manner. The reactive server had consistently low and stable memory usage, platform threads had high and volatile memory usage and virtual threads performed on a varying sale in between reactive and platform, with volatility varying with the test method.



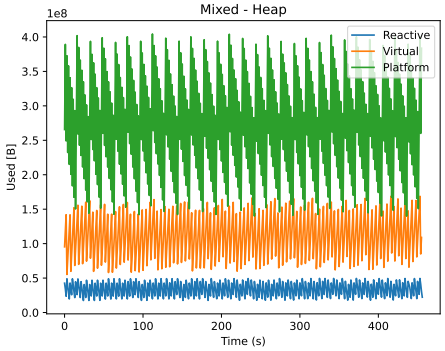
(a) IO test



(b) Computation test



(c) Matrix multiplication test

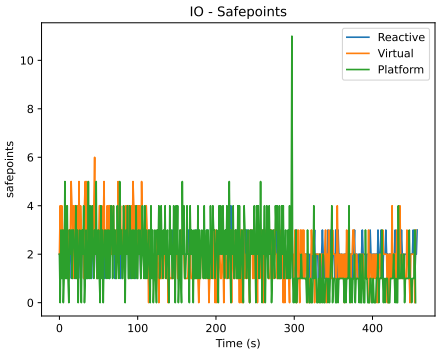


(d) Mixed test

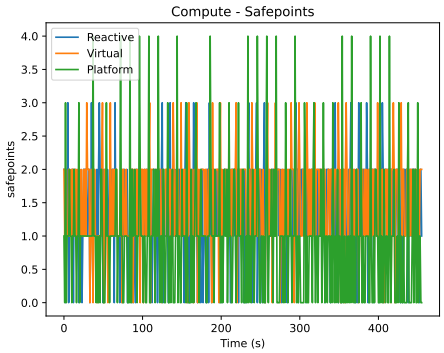
Figure 4.23: Heap usage (bytes) vs time (s) for different tests.

4.2.3 Safepoint Measurements

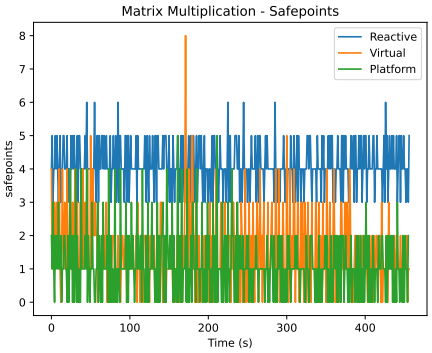
In the JVM, a safepoint represents a state in which all references on the stack are mapped and can be accounted for. The biggest difference between the technologies tested can be seen for the mixed methods, where reactive has the most safepoints. Hence allowing the reactive server to manage memory better due to more frequent garbage collection.



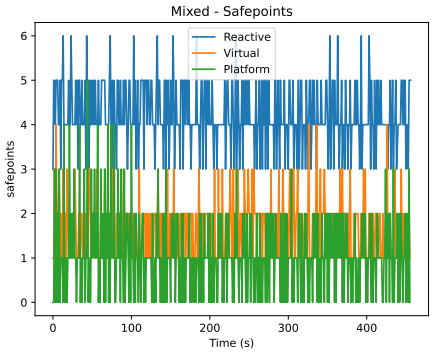
(a) IO test



(b) Compute test



(c) Matrix multiplication test

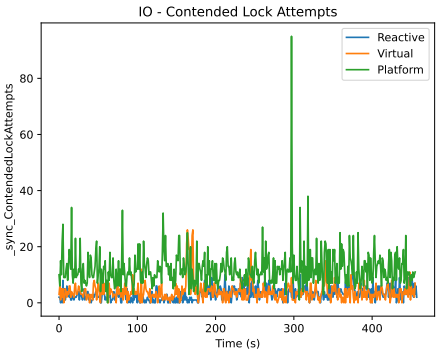


(d) Mixed test

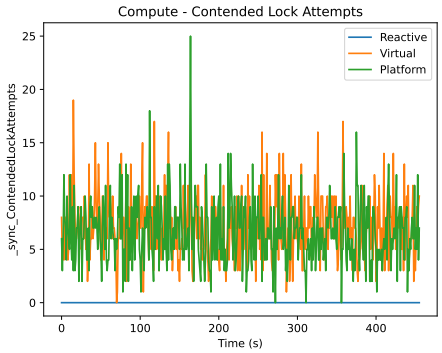
Figure 4.24: Number of safepoints vs Time (s) for different tests.

4.2.4 Sync Measurements

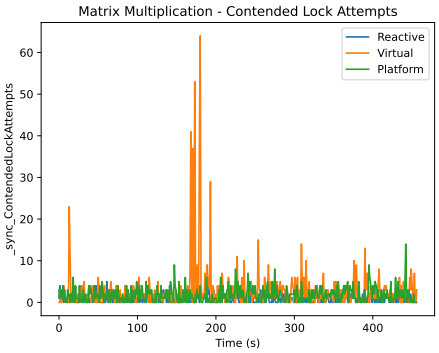
Contended lock attempts are included to better understand the performance differences presented above. They are a good measure of how well the technologies handle concurrency and synchronization, as time spent waiting for a busy lock is time spent stalling (and for virtual threads something that might cause it to be unmounted from it's kernel thread and stored on the heap). The main takeaway from the results below is that the reactive server had a consistently lower number of contended lock attempts.



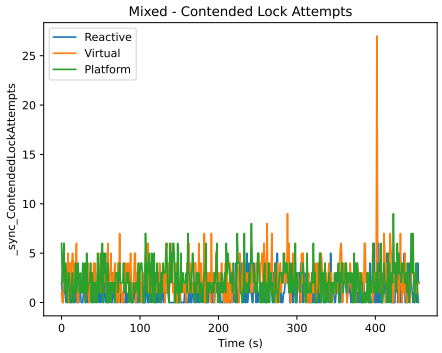
(a) IO test



(b) Computation test



(c) Matrix multiplication test



(d) Mixed test

Figure 4.25: Number of Contended lock attempts vs Time (s) for different tests.

4.2.5 Call Stack Analysis

The call stack was analyzed with Brendan Gregg's Flame graph tool, with data collected using JStack (Oracle Help Center 2024). The call stack analysis was included in this report in order to get a better understanding on how much overhead factored into eventual performance difference between the three concurrency technologies tested. We have deemed two measurements to be of special importance, namely the height of the call stacks and the widths of the calls to the methods ran on the servers (i.e. what amount of the programs' performance was used to do what we wanted it to).The reason for this is that the width of the call stack can help give a understanding how much CPU time the server spend on the actual benchmark versus managing the server and the concurrency. The height of the call stack simply indicate how many function calls certain technologies require to manage concurrency. The heights of the call stacks were:

- 56 for platform threads
- 56 for virtual threads
- 85 for reactive

Rates: 1200 requests per second for compute, 1600 for I/O, 300 matmul compute and 600 matmul sleep.

The percentage of CPU usage dedicated to the methods implemented is presented in the table below. It should be read as the complement to what percentage of the CPU usage was spent on overhead.

Table 4.4: Percentage of CPU usage spend on the actual benchmark.

CPU used for method (%)				
Concurrency type	IO	Compute	Matrix	MatrixSleep
Reactive	0	29	21	10
Virtual threads	3	43	28	20
Platform threads	0	35	27	20

From the data in table 4.4 we can deduct that Java Reactive (asynchronous concurrency) has a significant amount of overhead compared to platform and virtual threads. This is deducted by the fact that all time spend on functions outside of the primary benchmark can be viewed as logistical overhead managing the server, network connection and foremost the concurrency. Even though the height of the call stacks were the same for virtual- and platform threads, it is reasonable to conclude that the scheduler for the virtual threads (they are handled in a ForkJoinPool) is more efficient than that for platform threads.

4.2.6 Network Connection Analysis

As described in the method chapter, we conducted two tests of the network itself. First we tried calling our "empty" method at increasing rates. When doing this very high rates were reached. WireShark was then used to analyze the networks potential as a bottleneck during runs of some of our load ramping tests. The tool gave interesting insights into the anatomy of the server crashes. The most common scenario was that the client began asking for re-transmissions as it got overloaded and could not keep up. This was often soon followed by e.g. duplicate acknowledgements and other signs of inefficiencies. After this breakpoint, most of the network traffic was "overhead" until the server sent a request to close the connection. The results of these experiments speaks against the network connection being a limiting factor of this study as we were able to transmit at much higher rates than used in our experiments in the dry run and WireShark showed the errors originated from the server process.

4.3 Summary of the Results

In the I/O tests, virtual threads had the overall best performance regarding latency in the 99th and 90th percentiles. The virtual threads had up to 1ms less latency than the reactive threads in the 90th percentile(figure 4.8), with the difference being higher at lower rates. Platform threads, both limited and unlimited, had a noticeably worse performance than reactive and virtual threads for latency in all percentiles having a median performance that was 1-3 milliseconds slower for most of the attack rates, and a 90th percentile performance that was around 4-5 milliseconds slower for most rates. The platform threads limited to 100 threads achieved a latency that was much higher (around a factor twenty at mid-range) than all the other technologies. This confirms the result from Pufek et al. (2020), Beronić et al.

(2022) and Beronić et al. (2021) where limited platform threads had a significantly worse performance that virtual threads. The differences between unlimited platform threads and virtual threads were not as obvious, however unlimited platform threads used significantly more memory. The reactive approach had the lowest resource utilization in terms of memory and CPU. The heap for limited platform threads did increase drastically as the rate reached 1000 requests per second right before the approach crashed. The reason for the crash was simply that it did not have enough threads for a higher level of concurrency. A delay of 100ms means that one blocking thread can handle a maximum of ten requests per second, where 100 blocking threads can at maximum handle 1000 requests per second. After that the requests are put in a queue leading to increased latencies. Virtual and unlimited platform threads had a similar CPU utilization that increases linearly while their heap utilization increased exponentially with the request rate.

For the computationally intensive method, platform threads had the best overall 50th, 90th and 99th percentile latencies. They were around 0.75 ms faster than virtual threads in the median case. For rates less than 2200, virtual threads had a slightly better performance than reactive streams in terms of latency in all percentiles. After that the results became unstable, most likely due to a high resource utilization. Regarding resource utilization the platform threads had the lowest CPU utilization followed by virtual threads and lastly reactive. For memory utilization, reactive streams are the most efficient followed closely by virtual threads, although the memory utilization for virtual threads increases exponentially. Both the platform threads have overall worse memory utilization than reactive and virtual threads.

For the matrix multiplication with 25 milliseconds of delay, virtual threads had lower latencies for all percentiles followed by platform threads and lastly reactive. For higher rates platform threads perform better than reactive. The CPU utilization for all technologies is relatively similar although limited platform threads have a lower CPU usage than unlimited platform threads (most likely due to the additional CPU requirements created by managing more platform threads). Reactive has the lowest memory utilization overall. Both types of platform threads have a relatively constant and stable memory utilization, while that of the virtual threads increase exponentially at higher rates.

For the mixed method virtual threads had lower latencies than the other technologies in all collected samples for the 50th, 90th and 99th percentile. They were followed by reactive, with a decreasing gap as the rate increases. Limited platform threads do have an exponentially increasing latency as it cannot create more threads. Virtual threads fail before platform threads (unlimited) and reactive threads. The reactive server performed best after virtual threads had failed. For lower rates, reactive, virtual and limited platform threads had similar CPU utilization while unlimited platform threads have a higher CPU utilization. Reactive had the lowest memory utilization followed by virtual threads and limited platform threads.

The constant load tests leads to similar conclusions to the varying load ones. The reactive server runs more stably and utilizes less memory than the platform- and virtual thread servers. However it has significantly larger latencies than the virtual thread- based one.

Chapter 5
Conclusions

5.1 Addressing Research Questions

Given the collected data and the rigorous methodology we feel comfortable to answer the research questions in the context of Spring and the hardware utilized. **RQ1** asks if there is a difference between different concurrency techniques within the JVM for high load systems. The findings of this thesis indicate that there are in fact differences. The performance differences between the technologies can be summarized by:

i) For purely computational tasks platform threads perform better than virtual threads and reactive in terms of latency.

ii) For I/O-bounded tasks, virtual threads and reactive perform better than platform threads, where virtual threads has slightly better latency in the 90th and 99th percentile.

iii) For a method utilizing both I/O and computation (matrix multiplication) virtual threads has the best performance regarding latency.

iv) For a method utilizing both I/O and computation (with more I/O), virtual threads has the best performance. However, it is closely followed by reactive streams.

The hardware utilization differences between the technologies can be summarized by:

a) For purely computational tasks platform threads have the best CPU utilization while they have the worst memory utilization (using more memory) while reactive streams have the best memory utilization.

b) For purely IO tasks reactive streams have the best CPU and memory utilization. Virtual threads and unlimited platform threads have similar memory and CPU utilization even though platform threads create 20x more kernel threads.

c) For a method utilizing both I/O and computation (matrix multiplication) all technologies have similar CPU utilization. Reactive streams have the best memory utilization, while virtual threads and unlimited platform threads have a poor memory utilization. Unlimited platform threads have an overall higher memory utilization, but it scales exponentially for virtual threads.

d) For a method utilizing both I/O and computation (with more I/O) reactive streams have the best CPU and memory utilization. Unlimited platform threads have the worst memory and CPU utilization.

Another observation is that reactive streams are less inclined to fail requests compared to the other approaches.

The reason why platform threads have the best performance in purely computational tasks (i) can be explained by the approach being more effective for managing CPU resources. The reason why reactive streams and virtual threads perform better than platform threads for pure I/O (ii) can partly be explained by the fact that platform threads are limited to kernel threads where additional kernel threads lead to an increased CPU usage for overhead. The reason for virtual threads performing the best in a mix between I/O and computation (iii) and (iv) can most likely be described by less overhead (compared to reactive streams) and less resources spend on managing threads.

To explicitly answer **RQ1**, there is a difference between the technologies. Overall, virtual threads offer the best latency for all methods that contain some I/O elements while reactive streams have the overall best hardware utilization. Furthermore reactive streams are superior to the other technologies regarding failure rates. It is important to state that this conclusion is based on the experiment where the servers are not overloaded due to hardware saturation (unrealistic high usage of CPU and memory). The reason that these observations are discarded is that the servers tend to act unpredictably as they get too overloaded. Furthermore, the zero acceptance for failures means that the significance of the results decreases as the rate increase due to increased error rates for all technologies but reactive streams.

RQ2 investigates whether virtual threads is a viable alternative to reactive systems. Based on the answer from **RQ1** the answer is yes. However, it is important to state that reactive systems is still a very viable method. When choosing between virtual threads or reactive streams one must consider important factors such as performance (latency), hardware utilization and stability. The results in this thesis indicate that reactive streams have higher latencies than virtual threads, but have a superior hardware utilization (especially regarding memory) and reactive streams are overall more stable. Thus, introducing virtual threads may lead to higher hardware costs and it may lead to more failed requests even though they can be expected to be faster for some use cases.

5.2 Contribution and Future Research

Our study confirms the result from (Beronić et al. 2021), (Beronić et al. 2022) and (Beronić et al. 2022) regarding virtual threads performing better than limited platform threads for I/O operations. Furthermore the study partly extends Navarro et al. (2023) by comparing virtual threads and reactive stream in Spring applications. Our study achieved similar conclusions as Navarro et al. (2023) related to reactive streams having the best resource utilization. However, the performance results where different as Navarro et al. (2023) concluded that reactive streams had better performance, although it is very important to state that

Navarro et al. (2023) compared the performance at higher loads when the CPU utilization was between 50% and 100% while in this thesis the comparison was mainly based on results for lower loads. This was done as results on loads that lead to high hardware utilization was deemed to be unreasonable and less precise. Thus, the studies complement each other evaluating the performance of the technologies on the entire spectrum of concurrency levels. The results from our study supports their results as the latency for virtual threads tend to be worse than reactive streams for some methods during high loads. Other potential differences can be attributed to different testing methodologies and different benchmarks.

Our study contributes to the current knowledge base by giving a detailed comparison between platform threads, virtual threads, and reactive streams in a high load context. The results of our study indicates that there are differences between the technologies and that virtual threads is a valid option to be used to handle server concurrency, as it performed very well regarding latencies. The result supporting virtual threads suggests that additional research should be conducted on this subject. Areas of particular importance for future research within this subject are stability, scalability, and bigger applications (with a risk of introducing black-box effects to the tests). Potential research questions could be:

Q1. Do virtual threads, reactive streams, and platform threads scale differently with changing hardware resources?

Q2. How do reactive streams and virtual threads differ in stability?

5.3 Limitations and Considerations

In our methodology, many precautions have been taken to stabilize the testing environment and mitigate the somewhat unpredictable behaviour of the JVM. After taking these precautions, the variance between the different test iterations was reduced. However, there is still some variance within the tests which may affect the results. This is very prevalent on the upper end of the load spectrum where high hardware utilization led to an increased number of failed tests. Even though the failed tests where not considered in our study, they led to fewer samples from which to form an average which reduces the significance of some portions of our tests. This, alongside relatively unpredictable behaviour at higher loads, made it difficult to draw any conclusions regarding the results with high resource utilisation. Since a very high resource utilisation is not used in real world servers, these results were not considered for our conclusions. Another limitation of our study is hardware. The results may be different on different hardware configurations and operative systems. Our hardware setup also did not allow us to control the CPU frequency in a good manner, which makes our CPU-intensive tests less reliable. However, we believe this is somewhat mitigated by the fact that the hardware environment and exhibited temperature variations were practically equal for the different technologies.

5.4 Conclusions in Summary

This thesis set out to investigate three concurrency techniques, namely virtual threads, reactive streams and blocking platform threads. The overall goal was to investigate if virtual threads are a viable option for highly concurrent applications. Two research questions were

formulated. **RQ1** set up to investigate whether there is any difference in performance between the different technologies. The answer to **RQ1** was yes where the primary findings indicate that virtual threads exhibit overall better performance than the other systems during I/O operations. However, reactive streams have better resource utilization and are far less prone to fail client requests. **RQ2** asks whether virtual threads are a viable alternative to reactive streams. The answer to **RQ2** was **yes**, but when selecting the technology one has to consider fast response times vs tolerance for failed requests and hardware resources. If fast response time is essential and there is some tolerance for failed requests, virtual threads may be a better option than reactive streams. But if there is a zero tolerance to failed requests and constraints on hardware reactive streams may be a better option.

In summary, this thesis found distinct differences between the technologies. The results indicate that virtual threads may offer advantages in performance. This alongside the fact that the implementation of virtual threads is easy compared to reactive streams indicate that further research ought to be done in the subject. Areas of interested in further research are related to scalability and stability. We also encourage the reader to draw their own conclusions from the wealth of data presented here.

Bibliography

(AWS), A. W. S. (2024), 'Amazon ecs best practices - capacity providers and auto scaling'.
URL: *https://docs.aws.amazon.com/AmazonECS/latest/bestpracticesguide/capacity-autoscaling.html*

Belson, B., Holdsworth, J., Xiang, W. & Philippa, B. (2019), 'A survey of asynchronous programming using coroutines in the internet of things and embedded systems', *ACM Transactions on Embedded Computing Systems (TECS)* **18**(3), 1–21.

Ben Weidig (2023), 'Looking at java 21: Virtual threads'.
URL: *https://belief-driven-design.com/looking-at-java-21-virtual-threads-bd181/*

Beronić, D. (2024), 'Dialog with researcher regarding their previous research', Personal communication. Email correspondence sent on 2024/03/15.

Beronić, D., Modrić, L., Mihaljević, B. & Radovan, A. (2022), Comparison of structured concurrency constructs in java and kotlin–virtual threads and coroutines, *in* '2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)', IEEE, pp. 1466–1471.

Beronić, D., Pufek, P., Mihaljević, B. & Radovan, A. (2021), On analyzing virtual threads–a structured concurrency model for scalable applications on the jvm, *in* '2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)', IEEE, pp. 1684–1689.

Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z. et al. (2006), The dacapo benchmarks: Java benchmarking development and analysis, *in* 'Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications', pp. 169–190.

Brian Olson (2019), 'Asynchronous programming is really hard'.
URL: *https://devblabs.medium.com/asynchronous-programming-is-really-really-hard-8f7d97d7cddf*

Brodu, E., Frénot, S. & Oblé, F. (2015), Toward automatic update from callbacks to promises, *in* 'Proceedings of the 1st Workshop on All-Web Real-Time Systems', pp. 1–8.

Bull, J. M., Smith, L., Westhead, M. D., Henty, D. & Davey, R. (1999), A methodology for benchmarking java grande applications, *in* 'Proceedings of the ACM 1999 conference on Java Grande', pp. 81–88.

Chitra, L. P. & Satapathy, R. (2017), Performance comparison and evaluation of node. js and traditional web server (iis), *in* '2017 International Conference on Algorithms, Methodology, Models and Applications in Emerging Technologies (ICAMMAET)', IEEE, pp. 1–4.

Choi, B., Park, J., Lee, C. & Han, D. (2021), phpa: A proactive autoscaling framework for microservice chain, *in* '5th Asia-Pacific Workshop on Networking (APNet 2021)', pp. 65–71.

Cloud, G. (2024), 'Google cloud bigtable - monitoring an instance'.
URL: *https://cloud.google.com/bigtable/docs/monitoring-instance*

Edwards, J. (2009), Coherent reaction, *in* 'Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications', pp. 925–932.

Eeckhout, L., Georges, A. & De Bosschere, K. (2003), How java programs interact with virtual machines at the microarchitectural level, *in* 'Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications', pp. 169–186.

Fan, Q. & Wang, Q. (2015), Performance comparison of web servers with different architectures: A case study using high concurrency workload, *in* '2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)', IEEE, pp. 37–42.

Gregg, B. (2016), 'The flame graph', *Communications of the ACM* **59**(6), 48–57.

Gu, D., Verbrugge, C. & Gagnon, E. M. (2006), Relative factors in performance analysis of java virtual machines, *in* 'Proceedings of the 2nd international conference on Virtual execution environments', pp. 111–121.

Haines, S. (2006), *Pro Java EE 5*, Springer.

Hamed, O. & Kafri, N. (2009), Performance testing for web based application architectures (. net vs. java ee), *in* '2009 First International Conference on Networked Digital Technologies', IEEE, pp. 218–224.

Harji, A. S., Buhr, P. A. & Brecht, T. (2012), Comparing high-performance multi-core web-server architectures, *in* 'Proceedings of the 5th Annual International Systems and Storage Conference', pp. 1–12.

Heisenberg, W. (1927), 'Über den anschaulichen inhalt der quantentheoretischen kinematik und mechanik', *Zeitschrift für Physik* **43**(3-4), 172–198.

Jiang, Z. M. & Hassan, A. E. (2015), 'A survey on load testing of large-scale software systems', *IEEE Transactions on Software Engineering* **41**(11), 1091–1118.

Jiang, Z. M., Hassan, A. E., Hamann, G. & Flora, P. (2009), Automated performance analysis of load tests, *in* '2009 IEEE International Conference on Software Maintenance', IEEE, pp. 125–134.

Jils Matthew et al. (1999), 'Analysis and development of java grande benchmarks'.
URL: *https://www.researchgate.net/publication/2610597_Analysis_and_Development_of_Java_Grande_Benchmarks*

Jonas Bonér, e. a. (2014), *The Reactive Manifesto*.

Kambona, K., Boix, E. G. & De Meuter, W. (2013), An evaluation of reactive programming and promises for structuring collaborative web applications, *in* 'Proceedings of the 7th Workshop on Dynamic Languages and Applications', pp. 1–9.

Kim, K.-J., Jeong, I.-J., Park, J.-C., Park, Y.-J., Kim, C.-G. & Kim, T.-H. (2007), 'The impact of network service performance on customer satisfaction and loyalty: High-speed internet service case in korea', *Expert Systems with Applications* **32**(3), 822–831.
URL: *https://www.sciencedirect.com/science/article/pii/S0957417406000388*

Lion, D., Chiu, A., Sun, H., Zhuang, X., Grcevski, N. & Yuan, D. (2016), {Don't} get caught in the cold, warm-up your {JVM}: Understand and eliminate {JVM} warm-up overhead in {Data-Parallel} systems, *in* '12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)', pp. 383–400.

Liu, M. & Ding, X. (2010), On trustworthiness of cpu usage metering and accounting, *in* '2010 IEEE 30th International Conference on Distributed Computing Systems Workshops', pp. 82–91.

Madsen, M., Lhoták, O. & Tip, F. (2017), 'A model for reasoning about javascript promises', *Proceedings of the ACM on Programming Languages* **1**(OOPSLA), 1–24.

Navarro, A., Ponge, J., Le Mouël, F. & Escoffier, C. (2023), Considerations for integrating virtual threads in a java framework: a quarkus example in a resource-constrained environment, *in* 'DEBS'2023-17TH ACM International Conference on Distributed and Event-Based Systems'.

Nor Sobri, N. A., Abas, M. A. H., Mohd Yassin, A. I., Megat Ali, M. S. A., Md Tahir, N. & Zabidi, A. (2022), 'Database connection pool in microservice architecture', *Journal of Electrical and Electronic Systems Research (JEESR)* **20**, 29–33.

OpenSignal (2020), 'Average 4g and 3g network latency by provider in the united states in 2019 (in milliseconds)'.
URL: *https://www-statista-com.ludwig.lub.lu.se/statistics/818205/4g-and-3g-network-latency-in-the-united-states-2017-by-provider/*

Oracle Corporation (2023), 'Java 21 documentation'.
URL: *https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-2BCFC2DD-7D84-4B0C-9222-97F9C7C6C521*

Oracle Corporation (2024*a*), 'Java 21 documentation'.
URL: *https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html#GUID-8AEDDBE6-F783-4D77-8786-AC5A79F517C0*

Oracle Corporation (2024*b*), 'Visualvm'.
URL: *visualvm.github.io*

Oracle Help Center (2024), 'Jstack'.
URL: *https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jstack.html*

Pariag, D., Brecht, T., Harji, A., Buhr, P., Shukla, A. & Cheriton, D. R. (2007), 'Comparing the performance of web server architectures', *ACM SIGOPS Operating Systems Review* **41**(3), 231–243.

Park, J., Choi, B., Lee, C. & Han, D. (2021), Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices, *in* 'Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies', pp. 154–167.

Ponge, J., Navarro, A., Escoffier, C. & Le Mouël, F. (2021), Analysing the performance and costs of reactive programming libraries in java, *in* 'Proceedings of the 8th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems', pp. 51–60.

Pufek, P., Beronić, D., Mihaljević, B. & Radovan, A. (2020), Achieving efficient structured concurrency through lightweight fibers in java virtual machine, *in* '2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)', IEEE, pp. 1752–1757.

Reactive Streams (2024), Reactive streams.
URL: *https://www.reactive-streams.org/*

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M. & Al-Hammadi, Y. (2017), Performance comparison between container-based and vm-based services, *in* '2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)', IEEE, pp. 185–190.

Schuler, L., Jamil, S. & Kühl, N. (2021), Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments, *in* '2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)', IEEE, pp. 804–811.

Sim, S. E., Easterbrook, S. & Holt, R. C. (2003), Using benchmarking to advance research: A challenge to software engineering, *in* '25th International Conference on Software Engineering, 2003. Proceedings.', IEEE, pp. 74–83.

Spring (2024), Spring web reactive.
URL: *https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#web.reactive*

Spring (2024), 'Spring framework'.
URL: *https://spring.io/projects/spring-framework*

Vahalia, U. (1996), *UNIX internals : the new frontiers.*, An Alan R. Apt book, Prentice Hall.

Zhang, Y., Li, M. & Tong, F. (2023), 'Energy-efficient load balancing for divisible tasks on heterogeneous clusters', *Transactions on Emerging Telecommunications Technologies* **34**(10), e4829.

Appendices

Appendix A

A

A.1 Scripts

A.1.1 Script for sending HTTP requests

This script was used on the client for sending HTTP requests with Vegeta and saving the results nicely in a file.

```
#!/bin/bash
# Called with: bash bashScript "testSleep"  "B"  "S" "output" "final" "10s"
ulimit -u 4000
ulimit -n 20000
ulimit -s 20000
ulimit unlimited


# warm-up
echo "POST http://169.254.249.75:8080/$1"  | vegeta attack -duration=60s -rate=30
     | tee results.bin | vegeta report
curl -X POST "http://169.254.249.75:8080/gc"
Sleep 20

# Adjust rate to around 60 CPU usage
echo "POST http://169.254.249.75:8080/$1"  | vegeta attack -duration=60s -rate=10
    | tee results.bin | vegeta report
curl -X POST "http://169.254.249.75:8080/gc"
Sleep 20

echo "POST http://169.254.249.75:8080/$1"  | vegeta attack -duration=60s -rate=10
     | tee results.bin | vegeta report
```

```
curl -X POST "http://169.254.249.75:8080/gc"
Sleep 20

for i in {1..150}
do
 echo "start Attack rate :  :" $(( 50 * $i ))
 echo "POST http://169.254.249.75:8080/$1" | vegeta attack -duration="$6"
    -timeout=70s -rate=$(( 50 * $i )) -timeout=0 -max-workers 100000
    | tee results.bin | vegeta report --type json |
    python3 script.py -rate=$(( 50 * $i )) -type="$2" -method="$3"
    -fileName="$4" -durat="$6"
 echo "sleep too reset"
 Sleep 60
 curl -X POST "http://169.254.249.75:8080/gc"
 Sleep 20



python3 processAttack.py -fileName="$4"  -resultFile="$5"
```

Appendix B
Data

B.1 Iteration tests

Method	Compute	IO	Matrix multiplication	Mixed method	Sum
Reactive	247	423	160	165	995
Virtual Thread	251	373	109	115	848
Platform - Limited	276	123	139	105	643
Platform - Unlimited	265	257	51	125	698
Sum	1039	1176	459	510	3184

Table B.1: Number of observations separated by benchmark and technology.

Method	Compute	IO	Matrix multiplication	Mixed method	Sum
Reactive	10868	18612	7040	7260	43780
Virtual Thread	11044	16412	4796	5060	37312
Platform - Limited	12144	5412	6116	4620	28292
Platform - Unlimited	11660	11308	2244	5500	30712
Sum	45716	51744	20196	22440	140096

Table B.2: Number of total samples (rows times columns) separated by benchmark and technology.

Appendix C
Benchmark Methods

C.1 Imperative

C.1.1 IO

```
public void io() throws InterruptedException{
    Thread.sleep(DELAY);
}
```

C.1.2 Compute

```
public double compute() {
  double result = 0.0;
  double rand = System.currentTimeMillis();
  int N = 1000;
  for (int i = 3; i < N; i += 1) {
    for (int j = 1; j < N; j += 1) {
      result += rand / 3;
      rand -= j;
    }
  }

  return result;
}
```

C.1.3 Matmul and Mixed

The dim1, dim2 and delay parameters were changed between the matmul and mixed tests in order to achieve the desired properties.

```
  public long matmul() throws InterruptedException{
  Random rand = new Random();
  long[][] m = new long[dim1][dim2];

  for (int i = 0; i < dim1; ++i) {

    m[rand.nextInt(dim1)][rand.nextInt(dim1)] = System.currentTimeMillis();

    for (int j = 0; j < dim1; ++j) {
      m[i][j] = i * j;
      for (int k = 0; k < dim2; ++k) m[i][j] += a[i][k] * b[k][j];
    }
  }

  Thread.sleep(DELAY);

  return m[rand.nextInt(dim1)][rand.nextInt(dim1)];
}
```

C.2 Reactive

C.2.1 IO

```
  public Mono<ServerResponse> io(ServerRequest request) {

  return ServerResponse.ok()
      .contentType(MediaType.APPLICATION_JSON)
      .body(BodyInserters.fromValue(("success")))
      .delayElement(d);
}
```

C.2.2 Compute

```
  // helper function for compute test
private double calc_res() {
  double result = 0.0;
  double rand = System.currentTimeMillis();
  int N = 1000;
  for (int i = 3; i < N; i += 1) {
    for (int j = 1; j < N; j += 1) {
      result += rand / 3;
      rand -= j * i;
    }
  }

  return result;
```

```
}

// cpu intensive way of testing concurrency
public Mono<ServerResponse> compute(ServerRequest request) {
  double result = calc_res();
  return ServerResponse.ok()
      .contentType(MediaType.APPLICATION_JSON)
      .body(BodyInserters.fromValue(result));
}
```

C.2.3 Matmul and Mixed

The dim1, dim2 and d parameters were changed between the matmul and mixed tests in order to achieve the desired properties.

```
public Mono<ServerResponse> matmul(ServerRequest r) {
  long[][] m = new long[dim1][dim2];
  Random rand = new Random();

  for (int i = 0; i < dim1; ++i) {
    m[rand.nextInt(dim1 - 1)][rand.nextInt(dim1 - 1)] = System.currentTimeMilli
    for (int j = 0; j < dim1; ++j) {
      m[i][j] = i * j;
      for (int k = 0; k < dim2; ++k) m[i][j] += a[i][k] * b[k][j];
    }
  }
  return ServerResponse.ok()
      .contentType(MediaType.APPLICATION_JSON)
      .body(BodyInserters.fromValue(m[rand.nextInt(dim1 - 1)][rand.nextInt(dim1
      .delayElement(d);
}
```

Institutionen för Datavetenskap | Lunds Tekniska Högskola | Redovisas 31 maj 2024
Examensarbete: Study of reactive services vs regular blocking vs virtual threads for a high-load service
Studenter: Oliver Nederlund Persson, Elias Gustafsson
Handledare: Jonas Skeppstedt (LTH), Samuel Alberius (Sinch AB)
Examinator: Flavius Gruian

Undersökning av asynkron programmering, blockerade trådar och virtuella trådar för högt belastade system.

Populärvetenskaplig Sammanfattning av ***Oliver Nederlund Persson, Elias Gustafsson***

Java är ett av världens mest använda programmeringsspråk som utgör basen för ett flertal populära dataprogram, till exempel *Minecraft* och globala plattformar såsom *Netflix* och *Spotify*. 2023 introducerades Java 21 som innehåller ett nytt koncept för språket, virtuella trådar. Virtuella trådar avses kunna förbättra diverse programs förmåga att hantera många användare parallellt på ett effektivt sätt. Denna studie visar att virtuella trådar har god potential.

Det är inte gångbart för många applikationer att hantera en användare i taget. Ifall det till exempel tar 0,1 sekunder att skicka ett meddelande över internet för en användare, så hade det kunnat ta en timme för mottagaren att få meddelandet om 36 000 användare hade skickat samtidigt. Detta är helt enkelt inte acceptabelt för de flesta applikationer och således vill programmerare att allt detta skall ske parallellt, det vill säga att tiden det tar för en användare att skicka ett meddelande i ovanstående exempel är nära 0.1 sekunder oberoende av antalet användare. Detta kallas för flertrådad programvara och det bygger på programs förmåga att använda hårdvaruresurser på ett effektivt sätt. Det finns flertal tekniker för att göra detta, inom denna studie undersöker vi virtuella trådar, plattformstrådar och asynkron programmering.

Platfformstrådar är kraftiga trådar som kan hantera flera uppgifter, men de kan blockera hårdvara när de väntar på att en uppgift skall slutföras. Asynkron programmering löser plattformstrådarnas problem genom att inte vänta på att en uppgift skall slutföras innan nästa uppgift påbörjas (eftersom de arbetar icke-sekventiellt). Virtuella trådar är lättviktiga trådar som ej blockerar datorer när de väntar på att uppgifter skall lösas. Detta eftersom de monteras nedöch sparas i minnet medan de inte kan arbeta.

Denna studie testar dessa tre tekniker genom att implementera tre olika servrar som drivs av de tre olika teknikerna. Därefter konstrueras fyra tester som testar de olika teknologierna i olika arbetsområden såsom beräkningstunga operationer, minnestunga operationer och framför allt I/O-begränsade operationer, där I/O innebär att datorn tar emot och skickar ut data. Slutligen skickar en annan dator förfrågningar till servrarna med varierande belastning som simulerar flera samtidiga användare. Ett flertal steg togs för att säkerställa en bra testmiljö, vilket är A och O för den här typen av studier.

Resultaten av dessa tester visar att virtuella trådar i de flesta fall erbjuder bäst prestanda (d.v.s. är snabbare) än alternativen. Däremot använder en server som drivs av asynkron programmering minst hårdvaruresurser och har den lägsta felfrekvensen. I det testfall som inte innehöll några blockerande operationer presterade de vanliga platformtrådarna bäst.

Implikationerna av detta är att valet mellan att använda asynkron programmering eller virtuella trådar för en server handlar om en avvägning mellan prestanda, stabilitet och hårdvarukostnader.