TU WIEN Informatics

Nebenläufigkeit in Java mit Threads und Reactive Programmierung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Ievgenii Gruzdev, Bsc.
Matrikelnummer 01225821

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Pof. Dipl.-Ing. Dr.techn. Franz Puntigam

Wien, 19. März 2025

Ievgenii Gruzdev Franz Puntigam

TU WIEN Informatics

Concurrency in Java with Threads and Reactive Programming

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Ievgenii Gruzdev, Bsc.
Registration Number 01225821

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Pof. Dipl.-Ing. Dr.techn. Franz Puntigam

Vienna, 19th March, 2025

Ievgenii Gruzdev	Franz Puntigam

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Ievgenii Gruzdev, Bsc.

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix “Overview of Generative AI Tools Used” I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Wien, 19. März 2025

Ievgenii Gruzdev

Danksagung

An dieser Stelle möchte ich meinen besonderen Dank nachstehenden Personen entgegen bringen, ohne deren Mithilfe die Anfertigung dieser Masterarbeit niemals zustande gekommen wäre.

Besonderen Dank kommt meiner Mutter Alla Hruzdeva zu, die mir immer wieder Zuversicht gegeben hat, auch in schwierigen Zeiten nicht aufzugeben und mein Ziel konsequent weiter zu verfolgen.

Ich danke auch Ao.Univ.Pof. Dipl.-Ing. Dr.techn. Franz Puntigam für die Betreung meiner Masterarbeit und sein Verständniss für alle meine Fragenstellungen. Er hat mich immer wieder mit der richtigen Hilfestellung auf den richtigen Weg geführt. Damit konnte ich mich besser auf die zentralen Fragestellungen der Arbeit konzentrieren.

Ein herzliches Dankeschön geht an meine Oma Lidija Klimova, die mir die Welt der Mathematik eröffnet hat und an meine Englischlehrerin Ludmila Rinskaya, ohne die schreiben dieser Arbeit nicht möglich gewesen wäre.

Nicht zulezt gebührt ein ganz spezieller Dank meinen lieben Frau Mariia Halushchak und Sohn Daniel Gruzdev für die bedingungslose Unterstützung während meines Masterstudiums. Sie waren immer für mich da und die ganzen Jahren in vielerlei Hinsicht eine große Stütze.

Acknowledgements

At this point I would like to express my special gratitude to the following people, without whose help the preparation of this master's thesis would never have come about.

Special thanks go to my mother Alla Hruzdeva, who has always given me confidence not to give up even in difficult times and to consistently pursue my goal.

I would also like to thank Ao.Univ.Pof. Dipl.-Ing. Dr.techn. Franz Puntigam for supervising my master thesis and his understanding for all my questions. He always guided me on the right way with the right assistance. This enabled me to concentrate better on the central issues of the thesis.

A big thank you goes to my grandma Lidija Klimova who opened the world of mathematics to me; and to my English teacher Ludmila Rinskaya, without whom writing this thesis would not have been possible.

Last but not least, a very special thanks to my dear wife Mariia Halushchak and son Daniel Gruzdev for their unconditional support during my master studies. They have always been there for me and have been a great support in many ways throughout the years.

Kurzfassung

Gleichzeitig stattfindende Aktivitäten begleiten unser tägliches Leben und sind für den Menschen selbstverständlich. In der Programmierung werden solche Aktivitäten durch nebenläufige Aufgaben ausgedrückt. Zwei gängige Ansätze für den Umgang mit Gleichzeitigkeit in Java sind die Java-Threads (Plattform-Threads und virtuelle Threads) und das modernere, reaktive Programmierparadigma, das in dieser Arbeit auf der Grundlage von RxJava analysiert wird. Diese Arbeit bietet einen umfassenden Vergleich von Threads und RxJava, wobei der Schwerpunkt darauf liegt, zu verstehen, warum die reaktive Programmierung für die Bewältigung von Nebenläufigkeitsproblemen gut geeignet ist und wie sie sich grundlegend von Thread-basierten Ansätzen unterscheidet. Außerdem wird untersucht, wie sich Leistung und Speicherverbrauch zwischen den beiden Paradigmen unterscheiden. Darüber hinaus wird in dieser Arbeit der Entwicklungsaufwand untersucht, der für die Implementierung vergleichbarer Szenarien mit den beiden Ansätzen erforderlich ist.

Die Methodik folgt einem strukturierten Ansatz, der mit einer umfassenden Literaturrecherche beginnt, um eine theoretische Grundlage zu schaffen. Basierend auf dieser Grundlage wird ein realistisches Szenario entworfen. Dieses Szenario wird dann sowohl mit Java Threads als auch mit RxJava implementiert. Zum Vergleich werden die wichtigsten Bewertungskennzahlen für Leistung, Speicherverbrauch und Entwicklungskomplexität definiert und gemessen. Die gesammelten Daten werden analysiert, um die Stärken und Grenzen jedes Ansatzes zu ermitteln, und es werden iterative Verfeinerungen des Szenarios vorgenommen, um die Gültigkeit und Relevanz der Ergebnisse zu verbessern.

Die Ergebnisse zeigen, dass RxJava für die Verwaltung der Gleichzeitigkeit sehr effektiv ist, insbesondere bei I/O-gebundenen Aufgaben, bei denen die nicht blockierende Natur und die Wiederverwendung interner Threads eine bessere Zeitleistung bieten. Außerdem weist RxJava aufgrund seines ereignisgesteuerten Gleichzeitigkeitsmodells einen geringeren Speicherverbrauch auf. Die steilere Lernkurve der reaktiven Programmierung erfordert jedoch zusätzliche Entwicklerfähigkeiten.

Virtuelle Java-Threads verbessern die zeitliche Leistung von Plattform-Threads insbesondere bei I/O-gebundenen Aufgaben. Virtuelle Threads verbrauchen viel weniger Speicher als Plattform-Threads und können daher eine gute Alternative zu RxJava sein, insbesondere für konkurrierende Szenarien mit einer geringen Anzahl von konkurrierenden Aufgaben und einer großen Anzahl von blockierenden Aufrufen. Allerdings haben virtuelle Threads im Vergleich zu Plattform-Threads keine signifikante Verbesserung der

zeitlichen Leistung bei der Bearbeitung von CPU-gebundenen Aufgaben. Die Messungen zeigen vergleichbare Ergebnisse wie die für die RxJava-Implementierung durchgeführten Messungen.
Diese Untersuchung kommt zu dem Schluss, dass Threads zwar nach wie vor für CPU-gebundene Aufgaben und Szenarien, die eine präzise Steuerung erfordern, von Vorteil sind, RxJava jedoch erhebliche Vorteile in Bezug auf Abstraktion, Ressourceneffizienz und die einfache Verwaltung asynchroner Aufgaben bietet, was es zu einem wertvollen Werkzeug für die moderne Softwareentwicklung macht.

Abstract

Simultaneous activities accompany our everyday lives and are natural for humans. In programming such activities are expressed via concurrent tasks. Two common approaches for handling concurrency in Java are Java threads (platform threads and virtual threads) and the more modern, reactive programming paradigm analysed in this thesis on the basis of RxJava. This thesis provides a comprehensive comparison of Threads and RxJava, focusing on understanding why reactive programming is well-suited for addressing concurrent problems and how it differs fundamentally from thread-based approach. We also investigate how performance and memory usage vary between the two paradigms. Additionally, we explore the development effort required to implement comparable scenarios using each approach.

The methodology follows a structured approach, beginning with a comprehensive literature review to establish theoretical foundations. Based on these foundations, a realistic scenario is designed. This scenario is then implemented using both Java Threads (platform threads and virtual threads) and RxJava. For comparison, key evaluation metrics for performance and memory consumption are defined and measured. The collected data is analyzed to identify the strengths and limitations of each approach, and iterative refinements to the scenario are performed to improve the validity and relevance of the findings.

Results indicate that RxJava is highly effective for managing concurrency, particularly in I/O-bound tasks, where its non-blocking nature and reuse of internal threads provide superior time performance. Additionally, RxJava demonstrates lower memory consumption due to its event driven concurrency model. However, the steeper learning curve of reactive programming and rather complicated analysability of code require additional developer proficiency.

Java virtual threads improve the time performance of platform threads especially for I/O-bound tasks. Virtual threads use much less memory than platform threads and for this reason they can be a good alternative of RxJava especially for concurrent scenarios with a low number of concurrent tasks and a big number of blocking calls. However, virtual threads do not have significant improvement of time performance while dealing with CPU-bound tasks comparing to platform threads. The measurements show comparable results with the measurements conducted for the RxJava implementation.

This research concludes that while threads are still advantageous for CPU-bound tasks and scenarios requiring precise control, RxJava provides significant benefits in terms of

abstraction, resource efficiency, and ease of managing asynchronous tasks, making it a valuable tool for modern software development.

Contents

CHAPTER 1

Introduction

“Begin at the beginning,”, - the King said gravely, - “and go on till you come to the end: then stop.”

— Lewis Carroll, *Alice in Wonderland*

Today’s modern world is composed of many simultaneous activities. This activities can block each other from performing their tasks and introduce waiting times for completing tasks and finishing activities. In programming, simultaneous activities are usually expressed as concurrent processes. Numerous models and techniques for concurrent process realisation in various programming languages have been proposed during the last decades. Today the topic of concurrency in programming is still actual, because modern web applications make a huge amount of blocking calls to and from the database in the back end, which can potentially influence performance and consequently popularity of an application. To stay competitive and to achieve best performance and usability of an application, developers must keep in mind concurrency, they have to think about some technique which will be able to process parallel requests quick enough and without any loss of information.

In this work the comparison of 2 different "ideas", how concurrency can be realised in the Java programming language, will be presented.

The first approach is a well known *java threading abstraction*. Threads are sometimes referred to as so-called *lightweight processes*. Like processes they are independent, have their own stack, and their own program counter. However, threads within a process are less insulated from each other than separate processes are, for instance, they can share memory or file handlers. There are many reasons to use threads in Java programs. With threads programmers and users can benefit from the advantages of multiprocessor systems, make and use more responsive user interfaces (UI), perform asynchronous and background calculations, etc.

Using threads can also be risky. When multiple threads access the same data item, there

is a need to coordinate the read/write procedures appropriately to ensure a consistent view of the data. Such kind of coordination is called *synchronization*. Having a very big number of threads and one central data item, which each thread tries to access, it can quickly lead to synchronization problems between threads, deadlocks and general decrease of performance [53].
Reactive programming can be observed as an alternative to threading. The paradigm introduces asynchronism as the core concept, which is actually an opposite to threading, where "everything happens in one thread". It is capable to handle data streams which are represented as a collection of coherent and cohesive digital signals created on a continual or nearly-continual basis. These data streams are sent from the source as a reaction to some triggers like events, calls, messages, etc. A reactive program implements observers which are able to recognize changes and to consume data from a stream and perform necessary tasks to handle these changes [78].
A big advantage of the reactive programming approach is that the code to be executed and the thread are decoupled, thus there are less expensive context switches on operating system level. Due to asynchronous request processing, reactive programming can also have advantages over threads while accessing a single critical resource like a single database or file. The use of a big number of threads can lead to synchronization problems in this case.
Reactive programming, however, has also noticeable disadvantages like an increased level of difficulty in reading, writing and debugging the code, consequently a steeper learning curve by developers. Writing of tests and debugging of code can also be difficult.

1.1 Problem Statement

This study aims to compare Java threads with the modern reactive programming approach (implemented with RxJava) to identify the more effective method for addressing contemporary concurrency challenges within the same concurrent scenario.
Concurrent tasks can be executed independently, meaning they do not require communication or rely on each other. Such an environment, where no interaction between concurrent tasks occurs, is referred to as *stateless*. In contrast, tasks may share memory, or the execution of one task may depend on the completion of another task. Such type of environment, where tasks are interdependent, is known as *stateful*. Performance and memory consumption can vary significantly between these environments, making them an intriguing subject for investigation. We distinguish between different forms of concurrency organisation:

- tasks in a stateless environment;
- tasks in a stateful environment.

Tasks themselves can differ in their purpose. Some tasks can be more input/output (I/O) centred, others can use more computational power and less I/O operations. Concurrent tasks in stateless and stateful environments can have different performance dependent on the nature of tasks [59]. In this thesis we will distinguish the following kinds of tasks:

- I/O bound tasks;
- CPU-bound tasks;
- mixed (I/O and CPU-bound) workloads.

While this thesis was being written, JDK 19 reached the general availability (September 20, 2022). This Java version introduced virtual threads for the first time. The concept was further refined in JDK 20 (March 21, 2023) and finalized in JDK 21 (September 19, 2023) [55]. Virtual threads became a notable case for comparison in this thesis. We distinguish between the following implementations of Java threads:

- platform threads[1];
- virtual threads.

Specifically, following research questions will be asked:

RQ1. How and why does reactive programming qualify itself for treating concurrent problems in programming? What are major differences to traditional threads?

RQ2. How does the performance differ between threads and reactive programming approaches in different forms of concurrency organization?

RQ3. How does memory consumption differ between threads and reactive programming approaches in different forms of concurrency organization?

RQ4. How does the developing effort of the same scenario differ while developing the solution using threads and using reactive programming?

1.2 Methodological Approach

This research will be conducted using following methodologies:

1. Literature review with an aim to identify and understand core ideas of both concepts (threads and reactive programming);
2. Define a non-trivial scenario of a concurrent application where advantages and disadvantages of both concepts can be shown;

[1]In this thesis we will often call platform threads just *threads* or *traditional threads*. When referring to virtual threads, we always specify them explicitly.

3. Develop two implementations of this scenario, one using threads and another using reactive programming;
4. Define important indicators for the comparison of two approaches;
5. Compare and evaluate two implementations of the same scenario based on indicators defined earlier and provide the results in an understandable form;
6. Analyse evaluation results and improve the scenario and/or the implementation to achieve better or more precise results (repeat item 5 in case of an improved implementation);
7. Conclude with conclusive remarks and propose ideas for further work;

1.3 Structure of the Thesis

This thesis has the following structure:

- Chapter 2 describes the notion of *concurrency* in general with the focus on concerns relevant for our further discussion.
- Chapter 3 presents *threads* and provides all necessary information about them.
- Chapter 4 acquaints *reactive programming* and discusses the toolset it provides for solving concurrency problems.
- Chapter 5 concerns the implementation of an concurrent scenario using threads and reactive programming.
- Chapter 6 evaluates both programming techniques based on implementations of the same scenario.
- Chapter 7 provides conclusion and ideas for future work.

CHAPTER 2

Concurrent Computations

For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers.

— Gene Amdahl, 1967.

Every computation is based on the execution of some atomic instructions. We call a computation *sequential* if the instructions are executed one after another (in a sequential manner). Multiple sequential computations executing in overlapping time periods on a single or on multiple processing units are called *concurrent*. In other words, *Concurrency* in computer programming is in general a simultaneous execution of two or more processes[1] with no specific order. As a result, concurrency is *non-deterministic*, meaning that repeated invocation of the same concurrent program with the same inputs can result in different outputs [3].

In this chapter, we cover the foundations of concurrency, discuss correctness properties of concurrency, and explain the distinction to parallel and distributed computing.

2.1 Foundations of Concurrency

During the last 50 years, the hardware industry has experienced a big revolution. Processor manufacturers have constantly delivered faster single-core CPUs, and in 2005 they finally

[1]In general concurrency theory uses the term *process* while talking about execution of abstract computations. However, the term *process* will interfere with another meaning of this term, which will be presented in the next chapter of this thesis. In oder to avoid misunderstanding, we will denote abstract computations as *tasks* throughout the remainder of the thesis.

reached the limit of CPU speed predicted by Gordon Moore[2]. Moore predicted in 1965 that the density and speed of transistors would double every 18 months before reaching a maximum speed beyond which technology could not advance. This prediction is widely known as Moore's Law. Now, decades past his estimate, we can only confirm that he was right. Modern single-core CPU can already perform computations with a speed nearly equal to the speed of light, regardless of the energy dissipation, which produces enormous heat. This heat is a limiting factor for further improvements [76].
The trend of producing faster CPUs has matched with desires of developers and users, who have always sought to increase the speed and the complexity of their applications. The new tendency in multicore processor design has brought concurrent computations in game. Multicore processors offer a possibility to develop more efficient software, but this requires more effort and some mind change by developers, who want their programs to maximize hardware utilization [76]. Operating systems (OS) have also evolved to allow multiple tasks to run simultaneously. For OS developers there were several motivating factors for creating such systems (defined by [24]):

- **Resource utilization.** Tasks sometimes wait for some external events like input from users, or output from other systems, and can perform other useful work while waiting. Such motivation is natural to human everyday life, for instance, it is possible for humans to read a newspaper while waiting for a bus on a bus stop. In this example, the main task of a human is to board a bus and to travel from the starting point to the destination. If the person arrives to the bus stop some minutes earlier, he/she can in the meantime read some newspaper (performing another task), and then as soon as the bus comes, stop reading and board the bus (continue with the main task).

- **Fairness.** Multiple tasks may have equal claims on accessing some machine resource. In this case, some time slicing can be applied for task executions. Such time slicing is called *tasks scheduling*. Scheduling is the process of allocating system resources, such as CPU time, to multiple tasks or threads in an efficient manner. It determines the order and timing in which tasks are executed, balancing priorities, resource availability, and dependencies. Schedulers use algorithms like First-Come-First-Served (FCFS), Round Robin, or Priority Scheduling to manage tasks, ensuring fairness and optimal performance. In concurrent systems, task scheduling can involve preemptive techniques, where tasks are interrupted and resumed, or cooperative techniques, where tasks yield control voluntarily. Effective scheduling is critical to achieve high throughput, low latency, and efficient resource utilization in multi-tasking environments.

- **Convenience.** For developers, it is often more convenient to divide a complex program into several smaller, independent programs, each responsible for perform-

[2]**Gordon Moore** (January 3, 1929, San Francisco, California, USA - March 24, 2023, Hawaii, USA) is an American engineer, businessman and co-founder of Intel Corporation (the world's largest semiconductor chip corporation).

ing a single, relatively simple computation. These smaller programs can then communicate and coordinate with one another as needed to collectively achieve the functionality of a larger, more comprehensive system. This approach simplifies the development process by breaking down the overall problem into manageable pieces, allowing developers to focus on solving one part at a time. Additionally, this modular design improves code readability, maintainability, and reusability, as individual components can be developed, tested, and debugged independently. By orchestrating these smaller components to work together, developers can efficiently create robust and scalable systems.

Leslie Lamport[3] in his Turing Award[4] speech [38] mentioned that it is not reliably known when the concurrency computation science started, but according to him, it began with Edsger Dijkstra's[5] seminar paper he published in 1965 about the so-called *mutual exclusion* problem. He posed the following problem: there are N processes running simultaneously and each process in some period of time has to access some *critical section* (shared resource), so that the properties below are satisfied:

- **Mutual Exclusion** – is the property of a task to never enter the critical section, while another task is already accessing it.
- **Livelock Freedom** – if a task n is waiting to execute its critical section, then it is guaranteed by design, that this task will eventually execute its critical section.

Nearly 60 years have passed after Dijkstra's problem formulation and many effort of diverse computer scientists was invested into solving this problem. The first solution was the so-called bakery algorithm, which was based on the idea of retail shops, where customers take successive numbers and then the waiting customer with the lowest number is served next [38] [39]. Today, after the 6 decades, the interest to concurrency topics has not expired and is even larger than ever before. Concurrent task execution is now used almost for every web page, every modern PC game, etc. Problems, which were defined by Dijkstra are today known as base properties of concurrency under different names, namely mutual exclusion is today called a *safety* property, and livelock freedom is called

[3]**Leslie Lamport** (born on February 7, 1941, New York, USA) is an American computer scientist and mathematician, who in 2013 won a **Turing Award** for imposing clear, well-defined coherence on the seemingly chaotic behaviour of distributed computing systems, in which several autonomous computers communicate with each other by passing messages. He made significant achievements in improvement of correctness, performance, and reliability of computer systems.

[4]**Turing Award** is an annual prize given by the *Association for Computing Machinery* (ACM) for contributions of lasting and major technical importance to computer science. It is often referred to as the "Nobel Prize of Informatics". The award is named after the British mathematician **Alan Turing** (June 23, 1912, London, UK – June 7, 1954, Wilmslow, Cheshire, UK).

[5]**Edsger Dijkstra** (May 11, 1930, Rotterdam, Netherlands - August 6, 2002, Nuenen, Netherlands) was a Dutch computer scientist pioneer, one of the most influential figures in computer science, who shaped the discipline both, as a theorist, and as an engineer. His fundamental contributions cover diverse areas of computing science, including compiler construction, operating systems, distributed systems, etc.

a *liveness* property of concurrent computations. Both properties were defined formally in 1985 and are today known as correctness properties of concurrency [38].
In the next section the correctness properties of concurrency will be discussed.

2.2 Correctness Properties of Concurrency

As defined earlier, each task represents some abstract computation, and in its turn every computation is a list of instructions which can be executed. On a fundamental level tasks serve the needs of one or more users. Internally task execution can exhibit a complicated, non-deterministic and concurrent behaviour. However, for the users only the external communication (like printing of results) and external visible reactions are relevant. In particular, such reactions should occur within a finite amount of time. Formally, each task has to satisfy the specification defined by users and terminate in a finite amount of time [54].
For tasks which are executed concurrently two properties are relevant for correctness ([35], [24], [11]):

- **Safety** asserts that nothing "bad" will ever happen during the execution (in other words, that tasks will never enter an unacceptable state).
- **Liveness** asserts that something "good" will eventually happen during the execution (in other words, operations that must be done, will be done eventually in the future).

A safety property is an invariant and holds in every state of every execution. In contrast, a liveness property must hold in some state of every execution. As a result, safety and liveness have the so-called duality relationship – if some property does not hold, the other will be also not be satisfied [35].
Safety relates to *partial correctness* (the result is correct if the program terminates), and liveness is the properties of *total correctness* (the program terminates with the correct result) [11].
Certainly, there exist situations, where safety and/or liveness properties are violated, such situations are called *hazards*. We will discuss some common hazards in the next chapter of this thesis.
In the next section the motivation for concurrency in general and specific in this thesis will be discussed.

2.3 Motivation for Concurrency

Concurrency is essential for contemporary applications, particularly those requiring real-time data processing, intensive computational tasks, or the maintenance of high responsiveness and system interactivity [24]. In the process of modelling real-world systems, it is necessary to compose and coordinate these concurrent executions effectively. Following examples depict concurrent systems used in every day life: railway networks,

banking systems, travel reservation systems, multiplayer games, and even the human brain (as shown in Minsky's[6] "Society of Mind" [47]).
Outlined below are key factors underscoring the significance of concurrency (inspired by [7] [74] [22] [24]):

- **Improved application performance**
  A primary advantage of concurrency lies in its capacity to enhance application performance by leveraging multiple threads to execute tasks concurrently. This parallel execution allows more efficient utilization of computational resources, significantly reducing execution time compared to single-threaded applications, which process tasks sequentially.

- **Optimal CPU Utilization**
  Modern processors are typically equipped with multiple cores, and concurrency enables efficient utilization of these cores. Standard non-concurrent applications are restricted to operating on a single core, leaving the remaining cores underutilized. In contrast, concurrency allows each core to execute distinct tasks concurrently, thereby optimizing CPU usage and improving overall computational efficiency.

- **Better Resource Sharing**
  Concurrency facilitates efficient resource utilization by enabling simultaneous access to shared resources. In large-scale applications, resources such as memory, file handles, and network connections are often accessed by multiple users or processes. By allowing threads to operate on shared resources simultaneously, concurrency minimizes latency and alleviates potential bottlenecks, thereby enhancing overall system performance and resource efficiency.
  However, it is crucial to appropriately manage and orchestrate the simultaneous access of multiple tasks to shared resources. A variety of techniques are available to facilitate such coordination effectively.

- **Handling Multiple Tasks Simultaneously**
  Concurrency enables an application to manage multiple tasks. In the context of web servers, for instance, processing thousands of client requests would be infeasible without concurrency. By assigning each client request to an individual thread, the server's capacity to handle multiple users simultaneously is significantly enhanced, thereby improving overall concurrency and performance.

- **Increased Responsiveness in User Interfaces (UI)**
  In user-facing applications, maintaining responsiveness is critical for delivering a seamless and engaging user experience. Consider a scenario where a desktop application executes a computationally intensive task, such as downloading a file,

[6]**Marvin Lee Minsky** (August 9, 1927, New York City, USA - Januar 24, 2016, Boston, Massachusetts, USA) was an American cognitive and computer scientist concerned primarily with research in artificial intelligence (AI). In 1969 he received the Turing Award.

on the main thread. During this operation, the application would cease to respond, rendering the user interface unresponsive and causing potential frustration for users. By leveraging concurrency, tasks such as file downloads, database operations, or computationally intensive calculations can be delegated to background threads. This approach ensures that the user interface remains responsive, thereby enhancing overall user satisfaction and interaction quality.

In this thesis we particularly focus on the comparison of Java threads and RxJava approaches within the context of the same concurrent scenario. This scenario will involve the execution of multiple simultaneous users which will communicate between each other, and the utilization of shared memory (database) for the storage of data. The information about the users and all the messages each user will send or edit will be saved in the database.

Java allows developers to explicitly manage the execution of concurrent tasks through the use of multiple threads, which can be executed concurrently on different processor cores. Simultaneous tasks are typically handled by creating multiple threads that run independently of one another but can use shared memory in order to save or retrieve data. In Java, threads can be instantiated dynamically as required, or alternatively, they can be managed with a thread pool, allowing the reuse of threads across multiple tasks [7]. In this thesis we will consider *only threads dynamically created on request*. Each thread in the system will correspond to an individual user. A thread will be instantiated upon the user's entry into the system and terminated concurrently with the user's departure from the system. According to [24] the thread pool usage is not recommended if there are stable identities associated with each thread, or each thread is dedicated to its own task.

RxJava provides a more abstracted, declarative approach to concurrency using the reactive programming model. RxJava employs *Schedulers* to specify the threads on which operations should be executed. Tasks can be executed concurrently on different threads, while RxJava automatically manages the synchronization of the task's execution, reducing the need for explicit locking or synchronization mechanisms [52].

Both threads and RxJava manage shared memory access, but through different approaches. Threads can utilize either blocking or non-blocking synchronization techniques. Blocking mechanisms, such as semaphores or traffic lights, regulate access to shared resources by coordinating the sequential passage of competing entities. In contrast, non-blocking techniques leverage advanced algorithms to ensure the safe concurrent traversal of shared resources by multiple participants, thereby preventing conflicts or accidents. However, non-blocking methods often involve trade-offs, such as decreased performance and increased complexity in maintenance and implementation. This thesis primarily *focuses on blocking synchronization mechanisms in threads* due to their simplicity and widespread support. These mechanisms are deeply integrated into Java's concurrency utilities and have been extensively optimized across various platforms and JVM implementations, ensuring robustness, efficiency, and reliability. Blocking synchronization is traditionally employed for orchestrating concurrent access to shared resources [24].

In contrast, RxJava capitalizes on asynchronous processing to manage shared memory

access, thereby avoiding the explicit blocking of tasks [52].
Speaking about concurrency there is often a confusion, because the dictionary meaning of concurrency is almost the same as another term, - *"parallel"*. So, Merriam Webster Dictionary[7] defines "concurrency" as "happening, or existing, or done at the same time"; and "parallel" is defined as "very similar and often happening in the same time". However, in computer science there is a significant difference. The meaning of "distribution" and "concurrency" is also sometimes confusing. In the next section the distinction between "concurrency", "parallelism" and "distribution" will be discussed.

2.4 Concurrency, Parallelism and Distribution

This section aims to establish a common understanding about the terms related to concurrency and simultaneous execution of tasks. We assume that each task consists of some finite number of atomar subtasks. If all subtasks of a certain task are executed one after another, we talk about *sequential* subtask execution.
The term *concurrency* is used to refer to processing more than one task at the same time on one or several processors. In general no assumptions about the quantity of processors is made. If there is only one CPU, it is clearly impossible to execute subtasks of 2 or more tasks at exactly the same time. To achieve concurrency on the CPU, for example, time slicing can be used. Thus, if we talk about concurrent executions of 2 or more tasks, we assume that it does not completely finish one task before it begins the next one [76].
Parallelism is the concept of executing multiple subtasks literally at the same time on different CPUs. Strictly speaking it does not require the existence of 2 or more tasks. It is capable to run subtasks simultaneously with use of multi-core infrastructure by assigning one core to each subtask. Thus, parallelism depends on the actual runtime environment, and it requires hardware support (for instance, multiple CPUs). The main goal of parallelism is to increase performance and throughput of a computer program [76].
The difference between *concurrency* and *parallelism* is a consequence of the context in which some tasks are executed. Dependent on the execution environment tasks may be run in both a sequential environment that employs time slicing and in a parallel environment with multiple processing units available. The notion of concurrency captures both potential execution environments, while parallelism is more specific and requires an environment with multiple processors [72].
To better uderstand the difference between concurrent and parallel execution of tasks observe Figure 2.1 (inspired by [72]). Here on each line a complete lifecycle of tasks is illustrated. Each task is divided in subtasks represented by white boxes. Each task is considered to be completely done if and only if each subtask of this task is executed. First observe the upper 3 task (*t1, t2, t3*) executions. Here we see concurrent, non-parallel execution: we begin with a task *t1* and execute its first subtask, before executing the

[7]**Merriam Webster Dictionary** is a English language dictionary edited in 19th century by American lexicographer **Noah Webster** (October 16, 1758, Hartford, Connecticut, USA - May 28, 1843 New Haven, Connecticut, USA)

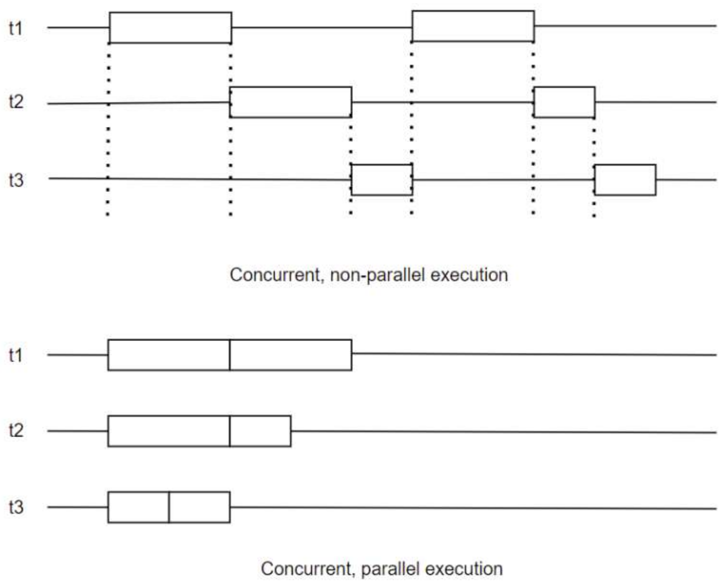
t1
t2
t3
Concurrent, non-parallel execution
t1
t2
t3
Concurrent, parallel execution

Figure 2.1: Comparisson of Concurrent Non-Parallel and Parallel Executions.

second subtask of *t1* we give a time slice to task *t2* and execute its first subtask, and so on until all 3 tasks are completely executed. The system has started the execution of tasks *t2* and *t3* without finishing the execution of task *t1* beforehand. Now observe the lower 3 task (*t1, t2, t3*) executions. Here we see concurrent, parallel execution: all three tasks are executed truly simultaneously on three different cores.
Distributed computation is another term which is often confused with concurrency and parallelism. However, in literature and in science distributed computing is observed as separate from the concurrent computation discipline. Nevertheless, both concepts are built on the same fundamental idea: executing tasks at the same time [2]. Distributed computation is a technique of executing the components of the complex software system shared among multiple devices (like computers or servers) often called *nodes*. All the nodes are able to communicate to each other via a network and coordinate their actions by passing messages to one another. Each single node can itself have one ore more processors and can be able to execute tasks concurrently [75].
Consequently, we can make fundamental observations about the interrelations of concurrency, parallelism and distribution (based on [7]):

1. Concurrent systems can be parallel and distributed.

2. Parallel and distributed systems are inherently concurrent.

3. Distributed systems with two or more nodes are parallel.

In this chapter we briefly discussed the history of concurrent computations, defined fundamental correctness properties of concurrency and showed the distinction between concurrent, parallel and distributed computations using the abstract notion of task. The next chapter will be devoted to *threads*. In order to explain threads we will need to define more specific computation units like programs and processes. We will also discuss benefits and risks of threads, define thread hazards and describe mechanisms how threads can be synchronized in order to avoid this hazards.

CHAPTER 3

Threads

Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end.
— Herb Sutter.

If you think it's simple, then you have misunderstood the problem.
— Bjarne Stroustrup.

Sequential execution of tasks appears intuitive and natural for developers and end users. However as stated in the previous chapter, there are several motivating factors, which cause rapid development of concurrent computations. The same concerns have also encouraged the development of *threads*. Nonetheless, to explain threads we can no longer be on abstract computation levels and need to dive deeper and define specific concepts like programs and processes.

3.1 Programs, Processes, and Threads

Throughout this thesis we will call any sequence of instructions that a computer can interpret and execute a *computer program* or just a *program*. A program written in some programming language in a human-readable form is called *source code*. Source code needs another program to translate human-readable code into machine instructions with an aim to convert it into an executable instance. A set of computer programs, source codes, specifications, documentations and other data is called *software*. For instance, Google's Chrome browser chrome.exe is a program (represented as executable file) which stores all instructions and when executed allows users to open and use the browser. The source code of the browser is a file which stores all the instructions in some programming

language. The Google Chrome browser software product includes the program itself, its source code, related data and documentation like usage manuals, licence, etc.
From the definitions above two important conclusions about programs can be deduced:

1. A computer program is a passive, static object. It only stores the list of instructions which can be executed.
2. Computer programs are stored on the disk or secondary memory.

A *process* is an execution of a specific program. In contrast to a program, a process is an active instance that acts according the purpose of the software. Multiple processes can be related to the same program. For example, Google's Chrome browser can be started twice, and it will result in two separately running instances of the browser. By starting the browser for a second time, we create the second process which relates to the same program.
In general, each process consists of the following resources (based on [70], [77]):

- An image of the executable machine code program translated from the source code.
- Memory which holds the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.
- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows terminology), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers and physical memory addressing. The state is typically stored in computer registers when the process is executing, and in memory otherwise.

Operating systems (OS) usually hold all this information in special data structures called *process control blocks*. When a new process has to be created, the OS builds all execution data structures and allocates address space in main memory needed for the process. In general four common events can lead to the creation of a new process. In a batch environment, a process is created as a response to the submission of a job. In an interactive environment, a process is created as a reaction to a user's log on. In both cases from above, the OS is responsible for the creation of processes. An OS may also create a process on request of the application or user. For example, if a user requests some file to be printed, the OS can create a process that will manage the printing. The process can also be spawned by another process. For example, a server process can generate a new process for each request that it handles. When one process spawns another, the former is

called *parent process* and the spawned one is referred to as *child process* [74].
During execution the process holds all needed resources and is isolated (means it does not share resources with other processes). Termination of a process can have many different reasons including: normal completion, I/O failure, termination by user or parent process request, time overrun, etc [74].
Threads are sometimes called *lightweight processes*, and most modern operating systems treat threads, and not processes, as the basic unit for scheduling [24]. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process [74]. Threads exist within a process, and every process has at least one thread of execution. Threads share the resources of the process, including memory and open files. Since threads share the memory address space of their owning processes, they consequently have access to the same variables and allocate objects from the same heap [24].
Multithreading is the ability of an OS to support multiple, concurrent paths of execution within a single process [74].

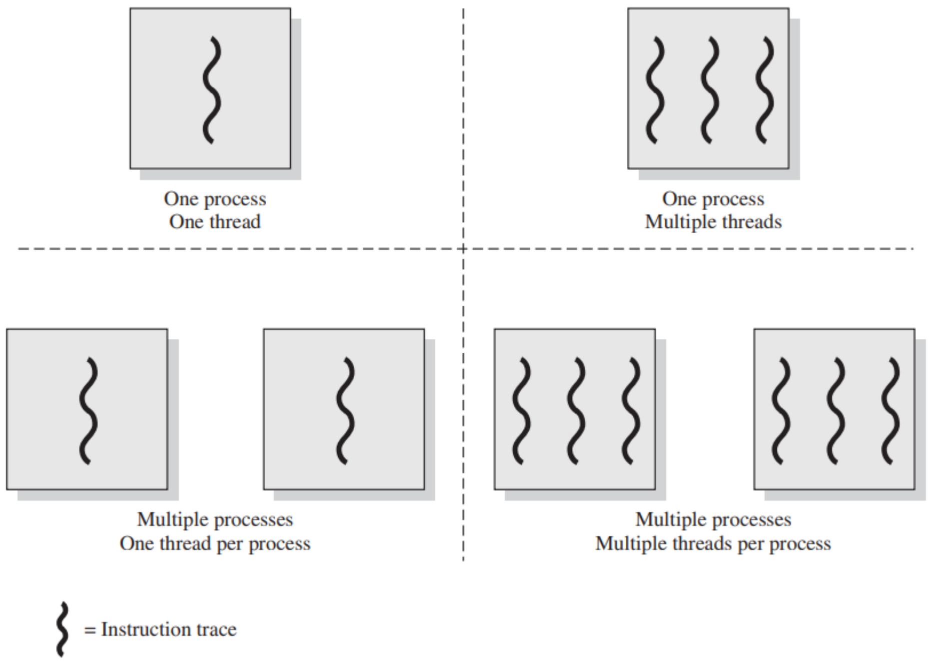

Figure 3.1: Processes and Threads [74]

Figure 3.1 shows different multithreaded approaches. The two arrangements shown on the left side of Figure 3.1 illustrate the traditional approach of a single thread of execution per process. MS-DOS is a perfect example of an OS that supports a single user process and a single thread within that process. The right half of the Figure 3.1 shows multithreading environments, where many threads can coexist within one single process. A Java runtime environment is an example of a multithreaded system.

The Java Virtual Machine (JVM) startup as a rule always includes the creation of a thread, consequently every Java program has at least one thread. The JVM continues to execute threads until either of two following events occurs: (1) the *exit()* method of class *Runtime* has been called and the security check has allowed the exit operation to take place; (2) termination of thread run by returning from the call of the *run* method or by throwing an exception that propagates beyond the run method [16]. In Java each thread at any point in time can exist in any of the following states (based on [17]):

1. **New:** when a new thread is created, the *New* state is automatically assigned to it. In this state the thread is not running.
2. **Runnable:** a thread in this state can either be running, or waiting for run, dependent on the scheduler, which assigns the time to run to a thread.
3. **Blocked:** a thread in this state is waiting for a monitor lock to enter or reenter a critical section.
4. **Waiting:** a thread in this state is waiting for another thread to perform a particular action.
5. **Timed waiting:** a thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
6. **Terminated:** a thread that finished performing its task.

As mentioned above, within a process there can coexist several threads, each associated with the following information (based on [74]):

- an execution state (New, Runnable, etc.);
- a saved thread context when not running;
- an execution stack;
- some pre-thread static storage for local variables;
- access to the memory and resources of its process, shared with all other threads in that process.

A consequence from the last statement is that threads reside in the same address space and have access to the same data as a ”parent” process. When one thread changes an item of data in memory, other threads see the results if they access that item after the change is made. If one thread opens a file with read privileges, other threads in the same process can also read from that file simultaneously [74].
Unfortunately, the fact that threads share the memory and resources of its process brings also many potential problems and challenges for developers. One thread can modify

variables that another thread is in the middle of using, which can produce unpredictable results with each new run of a multihreaded program. Such bugs are very hard to debug and solve for developers. To avoid simultaneous write access to shared data, explicit *synchronization* of threads must be made [24]. In the next section some typical risks and hazards concerning thread synchronization will be discussed in more detail.
Nevertheless, usage of threads has major benefits for programs and some drawbacks. In the next section, both, advantages and disadvantages of using threads will be presented.

3.2 Benefits and Drawbacks of Using Threads

Multithreading is a powerful tool for enhancing the performance of programs. In literature authors often denote the following **benefits** of using threads (inspired by [24] [74] [53] [76]):

- **Exploiting multiple processors.**
  Programs with multiple active threads can execute simultaneously on multiple processors. When properly designed, multithreaded programs can improve throughput by utilizing available processor resources more efficiently. As consequence, the performance of the program can increase.

- **Performance gain for single-processor systems.**
  Assume the situation when a single-threaded program remains idle while waiting for some input from the user. The I/O operation completely blocks thread execution which can only be continued after the operation is complete. In a multithreaded environment, another thread can still run, while the first thread is waiting for the I/O operation completion. In such a way multithreading allows the software to still make progress during blocking calls.

- **Simplification of servlets and Remote Method Invocation (RMI).**
  A complex, asynchronous workflow can be decomposed into a number of simpler, synchronous workflows each running in a separate thread. The program can be designed in such a way, that the interaction between those threads can happen in some defined check points. This benefit is used by frameworks like servlets or RMI. "Servlet writers do not need to worry about how many other requests are being processed at the same time or whether the socket input and output streams block; when a servlets service method is called in response to a web request, it can process the request synchronously as if it were a single-threaded program. This can simplify component development and reduce the learning curve for using such framework." [24]

- **Simplification of asynchronous event handling.**
  If a single-thread application goes to read from a socket some data which is not available, it will block an execution of a whole application and all subsequent requests till data appears in the socket. To avoid this problem, single-threaded

applications need to use the so-called non-blocking I/O, which is more complicated and error prone than synchronous I/O. Another solution will be to utilize multiple threads and assign an own thread to each request. In this case, blocking of one thread will not affect processing of other requests [24].

- **Creating more responsive User Interfaces (UI).**
  Modern GUI Frameworks (for instance, AWT; Swing; etc.) utilize the so-called *event dispatch threads* (EDT). An event thread is considered to be a main thread which calls the application-defined event handlers, as a response to some event like, for example, a button click. An event thread runs in its own thread under the control of the GUI toolkit rather than the application. It brings a big advantage to the response time of the UI, because the long-running tasks are executed in separate threads, and the event thread remains free for processing subsequent UI events.

Among **drawbacks** for developers the following is often mentioned:

- **Difficulty of debugging.**
  It is much harder to replicate an error in a multithreaded environment than it is to do so in a single-threaded. While using multiple threads with wrong synchronization of shared resource access, the result is unpredictable and can change with each run of the program (depending on timing when each thread accesses the shared resource). As a result, it is more difficult to identify the root cause of the problem and to understand why the error occurs.

- **Difficulty of testing.**
  The same thoughts from previous drawback are also valid here. Errors are more difficult to find and more difficult to reproduce.

- **Difficulty of managing concurrency.**
  Proper synchronization of threads can sometimes be very difficult and can introduce new unexpected errors in code which was assumed to work correctly.

- **Difficulty of converting single-threaded code to multithreaded version.**
  To convert the source code of an existing single-threaded program into mutithreaded code, is not a trivial task. Developers will need to remove all static variables and replace all functions and methods which are not thread-safe (correctness property[1]).

Using threads can also bring potential risks which are actually violations of correctness properties. In the next section we will discuss some risks of using threads.

[1]Thread-safety will be discussed in the next section in more detail.

3.3 Risks of Threads

In the previous chapters we discussed correctness properties of concurrent computations. Threads are not an exception and also have safety and liveness correctness properties. Unfortunately, due to difficulties of managing threads, described above, these properties can easily and unexpectedly become subtle. Problems which arise from violations of correctness properties of threads are called *hazards*. Thread hazards can be also observed as risks from the developers point of view, because they can easily lead to errors in software products and in many cases cannot be efficiently and effectively found by tests. In the next two subsections, safety and liveness hazards of threads will be discussed.

3.3.1 Safety Hazards

Observe Listing 3.1. This source code is supposed to generate a sequence of unique integer values, where each value is twice as much as the previous one. Compiled program execution will behave correctly in a single-threaded environment. Unfortunately, in a multithreaded environment its behaviour is totally unpredictable and depends on timing when each thread will access the value of the variable 'number'. It may appear that *getNext()* contains only 1 operation. However, there are at least 3 different operations involved, namely, each thread needs first to read the value of 'number', to multiply the value by 2, and to store the result of the new value to variable 'number'. Since operations in multiple threads can be arbitrarily interleaved at runtime, it is possible for two or more threads to, for example, read the value at the same time, multiply it by 2, and save the new value. The function will then return the same value for multiple calls in different threads.

```java
public class MultiplyByTwo{

   private int number;

   public int getNext(){
      return number *= 2;
   }
}
```

Listing 3.1: Not Thread Safe Sequence Generator

The example above illustrates the very common thread safety hazard called *race condition*. A race condition is a safety hazard, where the result of computation depends on the relative timing of events in the threads. Because threads share the same memory address space and run concurrently, they can access or modify variables that other threads might be using. This situation adds an element of non-sequentiality which can be confusing and difficult to reason about [24].
Race conditions often appear totally unexpectedly in a program that had worked fine for a large period of time in the past. Often they arise after hardware or software updates of

an environment where the program runs. For example, a new version of an operating system may improve the performance of an operation that was earlier relied upon the slower process, and now has to catch up to the faster process. This also explains the term ”race condition”. Threads are racing towards the critical section, and the correctness of the program relies on them finishing the race in the order assumed by developers [72].
Race conditions can be prevented by imposing *synchronization* on the threads that will access the critical section, which causes behaviour changes depending on the order of arrival of the concurrent threads to the critical section. This can be achieved using locks, semaphores, or mutual exclusion around the critical section. In the next chapter we will discuss how synchronization of threads can be handled in Java [72].
Introduction of synchronization can prevent race conditions, but introduce a bunch of potential liveness hazard, that we will discuss in the next subsection.

3.3.2 Liveness Hazards

It is critically important to pay attention to the safety issues when developing multi-threaded programs. As already mentioned in the previous chapter, safety properties guarantee that the result in the end of program execution will be correct. It is clear from the last subsections, that in order to assure the correctness of results, developers need to control the ordering of threads accessing the critical section. This action, in its turn, poses another potential problem.
Liveness assures that 'something good will eventually happen'. A liveness hazard occurs when some thread or some number of threads goes into a state such that it is permanently unable to make some progress toward the completion of the computation task [24].
Deadlock is a special situation in concurrency, where one or several threads are blocked waiting for an event that only the other blocked thread is able to generate. Some authors regard a deadlock as both, a liveness and a safety hazard [9] [74]. Consider the following example: Listing 3.2 shows the source code of a simple bank account program, where the variable ”balance” is a shared resource representing current balance on someone's account. Function ”withdraw” can subtract some amount from the balance, and function ”deposit”, can on the contrary, add some amount to the balance. Function ”transfer” is used to carry the amount of money from one account (”from” function input variable), to another one (”to” function input variable). The ”sync” and ”release” methods in the body of the ”transfer” function are representing locks and releases of the critical section respectively.
Assume the existence of two threads A and B, and two accounts k and p, where in thread A the function *transfer(k, p, 200)* is called, and in thread B, the function *transfer(p, k, 200)* is called, and both functions are called at the same time. In this situation, thread A will lock account k and at the same time thread B will lock account p and then thread A will try to lock account p, which is already locked by thread B; and thread B will try to lock account k, but it is already locked by thread A. In this situation both threads are waiting for release of a critical section, but this release will never happen. This situation is called a deadlock.

```
class Account {
  double balance;

  void withdraw(double amount){
    balance -= amount;
  }

  void deposit(double amount){
    balance += amount;
  }

  void transfer(Account from, Account to, double amount){
    sync(from);
    sync(to);

    from.withdraw(amount);
    to.deposit(amount);

    release(to);
    release(from);
 }
}
```

Listing 3.2: Deadlock Example

There are several recommendations for developers how a deadlock can be avoided (like avoiding of unnecessary and nested locks; using lock orderings, etc.), but none of them guarantees avoidance completely. Another idea how to prevent deadlocks is to allocate all required resources for the thread or a process before the start of its execution. Unfortunately, it leads to another liveness hazard, called **starvation**.
Starvation of a thread is said to occur when the thread does not get access to the resource, because another thread conspire (or appear to be conspiring) to lock it out. For example, many threads with high priority can block some resource for a long time, which is needed for successful run of threads with lower priority.
Livelock is a liveness hazard and a special case of starvation, where two or more threads constantly change their state, with neither thread making a progress. In this situation threads are changing states, and new states are clashing with each other forcing threads to update their states, and updated states are also blocking, forcing threads to return to the previous one, and this can run forever. For instance, livelock can be explained with an analogy of two very polite people passing through a narrow passageway and each tries to step around the other, but they end up swaying from side to side, getting in each other's way as they try to get out of the way. Livelock is different from deadlock in a way that both processes involved in the livelock are repeatedly changing their states with regard to each other, but still not progressing [24] [74] [72].
The solution of a livelock problem is to introduce some randomness into the retry

mechanism of changing state. Retrying with random waits for each livelocked party can be effective for avoiding livelocks in concurrent applications [24].
In the next section we will discuss methods provided in the Java programming language, which are able, on the one hand, to prevent race conditions and ensure thread safety, and, on the other hand, to avoid liveness hazards.

3.4 Ensuring Thread Safety in Java

Multithreading is a very powerful feature which comes at a certain price. In concurrent multithreaded environments, there is a need to write code in a thread-safe way. Writing thread-safe code is, at its core, about managing access to a *state*, and being more precise to a *shared mutable state* [24]. We denote a *state* of an object as all data stored in *state variables* like instance or static fields. In other words, an object's state represents any data that can affect its externally visible behaviour. The state of an object can also include fields from other dependent objects, for instance, a state of a HashMap is stored in the HashMap object itself and also in many Map.Entry objects [24].
We talk about *shared state* if an object and its state can be accessed from multiple threads. We say *mutable shared state* when the values of the object's state could change during its lifetime.
The main aim of a thread-safe code is to manage access to *mutable shared state* of an object without exposing erroneous behaviour or producing unpredictable results. The primary mechanism to achieve thread-safety in Java is *synchronization* with a *synchronized* key word, which provides exclusive locking on the critical section. Nevertheless, there are several other methods like *volatile* variables, explicit locks, atomic variables, etc.
In this section different approaches to achieve thread-safety will be discussed.

3.4.1 Stateless Implementations

Listing 3.3 shows a simple implementation of a salary calculator, where the method *getSalary()* takes the amount of hours an employee has worked as input, multiplies it with an hour rate defined with a constant value, and outputs the multiplication result. The class *CalculateSalary* has no fields and references no fields from other classes. The state for a particular computation exists solely in the local variables that are stored on the thread's stack and are therefore accessible only to the executing thread.

```
class CalculateSalary {
  public final double paymentPerHour = 10,5;

  public static double getSalary(double hours){
   return paymentPerHour * hours;
  }
}
```

Listing 3.3: Stateless Salary Calculator Implementation (Thread-Safe Example)

Thus, if two or more threads execute the code of the class, the execution process of one thread does not influence the execution process of the other thread, because the two (or more) threads do not share state, if they were accessing different instances. Since the execution of a stateless object by one thread cannot affect the correctness of execution results in other threads, **stateless objects are always thread-safe** [24].

3.4.2 Atomic Variables

What happens if we need to count the number of salaries we have calculated? Listing 3.4 differs from listing 3.3 only by adding a counter variable which counts the number of method *getSalary()* calls.

```
class CalculateSalary {

  public int counter = 0;

  public final double paymentPerHour = 10,5;

  public static double getSalary(double hours){
   counter++;
   return paymentPerHour * hours;
  }
}
```

Listing 3.4: Stateless Salary Calculator Implementation (Not a Thread-Safe Example)

Unfortunately, the implementation in Listing 3.4 is not thread-safe any more. The operation *counter++* is not atomic, meaning that it contains a so-called *read-modify-write* operation, where before each increment there is a need to read the actual value, then add 1 to it and save the value to the *counter* variable. If the counter is initially 1, and two threads will simultaneously perform the incrementation operation and save the results to the counter variable, then with some unlucky timing each thread can set a counter to 2. Thus, such source code modification creates a race condition hazard.

```
class CalculateSalary {

  private final AtomicInteger counter = new AtomicInteger();

  public final double paymentPerHour = 10,5;

  public static double getSalary(double hours){
   counter.incrementAndGet();
   return paymentPerHour * hours;
  }
}
```

Listing 3.5: Stateless Salary Calculator Implementation (Not a Thread-Safe Example)

To avoid race condition hazards and achieve thread-safety in such cases, Java provides a set of atomic classes including: *AtomicInteger, AtomicLong, AtomicBoolean,* and *AtomicReference.* Atomic classes allow performing of atomic operations without using any other synchronization mechanisms. Each such atomic operation is executed as one single machine level operation [17].
A thread-safe implementation of salary calculator with counter is provided in Listing 3.5. Here the type *int* is replaced with *AtomicIneger.* The resulting class is again thread-safe.

3.4.3 Thread-Local Fields

Listings 3.6 and 3.7 (inspired by [80]) depict two thread-safe classes that do not share state between threads, because their fields (letters, and numbers) are thread-local. Both threads will print out lists of objects (in case of listing 3.6 – a list of strings, and in case of listing 3.7 a list of integers), but even if the two threads will be executed simultaneously, there will be no interaction between them, because the list variables are thread local.

```java
public class FirstThread extends Thread{

   private final List<String> letters = Arrays.asList("a", "b",
       "c");

   @Override
   public void run() {
      letters.forEach(System.out::println);
   }
}
```

Listing 3.6: Thread With Local String Fields

```java
public class SecondThread extends Thread{

   private final List<String> numbers = Arrays.asList(1, 2, 3);

   @Override
   public void run() {
      numbers.forEach(System.out::println);
   }
}
```

Listing 3.7: Thread With Local Integer Fields

Java provides a special *ThreadLocal* construct from the *java.lang* package, which allows developers to store data that will be accessible only by a specific thread. Consider Listing 3.8 where class *State* holds the state and class *ThreadState* uses the ThreadLocal construct to provide a current state to the running thread.

```
public class State{

   private final String state;

   // ... constructor; getter/setter methods

}

public class ThreadState{

   public static final ThreadLocal<State> stateOfThread = new
       ThreadLocal<State>(){
      @Override
      protected State initialValue(){
         return new State("intial");
      }
   };

   public static State getState(){
      return stateOfThread.get();
   }

}
```

Listing 3.8: ThreadLocal Fields in Java

Thread local fields are very similar to local class fields, except that each thread can access them via a setter/getter method. In this way each thread gets an independently initialized copy of the field and consequently each thread has its own independent state.

3.4.4 Locking

Another idea for ensuring thread-safety is to control access to the critical section with use of *locking* among threads. The main goal of locking is to ensure that only one thread at a time can access a critical section, while blocking access to that section for other threads. Java provides a built-in locking mechanism: *synchronized blocks*. A synchronized block consists of two parts: the reference to the object that will serve as a lock, and a block of code to be guarded by the lock. Listing 3.9 shows an example of such locking: a counter variable is an integer and its value will be increased in the method *incrementCounter()*, where locking is achieved by providing a special lock object.
Every Java object can act as a lock for the purposes of synchronization; these built-in locks are called **intrinsic locks** or **monitor locks**. The lock is automatically generated after one thread enters a critical section and is hold until the thread leaves it. Unfortunately, this provides also a drawback, if a thread never leaves the locked critical section, it will block all other threads forever [24].

```
class CounterClass {

  public int counter = 0;
  private final Object lock = new Object();

  public void incrementCounter(){
   // additional unsynced operations
   synchronized(lock){
     counter++;
   }
   // additional unsynced operations
  }
}
```

Listing 3.9: Synchronized Counter – Object as a Lock

Since every object can act as a lock, the following code in Listing 3.10 is also valid. In comparison to Listing 3.9, *this* is used for locking. According to the literature ([53], [24]), developers should prefer the implementation provided in 3.9 because of several reasons:

- **Loss of encapsulation:** any external code holding a reference to the object can inadvertently or intentionally synchronize on it, leading to unpredictable behaviour or deadlocks. As consequence, there is an increased risk of deadlocks.
- **Unintended lock contention:** using *this* for locking ties the lock to the object itself. Any other code synchronizing on the same object (even outside developers control) competes for the lock, causing unnecessary contention. This can reduce the performance and scalability of the application
- **Difficulty in code maintenance:** locking on *this* makes it harder to identify and debug synchronization issues since the lock is not explicitly defined or documented.

```
class CounterClass {

  public int counter = 0;

  public void incrementCounter(){
   // additional unsynced operations
   synchronized(this){
     counter++;
   }
   // additional unsynced operations
  }
}
```

Listing 3.10: Synchronized Counter – This Object as a Lock

If the whole method needs to be locked, Java provides a *synchronized* shorthand that spans an entire method body, and whose lock is the object on which the method is being invoked. Listing 3.11 depicts an example for the synchronized method *incrementCounter*.

```java
class CounterClass {

   public int counter = 0;

   public synchronized void incrementCounter(){
    counter++;
   }
}
```

Listing 3.11: Synchronized Counter – Whole Method Locking

Technically, when one thread requests a lock for a section or a method where another thread is already operating, the requesting thread blocks. But because of the fact that the intrisic locks are *reentrant*, if a thread tries to acquire a lock that it already holds, the request will succeed. Reentrancy is implemented by associating with each lock a counter. When the counter is zero, the lock is considered unheld. When a thread acquires a previously unheld lock, the JVM will record the owner and increment the counter to 1. If the same thread acquires the lock again, the count will be incremented again. When the holding thread leaves the critical section, the counter will be decremented. Only when the counter reaches zero, the lock is considered to be released.
Reentrancy is very helpful by encapsulation of locking behaviour, and simplifies the development of object-oriented code. Consider an example in Listing 3.12. In the example there are 2 classes, where *CustomTest* extends *Test* and in the synchronized method *performTest* calls a synchronized method with the same name in a parent class. Without the reentrancy concept this code would deadlock. Because of the fact that both methods in classes *Test* and *CustomTest* are synchronized, each would try to acquire the lock on *Test* before proceeding. Without reentrancy, the call *super.performTest()* would never be able to acquire the lock, because it would be considered already held, and the thread would permanently be waiting for the lock it can never acquire.

```java
class Test {
    public synchronized void performTest(){
        // code
    }
}

class CustomTest extends Test{
    public synchronized void performTest(){
        super.performTest();
    }
}
```

Listing 3.12: Reentrancy

Another powerful locking mechanism to achieve thread-safety is the use of **ReadWrite-Lock** implementations. The construct uses the pair of associated locks, one for read-only operations, and the other for writing operations. Consequently, many threads can read a resource as long as no thread is writing to it. When one thread requests a write lock, then the read lock will prevent others from reading the resource until the writing thread is finished. Listing 3.13 provides an example for ReadWirteLock [80].

```java
class ReadWriteLockCounter {

   private int counter;
   private final ReentrantReadWriteLock readWriteLock = new
       ReentrantReadWriteLock();
   private final Lock readLock = readWriteLock.readLock();
   private final Lock writeLock = readWriteLock.writeLock();

   public void incrementCounter() {
      writeLock.lock();
      try {
         counter++;
      } finally {
         writeLock.unlock();
      }
   }

   public int getCounter() {
      readLock.lock();
      try {
         return counter;
      } finally {
         readLock.unlock();
      }
   }

}
```

Listing 3.13: ReadWriteLock [80]

3.4.5 Volatile Variables

Regular Java class fields can be cached by the CPU. Consequently, updates to a particular field, even if it is synchronized, can in some cases not be visible to other threads. To overcome this limitation and ensure that updates to a variable are propagated predictably to other threads, the Java programming language introduced *volatile variables*. They are not cached in registers or in caches where they are hidden from other processors, so a read of a volatile variable always returns the most recent write by any thread [22]. Accessing the volatile variable is non-blocking and performs no lock for all running treads, making them a light-weight synchronization mechanism compared to the *synchronized*

key word.
Moreover, the use of a volatile variable ensures that all variables that are visible to a given thread, will be read from the main memory as well. For instance, assume the existence of two threads A and B. When thread A writes to a volatile variable and subsequently thread B reads the same variable, the values of all variables that were visible to A prior to writing to the volatile variable become visible to B after reading the volatile variable. Thus, from a memory visibility perspective, writing to a volatile variable is like executing the *synchronized* block, and reading a volatile variable is like entering a synchronized block [24].
Listing 3.14 shows the typical use of the volatile variable *wait*. In this case the variable needs to be volatile to ensure thread safety, because otherwise the thread might not notice when *wait* has been set by another thread.

```
volatile boolean wait;

while(!wait){
   ...
}
```

Listing 3.14: Volatile Variable

Volatile variables provide thread safety in following use cases:

- When only one thread writes to a volatile variable and other threads read the value.
- When multiple threads write to the volatile variable and the operation is atomic. Meaning, that the new value of the variable is not dependant to the old one.

Due to the fact that volatile variables are a lightweight synchronization mechanism, they do not provide thread safety when non-atomic operations are performed on shared variables. The short time gap between composite operations can create race conditions while using volatility.

```
final class ImmutableStudent {
   private final long id;
   private final String name;

   public ImmutableStudent(long id, String name){
      this.id = id;
      this.name = name;
   }

   // get methods for the variables
}
```

Listing 3.15: The Class for Creation of Immutable Objects

3.4.6 Immutability

An immutable object is one whose state cannot be changed after construction. Immutable objects are always thread safe, because their invariants are created during the construction and since their state cannot be changed, these invariants hold forever.
In the Java programming language, an immutable object is created by using final classes, where all fields are also final and the class constructor is the one and only place responsible for class creation. Listing 3.15 depicts the class for creation of immutable objects.

3.5 Ensuring Liveness in Java

There is often a problem to ensure both safety and liveness properties at the same time, because of the use of locking strategies for establishment of thread safety. Liveness hazards, like deadlocks, tend most often to appear in situations when one thread locks a critical resource or critical section and does not release it back. The causes for this scenario can be different: the blocking thread can abruptly stop its work and never release the lock because of an error; the blocking thread can itself wait for the resources to continue its operation, which are also locked by another thread; the blocking thread can enter a never ending loop inside the critical section and thus, never quit from it, consequently never release the lock; etc.
The main objective to prevent liveness hazards is to solve liveness issues at the modelling stage of software development, so that they do not occur in the implemented software solution [35]. This requires, on the one hand, deep understanding of concurrency and concurrent code execution by developers, and, on the other hand, good organization and planning of software processes, complex reviews of ideas before implementation with an intent to find potential liveness problems and mitigate them. A completely general treatment of liveness is quite complex and requires the use of *temporal logic*[2] to specify the required liveness property [35].
In general, a program that never acquires more than one lock at a time cannot experience a deadlock. In programs where this is not a case, developers must identify where multiple locks could be aquired, and then perform a global analysis of all such instances to ensure that lock ordering is consistent across the entire program [24].
Another technique for ensuring liveness is to use the *tryLock* feature of the explicit *Lock* classes instead of intrinsic locking. Thereby, explicit locks let a developer specify a timeout after which *tryLock* returns failure, while intrinsic locks wait forever if the lock cannot be acquired. But, if a timed lock fails, the developer cannot be sure what was the reason of failing. It can be really a deadlock, or a loop took longer than expected, or maybe the activity in general takes more time to run than expected. Still this technique is convenient for liveness hazards analysis, because the developer gets an opportunity to record the failing attempts, log useful information about the failure, and restart the computation without killing and restarting the whole process [24].

[2]**Temporal Logic** is a branch of mathematical logic which is used for reasoning about propositions qualified in terms of time.

While preventing liveness hazards, *thread dumps* provided by the JVM can be helpful for analysing occurred liveness problems. The thread dump includes a stack trace for each running thread, locking information, such as which locks are held by which threads, in which stack frame they were acquired, and which lock a blocked thread is waiting for acquire. Explicit locks do not show up in thread dumps until inclusive Java 5. Java 6 does include deadlock detection with explicit locks, but the information on where locks are acquired is less precise than for intrinsic locks. The reason for this is that intrinsic locks are associated with the stack frame in which they are acquired, and explicit locks are associated only with the acquiring thread [24].
A significant role in liveness assurance play priorities of threads. The Thread API defines ten priority levels that the JVM can map to operating system scheduling priorities as it sees fit. This mapping is platform specific, so two Java priorities can map to the same OS priority on one system and different OS priorities on another system. This is because of the fact, that different OS have different numbers of priority levels. OS schedulers try to provide fair schedules to all threads according to their priorities. In most Java applications, all application threads have the same priority. In the literature it is strictly recommended to avoid changes of thread priorities, because with modification of priorities the software application becomes platform specific and risks of liveness issues can be introduced as soon as software will change the running platform, or modifications will take place in the original platform itself. Nevertheless, thread priorities are sometimes helpful for mitigation of *starvation* [22] [24].
The situation when two or more threads are blocked to continue their actions because they keep retrying an operation which can be made only by one thread at a time, is called *livelock*. As discussed above, a good strategy to avoid livelocks is to introduce some randomness into the retry mechanism, so that different threads make a retry not simultaneously, but after some random waiting time. In this case, shared operation will be processed sequentially by different threads, which will try to access the shared resource in different time periods.

3.6 Design of Concurrent Applications Using Threads

According to Clay Breshears [9], "concurrent programming is still more art than science". Therefore, we can define several rules applicable during the design phase of concurrent applications using threads, with an overall goal to facilitate the development, debugging and further maintenance of the application. Following the rules is not mandatory, but adhering to them can lead to an increase of code quality, simplification of development, decrease of bugs and software errors, and can have a positive impact on performance and efficiency of the application.
Rule 1. Identification of Independent Computations. There are situations where concurrency is not applicable and a solution must not be implemented concurrently. Developers shell identify such situations and shell not strive to implement them using concurrency mechanisms, but to implement the solution in a standard sequential way. Finding the boundaries of concurrency is in some cases not an easy task [24].

Consider a real-world scenario involving two pathways: one designated for joggers and another for vehicles. If these pathways intersect, a concurrency issue arises during software modelling of the situation. Specifically, a mechanism must be implemented to manage access, ensuring that either the joggers or the vehicles wait to allow the other party to cross safely. Conversely, if the pathways do not intersect (e.g., due to the presence of an overpass or an underpass), there is no requirement to handle this scenario with concurrency mechanisms, as the two streams of traffic operate independently.
Rule 2. Scalability must be planed in early phases of application design. Under *scalability* we understand the ability to handle changes like increases in the number of cores in hardware. Nowadays, there are multiple cores available in many hardware devices and the tendency of manufacturers is to further increase their number. Therefore, the code must be flexible enough to take advantage of different numbers of cores, especially if code can be executed concurrently [9].
Tuning the concurrent application for scalability means to find certain ways how to parallelize the problem to take an advantage of additional processing resources. In other words, there is a need to find a way to do more work with more resources.
In order to achieve higher scalability, the amount of work to process each individual task is often increased, so that each task gets divided into multiple "pipelined" subtasks. In such a case, the access to resources guarded by the exclusive lock (where only 1 thread can access it at a time) is the principal threat to scalability in every concurrent application. If one subtask locks the divided resource, all other subtasks can be blocked, if they require the same resource [24].
According to *Little's Law*[3] there are two main factors that influence the scalability of locking: how often the lock is requested and how long it is held. If the product of these values is relatively small, then locks will not pose a significant scalability impediment. If, however, the locks will occur often and last long, threads will be blocking each other and processors will stay idle and wait. To reduce lock contention we can, on the one hand, reduce the duration for which locks are held, and, on the other hand, reduce the frequency with which locks are requested [24].
Talking about scalability is also only possible if the problem itself can be solved faster with more resources. Taking an example from real life, if there are more farmers, harvesting can be completed faster, but if there are more drivers or even more cars, delivery of some object will not be completed faster. *Amdahl's Law* describes how much the program can theoretically be sped up by additional computing resources. The Amdahl's Law is explained in Appendix B.
Rule 3. Concurrency must be implemented on the highest level possible. Rich threading APIs, for instance, the Java concurrency API, offers different classes to implement concurrency in software applications. In the case of Java, the creation and synchronization of threads using the Thread or Lock classes can be controlled, but it also offers high-level concurrency objects, such as executors or the Fork/Join framework that

[3]**Little's Law:** the average number of customers in a stable system is equal to their average arrival rate multiplied by their average time in the system. Named after **John Little** (February 1, 1928 - September 27, 2024) was an Institute Professor at the Massachusetts Institute of Technology (MIT).

support the execution of concurrent tasks. In general, high-level mechanisms offer the following benefits (as an example the Java concurrency API is taken) [9] [24] [72] [10]:

- The Java concurrency API is able to control the creation and management of threads, so that the developer does not need to worry about the creation and management of threads.
- The Java concurrency API is optimized to give better performance than using threads directly. For example, it uses a pool of threads to reuse them and avoid thread creation for every task.
- The software application will have an ability to be migrated easier from one operating system to another, and it will be more scalable.

Rule 4. The usage of thread-safe libraries is mandatory. This rule is a consequence of the previous one. According to Breshears [9] there is no need to "reinvent the wheel" by writing code that is already encapsulated in some well-tested library. This brings also another benefit: If there is a bug in the library, then the odds are high that these bug occurred also in other programs and other developers have already introduced a workaround or a bug fix.
Rule 5. The particular order of thread execution can be never assumed. This rule takes its roots from the habit of developers to work with sequential programs, where it is absolutely clear in what order what line of code will be executed. While talking about threads, we are talking about non-deterministic execution order, which cannot be forecasted or guessed. In different situations, on different machines, and even with different users, concurrent program can behave absolutely unexpectedly.
Unfortunately, many developers still tend to rely on some assumptions of threads execution ordering, which sometimes leads to unexpected program behaviour. Due to traditional thinking patterns and a lack of cognitive flexibility among developers, identifying and implementing bug fixes can, in certain cases, require significant resource expenditure.
Rule 6. Synchronization is the necessary evil. Synchronization itself is useful if many threads try to access the same data storage. It gives an opportunity to provide data consistency for threads and to implement sort of a queue for threads to access the shared area.
Unnecessary synchronization can create additional waiting times for threads and decrease the performance of the whole application. The amount of synchronization must be kept as small as possible to avoid potential problems with software application appearance by the clients.
Rule 7. Locks must be associated with 1 single data object. This rule is a direct consequence of the so-called *Segal's Law*[4]. If two different lock objects protect access to the same variable, one part of the code may use the first lock for access, while another section of code can use another one. Each thread will assume the exclusive access to

[4]**Segal's Law:** a man with a watch knows what time it is. A man with two watches is never sure.

data, but in fact, two threads will perform operations on it.
Rules presented above give some ideas where to pay attention while designing the concurrent application using threads. We will use some of them during the implementation of the concurrent scenario.
Next Section is devoted to virtual threads.

3.7 Introduction to Virtual Threads

Java virtual threads are a new threading feature introduced in Java 19. In Java 21, they were finalized and now serve as an alternative to platform threads.

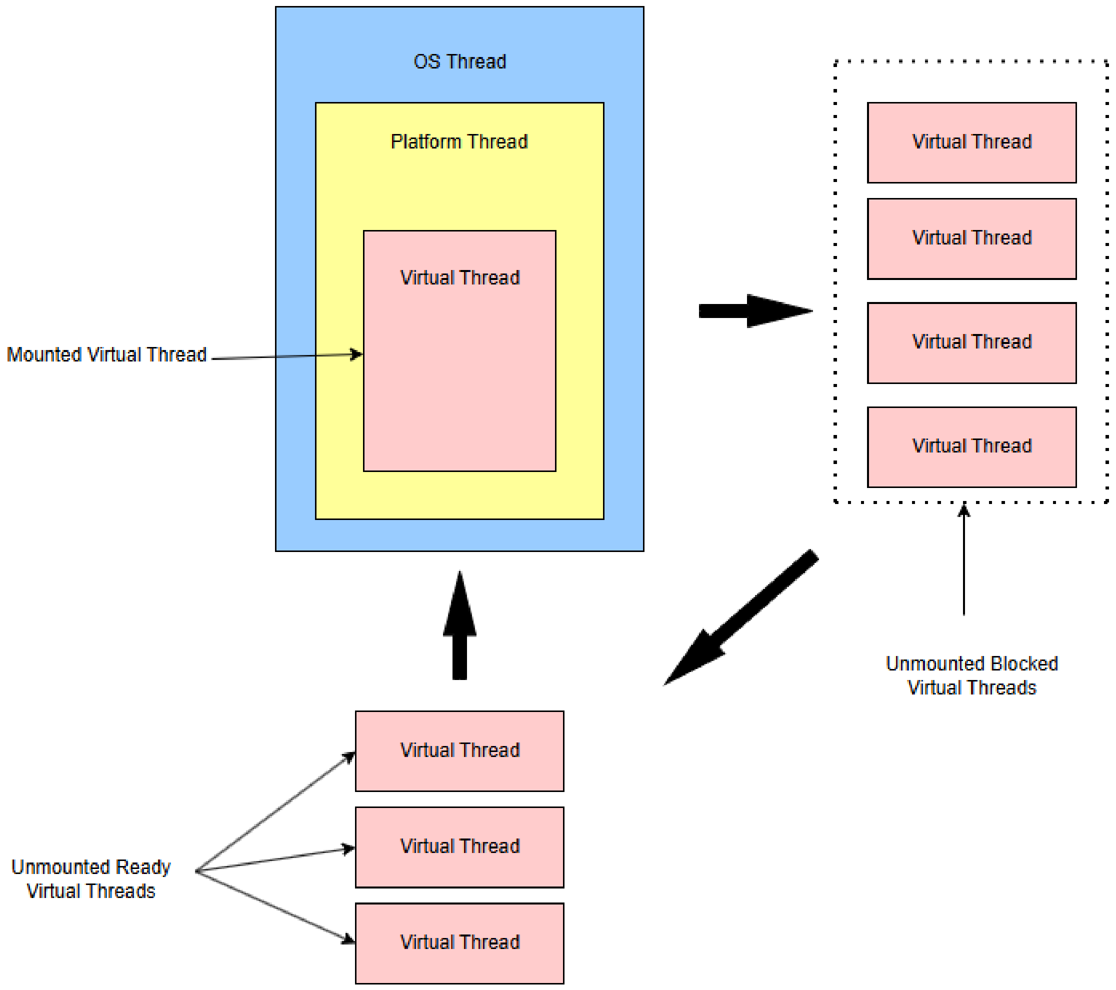

Figure 3.2: The Concept of Virtual Threads [30].

According to [82], the creation process of virtual threads does not require calls to the

operating system (OS) and does not allocate a default amount of memory like it is done for platform threads. Creating a single virtual thread requires only a few bytes of memory. When additional memory is needed, the virtual thread can dynamically allocate it, as its stack is resizeable. This allows the virtual thread to acquire more memory when necessary and release unused memory back to the application [82].
Figure 3.2 depicts the concept of virtual threads. Java virtual threads are executed by platform threads, which are executed by operation system threads. A platform thread can only execute a single virtual thread at a time. The virtual thread currently being executed by a platform thread is referred to as a *mounted virtual thread* and the platform thread is often called a *carrier thread*. When the currently executed virtual thread makes a blocking call, it can be suspended from the execution in a queue of *unmounted blocked virtual threads*. In the meantime the platform thread can process another virtual thread from the queue of *unmounted ready virtual threads*. As soon as a previously suspended virtual thread is ready for being executed again, it comes into the *unmounted ready virtual threads* queue and waits until some carrier thread will take it for execution. New virtual threads are also queued up into the *unmounted ready virtual threads* queue until a platform thread is ready to execute them [30] [82]. The mounting and unmounting process makes sure that the carrier thread, which is an expensive resource, is always doing a task [82].
By default, the number of carrier threads corresponds to the number of cores in the executing system. For instance, a four-core system will have four carrier threads. However, it is possible to modify the initial number of carrier threads [82].
When a virtual thread performs a blocking file system call, it remains mounted and does not unmount from the platform thread. During such operations, the virtual thread stays *pinned* to the platform thread. This means that the platform thread is unable to execute any other virtual thread while waiting for a response from the file system. However, to prevent performance degradation while virtual threads pinning, the JVM helps by creating temporary additional carrier threads. These extra threads will handle new virtual threads while the existing carrier threads are blocked. Once the extra carrier threads are no longer necessary, they are discarded. By default, the maximum number of carrier threads is limited to 256. This parameter is configurable [82] [30].
There are other scenarios that can pin a virtual thread to a platform thread. For example, when entering a synchronized block. If the virtual thread makes a blocking call within a synchronized block, it may also remain pinned to the platform thread [30]. As a solution to avoid virtual thread pinning in this case, the usage of reentrant locks (*java.util.concurrent.locks.ReentrantLock*) has been proposed [56]. However, if synchronized blocks are short and contention is low and/or if mutual exclusion must be guaranteed, there is no need to rewrite synchronized blocks into reentrant locks [82].
According to the official Java documentation it is recommended to "use virtual threads in high-throughput concurrent applications, especially those that consist of a great number of concurrent tasks that spend much of their time waiting. Virtual threads are not faster threads; they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency)."[56]

On the other hand, the key consideration when using virtual threads is how frequently and for how long they will block. If a virtual thread never blocks or only does so for a very short time, using platform threads might be a better choice [82].
Listing 3.16 shows an example how Java virtual thread can be started. In this example virtual threads are executed immediately after their creation. In Listing 3.17 *unstarted()* is used, meaning that execution of virtual threads will not start immediately after their creation. To start an unstarted virtual thread there is a need to call the *start()* method.

```
Thread virtualThread = Thread.ofVirtual().start(() -> {
       System.out.println("Running in Virtual Thread: " +
           Thread.currentThread());
   });
```

Listing 3.16: Starting Virtual Threads Immediately

```
Thread virtualThread = Thread.ofVirtual().unstarted(() -> {
       System.out.println("Running in Virtual Thread: " +
           Thread.currentThread());
   });
```

Listing 3.17: Virtual Threads Creation Without Starting Immediately

Verification of whether all virtual threads have terminated works the same way as with platform threads – developers can use the *join()* method.
In the next chapter reactive programming with RxJava will be discussed.

CHAPTER 4

Reactive Programming with RxJava

Everything changes but change itself. Everything flows and nothing remains the same... You cannot step twice into the same river, for other waters and yet others go flowing ever on.

— Heraclitus.

Reactive programming is a functional event-driven programming approach which began to receive a special attention recently. There are numerous reactive programming frameworks designed for various platforms and programming languages, among them, for example: *Project Reactor*[1], *Akka Streams*[2], etc [65].
At the same time Microsoft[3] created a reactive programming framework for .NET called **Reactive Extensions**. In several years reactive extensions were ported to several languages and platforms like C++, JavaScript, Python, Swift, and Java. **ReactiveX** or also known as **Rx** became a cross-language standard of reactive programming.
In November 2014 Ben Christensen from Netflix[4] and David Karnok released *RxJava 1.0* as the ReactiveX port for Java. In November 2016 they also released *RxJava 2.0*. RxJava became the backbone to other ReactiveX JVM ports such as RxScala, RxKotlin,

[1]**Project Reactor** is a reactive library, based on the Reactive Streams specification, for building non-blocking applications on the JVM. URL: https://projectreactor.io/ (accessed: 19.03.2025).

[2]**Akka Streams** is a library built on the Akka Actor Framework, which adheres the reactive stream manifesto. Akka Streams allow users to compose data transformation flows in a reactive, non-blocking, and asynchronous way. URL: https://akka.io/ (accessed: 19.03.2025).

[3]**Microsoft Corporation** is an American multinational technology corporation which was founded on April 4, 1975 by Bill Gates and Paul Allen, which produces computer software, consumer electronics, personal computers, etc. Headquarters is located in Redmond, Washington, United States.

[4]**Netflix, Inc.** is an American subscription streaming service and production company founded at August 29, 1997 by Reed Hastings and Marc Randolph in Los Gatos, California, United Sates.

and RxGroovy. It has become a core technology for Android[5] development and is also used in Java backend development [65] [49].

4.1 Reactive Programming Concepts

Reactive Programming is a design paradigm that relies on asynchronous programming logic to handle real-time updates to otherwise static content. Reactive programming utilizes *data streams*, which are coherent, cohesive collections of digital signals created on continual or nearly continual basis. These data streams are sent from the source (for instance, temperature sensor, transaction generator, product database, etc.) in reaction to an *event*. An event can be, for example, a call from some IoT system, some message, or a routine call. The programmer creates a program that can respond to events. Such style of writing software is called *event-driven programming*, where the flow of the program is determined by events like users actions (mouse clicks, key presses, etc.), sensor outputs or message passing from another programs [20] [37].
When an event occurs, reactive programming allows updates to propagate through the system without requiring explicit instructions for every dependency. For example, if a data source updates, all computations or actions depending on it automatically receive the update.
For instance, consider a hypothetical reactive system as a network of interconnected components. Each component represents a function or transformation, and the connections represent dependencies. When an input changes, it propagates through the system, updating all dependent components without the need to re-execute the entire network. This propagation happens automatically, respecting the declared relationships and transformations.
Many developers are getting confused while talking about asynchronous programming and the difference to its counterpart: synchronous programming. But because asynchronous programming is the core in the definition of reactive programming, the next subsection will introduce differences between these concepts.

4.1.1 Synchronous vs. Asynchronous Programming

In this subsection the differences between synchronous and asynchronous programming will be presented and discussed.
Observe Listing 4.1 written in pseudo code. There is a class *DataFromUrl* which consists of two methods: *fetchData()* and *getDataFromUrl()*. The *fetchData()* Method takes an URL as input, loads the data available under the given URL and returns a String with relevant information. Another method *getDataFromUrl()* uses *fetchData()* and saves the result in the *data* variable. Afterwards, the program prints data itself and the word "FINISH" to indicate that printing is finished.

[5]**Android** is a mobile operating system based on the modified version of *Linux kernel* and other open source software, designed primarily for touch screen devices such as smart phone and tablets.

```
public class DataFromUrl{

   private String url;

   public void getDataFromUrl(){
      String data = this.fetchData(url);
      print(data);
      print("FINISH");
   }

   public String fetchData(String url){
   ...
    }
  }
```

Listing 4.1: Synchronous Programming Illustration.

If the code from Listing 4.1 is executed, the code is executed line by line and the output will not be surprising: first, the content of *data* will be printed, then the user will see "FINISH". Such execution is called **sequential** and the programming style is called **synchronous programming**, because everything happens synchronously: after ending of previous execution, the next sequential execution (in our case, next line of code) is triggered.

Now observe Listing 4.2. There is the same scenario as in Listing 4.1, except the only difference: the *async* block inside the *getDataFromUrl()* method, where it is explicitly stated that fetching of data from the given URL and printing of the data variable content can happen in an asynchronous way.

```
public class DataFromUrl{

   private String url;

   public void getDataFromUrl(){
      async{
          String data = this.fetchData(url);
          print(data);
      }
      print("FINISH");
   }

   public String fetchData(String url){
   ...
    }
  }
```

Listing 4.2: Asynchronous Programming Illustration.

After the execution of the code given in Listing 4.2 the user usually will see first "FINISH", and then the data. This situation illustrates the difference between synchronous and asynchronous programming. During the execution of the code given in Listing 4.2 every line is still executed synchronously, except the lines inside the *async* block. This simply means, that the contents of the *async* block is executed independently from the main program flow. For this reason, the output can first show "FINISH", because it can take far less time for the main flow to reach the corresponding line of code than to fetch and print the data from the URL in an asynchronous manner.
Synchronous as well as asynchronous paradigm has its benefits and drawbacks. Synchronous programming is more efficient and has predictable results, but such programs are monolithic (hard to scale) and have limited concurrency possibilities. Asynchronous programming can benefit from better scaling possibilities, more concurrency affinity and higher throughput, but asynchronous programs are more complex and can have a significant risk of errors, which are hard to understand and debug [78] [13].
In the next subchapter the notion of asynchronous data streams will be defined and explained.

4.1.2 Asynchronous Data Streams

A *stream* is a sequence of ongoing events ordered in time. There are many streams around people in real life: currency exchange rates, prices for every product in supermarket, news published in newspapers or shown on TV, etc. Every change of a value by exchange rates, prices, or news creates an event which is atomic, finite and bound to time when the event happened. Consequently, the stream of events can be shown with use of a time arrow as depicted in Figure 4.1. Here the arrow represents a flow of time, and each circle is an event. Note, the distance between circles is equal, thus this example shows a synchronous stream of data.

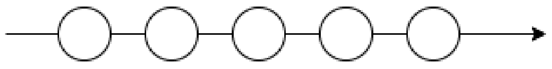
Figure 4.1: Synchronous Stream Of Events.

An **asynchronous data stream** is a stream of data where values are emitted with a delay between them. The data can appear anywhere in time: seconds, minutes, hours, days, months, or even years after the previous event. Reactive programming utilizes asynchronous data streams [50].
The fundamental idea behind reactive programming can be formulated as follows: *events are data and data are events*. Merging events and data allows the code to feel organic and represent the model of the world more accurately [65].
We will call the process of listening to the data stream **subscribing** and the functions which will utilize the data **observers**. The data stream is the subject being observed by observables, which in its turn is very similar to the Observer Pattern (a recap of the Observer Pattern is provided in Appendix A)

In the next section the reactive manifesto will be presented. The main goal of this document is to define what characteristics should a system have to be considered reactive.

4.2 Reactive Manifesto

On September 2014 J. Boner et al. proposed the **Reactive Manifesto**, which aims to present and describe 4 main characteristics of any reactive system. They postulated that all reactive systems are:

- **Responsive**: the system is responsive and provides rapid and consistent response times and deliver a consistent quality of service;
- **Resilient**: the system is responsive even if failures occur. Resilience must be achieved by replication, containment, isolation and delegation. The isolation of components from each other can ensure that failed components can recover without compromising the system as a whole. Recovery of a component can be delegated to another component and the availability can be ensured by replication if necessary.
- **Elastic**: the system stays responsive under different workloads. The system is reactive if it is able to adopt usage of resources to changes of the input rate. This feature must be achieved in a cost-effective way on commodity software and hardware platforms.
- **Message Driven**: reactive systems use asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

According to the authors of the reactive manifesto, the characteristics shown above are connected in a way shown in Figure 4.2.

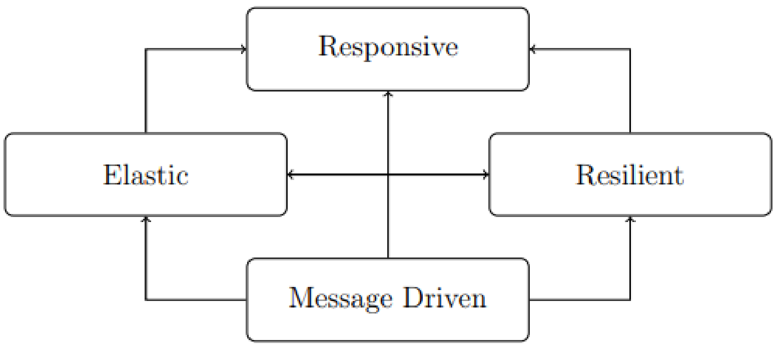

Figure 4.2: Connections Between Characteristics of Reactive System According to Reactive Manifesto.

The Reactive Manifesto is published online[6] and can be signed by everyone interested.

4.3 RxJava

RxJava is a Java VM implementation of Reactive Extensions: a library for composing asynchronous and event-based programs by using observable sequences.

4.3.1 RxJava Versions

Currently there are three co-existing versions of RxJava: 1.X, 2.X, and 3.X. The end-of-life term of 1.X was on March 31, 2018, and of 2.X was on February 28, 2021. For both versions no further development, support and maintenance will happen. The very last version of 1.X is 1.3.8, and for 2.X is 2.2.21. For both of these versions the documentation is still available [65].
The 3.X version of RxJava is currently actively developed and used, and the very last version for now is 3.1.9. This version will be used during the implementation of a concurrent scenario in boundaries of this work. It is released on August 13, 2024 and comes at exactly 3.7 MBs in size.
RxJava 3.1.9 has one dependency, called *Reactive Streams*, which is a core library, developed also by the creators of RxJava, that sets a standard for asynchronous streams implementation.

4.3.2 Observable and Observer

The core type of RxJava is the *Observable*. The *Observable<T>* class is a push-based composable iterator: it pushes items (which are called *emissions*) of type T, and the underlying structure provided by Rx libraries will propagate emissions to another component, called *Observer* [81].
Take a look at Listing 4.3, where the Observable interface is presented. It has one single method *subscribe*, which is used by Observer to receive emissions from the given Observer.

```
 interface Observable<T>{

     Subscription subscribe(Observer s);

}
```

Listing 4.3: The Observable Interface

Observable works by passing 3 type of events to an Observer:

- *onNext()*: this event passes one emission at a time from Observable to Observer.

[6]Reactive Manifesto link: https://www.reactivemanifesto.org/ (accessed: 19.03.2025)

- *onComplete()*: this event indicates that the observable has finished emitting items and has successfully terminated without any errors.
- *onError()*: this event communicates an error to Observer, which typically defines how to handle it. After this event the Observable terminates and no more emissions occur.

Listing 4.4 depicts the Observer interface. Events, which Observable passes to Observer compose the Observer type. The *onSubscribe()* method was added since RxJava 2.0, and it allows the Observer to have the ability to dispose the subscription at any time. For example, Observer's methods *onNext(), onComplete(),* and *onError()* can be implemented in a way, that they will have an access to Disposable, so that all three events will be able to call *dispose()* if the Observer does not want to get any more emissions. In such a way the unsubscribing of an Observer from the emissions sent by Observable is typically implemented.

```
import io.reactivex.rxjava3.disposable.Disposable;

   interface Observer<T>{

       void onSubscribe(@NotNull Disposable d);
       void onNext(@NotNull T value);
       void onError(Throwable e);
       void onComplete();

  }
```

Listing 4.4: The Observer Interface

Dependences between Observable, Observer and Disposable are depicted in the class diagram presented in Figure 4.3 [61].

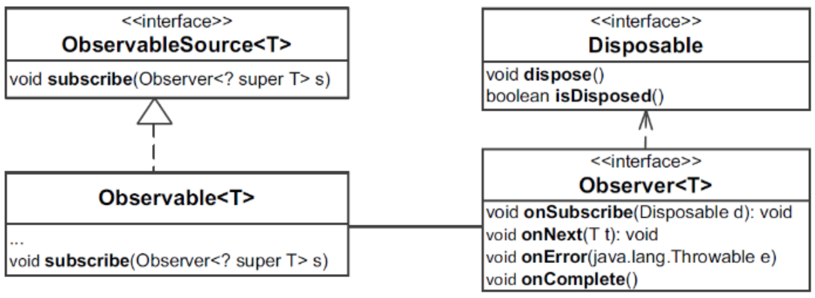

Figure 4.3: The Class Diagram of Observable, Observer, and Disposable [61]

.

For example, Listing 4.5 demonstrates how the Observable can be implemented and how Observer can subscribe to its emissions. Note, for creation of the Observable method *just()* is used, which enables an opportunity to just list all emissions, if the values are known before code execution. Methods *onNext(), onComplete(),* and *onError()* are implemented similarly, each method just prints the received values to the output console, and *onComplete()* prints just the word "DONE". The output of the program, will contain all capitals from source prefixed with "RECEIVED: ". At the end, when all emissions will be processed, the word "DONE" will appear.

```
  public static void main(String[] args){
    Observer<String> source =
        Observable.just("Kyiv", "Vienna", "London", "Rom");
     Observer<String> myObserver = new Observer<>(){
       @Override
       public void onSubscribe(Disposable d){
         // do nothing
       }

       @Override
       public void onNext(String line){
         System.out.println("RECEIVED: " + line);
       }

       @Override
       public void onError(Throwable e){
         System.out.println(e.getMessage());
       }

       @Override
       public void onComplete(){
         System.out.println("DONE");
       }
     };

     source.subscribe(myObserver);
  }
  }
```

Listing 4.5: The Observable-Observer Example

The Observables can be grouped in two different classes: **cold Observables** and **hot Observables**. We denote an Observable as cold, if it replays the emissions to all Observers and ensures that all Observer get all the data. A hot Observable broadcasts emissions to all Observers at the same time. If an Observer subscribes to a hot Observable, it automatically will get emissions, and if some other Observer will later subscribe to the same Observable, it will get currently emitted emissions and will miss previously send ones [81].
As an example from real life, which can help to distinguish between hot and cold Observables, can dean a comparison between a newspaper and news program on TV. In

a newspaper each reader can read all news one by one, page by page, and each reader can begin from the beginning and read to the end, thus each reader will receive all the news. This is a perfect example of a cold Observable. On the contrary, the hot Observable can be imagined as a TV news stream, only persons, who turned on the TV at the beginning of the news program will receive all the news, all others, who may turn on the TV later, will receive only later news and will not be able to revert time and get earlier reportings. There are plenty ways how Observable and Observer can be created and a large number of diverse basic operations, which can be applied on emissions received for Observable. Some of these methods and operations are very useful and will be used during the implementation of a concurrent scenario, but will not be discussed here in detail, because such a discussion lies beyond the boundaries of this thesis. The summary of basic operations and objects creation methods is provided in the literature (for instance, [65]).

4.3.3 Observable Cardinality

In the last subsection the basic Observable was presented. There are several specialized flavors of Observable that are explicitly set up for special cases of emission cardinality: *Single<T>, Maybe<T>, Completable.* They all follow Observable closely and can be used in special cases, which will be discussed further (based on [65]).
Single<T> is an Observable that always emits only one value or throws an error. For this reason, the *SingleObserver<T>* interface contains only two methods: *onSuccess()*, and *onError()*. Intuitively, the *onComplete()* method is not needed any more, since the Single<T> emits only one emission. A typical use case of Single observable would be when we make a network call and receive a response, which needs to be processed or acknowledged.
Maybe<T> is an Observable that may or may not emit a value. For example, we would like to know if a particular user exists in the database. The user may or may not exist. The *MaybeObserver* is much like a standard Observer, but instead of calling *onNext(), onSuccess()* is called instead. Note, *Maybe<T>* can still not emit more than one item.
Completable does not emit any data and is concerned with an action being executed. It does not posses *onNext()* or *onSuccess()* methods, but it does have *onError()* and *onComplete()*. This Observer can be used in cases where some update is happening and the process itself does not return any value, but developers want to be sure that the process ended successfully.

4.3.4 Backpressure and Flowables

In previous subsections we looked at different Observables and postulated the main task of an Observable – producing emissions. Assume the existence of a fast-producing Observable. It means, without loss of generality, that an Observable produces a large amount of emissions in a given time span. The Observer, which is subscribed to a fast-producing Observable can for some reasons be slower in consuming than an Observable in producing emissions. In such a case, the stream of messages originating from Observable will continue to grow, and the Observer will get more and more behind. RxJava's default

behaviour is to buffer these events until the observer gets around to processing them. This can lead to the problem of an *OutOfMemoryException*.
To conquer this problem the concept of **backpressure** was introduced. RxJava's backpressure system is where the downstream Observer is capable of communicating how many more requests it wants to fulfil. This information is passed upstream so that all upstream Observables only produce events at the given rate. This gives an opportunity to hold back the stream of events so that events do not sit in memory and take up the space [65] [20].
RxJava provides a solution for the backpressure by using a new type of Observable known as *Flowable*. RxJava's Observable, by default, is completely unaware of backpressure. Because of this, the standard Observable in RxJava can be susceptible to out of memory exceptions. The Flowable class is backpressure aware. This does put extra demands on Flowable subscriber implementations. To subscribe to Flowables the Subscriber interface is used. Subscriber has the same methods with the same semantics as Observer. Unlike the Disposable interface used by Observer, Subscriber uses a Subscription interface to cancel the reception of emissions. Figure 4.4 shows a class diagram and dependences between Flowable and Subscriber entities [65].

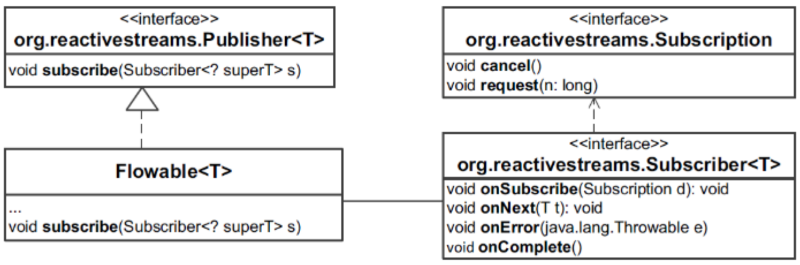
<<interface>>
org.reactivestreams.Publisher<T>
void subscribe(Subscriber<? superT> s)
<<interface>>
org.reactivestreams.Subscription
void cancel()
void request(n: long)
Flowable<T>
...
void subscribe(Subscriber<? superT> s)
<<interface>>
org.reactivestreams.Subscriber<T>
void onSubscribe(Subscription d): void
void onNext(T t): void
void onError(java.lang.Throwable e)
void onComplete()

Figure 4.4: The Class Diagram of Flowable, Subscriber, and Subscription [61].

There are several backpressure strategies which the Flowables can follow (inspired by [20]):

- *Drop* indicates that all the streamed items that cannot be processed until downstream can accept more of them will be ignored.
- *Buffer* creates a bounded or unbounded buffer that holds the emitted items that could not be processed by the downstream. Note, in case of buffer overflow (if the buffer is bounded), Flowable can produce a backpressure exception.
- *Latest* is similar to the drop strategy but it keeps the last emitted item. Unlike drop, it ensures that at least one element is retained.
- *Error* gives an opportunity to expect no backpressure. Consequently, a *MissingBackpressureException* should be thrown if the consumer can't keep up with the source

- *Missing* states that the source will push elements without discarding or buffering. The downstream will have to deal with overflows in this case.

Listing 4.6 [65] shows classic Flowables usage example. Here the Flowable is a range from 1 till 999999999. Each emission is represented by an Item object. Assume that a constructor of class Item sets the value id corresponding to the current range value of Flowable and prints a message like "Constructing Item number x", where x is the id of the current item. During subscription to emissions the subscriber receives Item objects and prints their ids.

```java
public static void main(String[] args){
  Flowable.range(1, 999999999)
    .map(Item::new)
    .observeOn(Schedulers.io())
    .subscribe(item -> {
      sleep(50);
      System.out.println("Item emission: " + item.getId());
    });
  sleep(Long.MAX_VALUE);
}
```

Listing 4.6: The Flowable Usage Example [65]

```
Constructing Item number 1
Constructing Item number 2
Constructing Item number 3
...
Constructing Item number 127
Constructing Item number 128
Item emission: 1
Item emission: 2
Item emission: 3
...
Item emission: 98
Constructing Item number 129
Constructing Item number 130
...
Constructing Item number 224
Item emission: 99
Item emission: 100
...
```

Listing 4.7: The Flowable Usage Example – Program Output

The output of this programm is as shown in Listing 4.7. Some output lines are omitted in order to highlight the key events. It is visible that the first 128 emissions were created and pushed to the subscriber. The subscriber in its turn has received and processed 98 of

them. Afterwards, another 128 new emissions were constructed and pushed. After that the Subscriber continued to receive emissions from the place where it stopped earlier. The unprocessed emissions are queued in a special queue. In theory, the Flowable operation started by requesting 98 emissions and continued to emit 98 emissions at a time, there can be moments where operation will need to wait for another 98 emissions (because of some latency, for example). Therefore, a queue is used, where unprocessed emissions are stored. During waiting time the unprocessed emissions from queue can be processed.
Note, the last line in Listing 4.6 introduces a sleep method with parameter corresponding to the maximal value of type Long. It is added in order to let the processing chain work through all the emissions. Otherwise, due to the asynchronous nature of processing, the application exits before any significant number of emissions is processed.
Now the question arises, when to use Flowables and when Observable. To answer this question there is a need to discuss advantages and disadvantages of both methods. According to the literature (i.e. [65], [20], etc.) Flowables have massive advantages in comparison to Observables, among them we mention leaner usage of memory and prevention of *OutOfMemoryError*. On the other hand, Flowables have also big disadvantage in comparison with Observables, namely they add overhead causing slower performance of the whole program.
In general it is recommended to use Observables when it is expected to have a relatively few number of emissions over the life of Observable or/and the emissions are far apart in time. Observables are also preferable if processing is synchronous and makes limited use of concurrency.
Flowables are used if a large number of emissions is expected and/or the density of emissions is high. It is used more often in different streaming and network services.
The next chapter is devoted to Subjects in RxJava, which are able to provide functionality for both Observer and Observable (Flowable).

4.3.5 Subjects

A **Subject** in RxJava is some sort of a proxy which is able to act as an Observer and Observable. Because it is an Observer, it can subscribe to one or more Observables, and because it is an Observable, it can pass through the items it observes by re-emitting them, and it can also emit new items. There are 5 types of Subjects: Publish Subject, Replay Subject, Behaviour Subject, Async Subject, and Unicast Subject. In the following all these types will be explained [78].
The functionality provided by Subjects can be best explained using an example. Assume some professor is giving a lecture and some student arrives late in the classroom. In this case a professor represents an Observable, which emits knowledge, and a student is an Observer, who subscribes to the knowledge stream given by the professor in the classroom.
If a student entered late into the classroom, he/she just wants to listen from that point of time when he/she entered the classroom, the *Publish Subject* can be used. The Publish Subject emits all the subsequent items of the source Observable at the time of subscription.

If a student entered late into the classroom, wants to listen from the beginning of the lecture, it is a use case for *Replay Subject*. The Replay Subject emits all the items of the source Observable, regardless of when the subscriber subscribes.
If a student entered late into the classroom, wants to listen to the most recent things (not from the beginning) being taught by the professor so that he/she gets the idea of the context, the *Behaviour Subject* can be used. The Behaviour Subject emits the most recently emitted item and all the subsequent items of the source Observable when an observer subscribes to it.
If a student entered the classroom at any point in time, and wants to listen only to the last thing (and only the last thing) being taught after the class is over, there is a use case for *Async Subject*. The Async Subject only emits the last value of the source Observable (and only the last value) only after that source Observable completes.
If a student entered the classroom at any point in time after the lecture given by the professor and wants to listen to the lecture in full replay, the *Unicast Subject* can be used. The Unicast Subject will buffer all emissions it receives unlit an Observer subscribes to it, and it will release all these emissions to the Observer and clear its cache.
Erik Meijer, the creator of ReactiveX, described Subjects as the "mutable variables of reactive programming". Every developer should strive to avoid the usage of Subjects, but nevertheless the construct is present and can be used as a necessary tool to reconcile an imperative paradigm with a reactive one. According to [65] code, where Subjects are used, it is very difficult to debug.
In the next section we are going to talk about how concurrency can be implemented using RxJava.

4.4 Concurrency with RxJava

It is often stated in different forums and conversations between software developers (i.e. in Stack Overflow[7]) that reactive programming and/or RxJava is concurrent by default. This is far from the truth as reactive streams are inherently asynchronous but sequential. By default, the Observable executes on the immediate thread, which is also the thread to which the Observer subscribes itself. Additionally, the emissions produced by Observable must be passed sequentially and one at a time to each subscribed Observer. This property is often called **The Observable Contract** and it must be satisfied in every program run. This means, that Observable cannot emit to different Observers in parallel. From the Observable point of view, it is also impossible to emit some subset of emissions to one Observer and another subset to another one. To bring more clarity, observe Listing 4.8. Here an Observable will emit 2 string values ("Hello" and "World"), and 2 Observers will receive emissions and print not only the content of each emission, but also the current thread name.

[7]**Stack Overflow** is popular question-answer website for software developers created by Jeff Atwood and Joel Spolsky and first launched in 2008.

```
public static void main(String[] args){
  Observable<String> source = Observable.create(
    e -> {
      e.onNext("Hello");
      e.onNext("World");
    }
  );

  source
  .subscribe(e ->
     System.out.println("Observer 1: " + e + " Thread: " +
        Thread.currentThread.getName()));

  source
  .subscribe(e ->
     System.out.println("Observer 2: " + e + " Thread: " +
        Thread.currentThread.getName()));
}
```

Listing 4.8: Observable With 2 Observers and Default Thread Execution

Listing 4.9 demonstrates the output of the program written in Listing 4.8. It is visible from the output that both observers executed sequentially in the same main thread which was kicked off by the main method. Both Observers received and processed the full set of emissions sequentially, thus the Observable Contract is not violated.

```
Observer 1: Hello Thread: main
Observer 1: World Thread: main
Observer 2: Hello Thread: main
Observer 2: World Thread: main
```

Listing 4.9: Observable With 2 Observers and Default Thread Execution – Output

Now observe Listing 4.10, the code here is similar to the code in Listing 4.8, but the Observable has now a thread definition inside and all the emissions will be produced inside a new thread. Technically, every new subscription to an Observable will generate a new Thread in the Observable body, which will produce emissions.
Listing 4.11 demonstrates the output of the program written in Listing 4.10. The output discloses that the first Observer gets emissions from Thread-0 and the second one from Thread-1. Thus, the Observers run on different threads. This is because, when a new Observer subscribes to an Observable, it will always get emissions from a new Thread (as defined in Listing 4.10). It guarantees on the one hand, that emissions happen sequentially and asynchronously for all Observers and each Observer will receive all the emissions the Observable produces. Again, the Observable Contract is satisfied.

```
public static void main(String[] args){
  Observable<String> source = Observable.create(
    e -> {
      new Thread ( () ->
      {
        e.onNext("Hello");
        e.onNext("World");
      }
      ).start();
    }
  );

  source
  .subscribe(e ->
     System.out.println("Observer 1: " + e + " Thread: " +
        Thread.currentThread.getName()));

     source
  .subscribe(e ->
     System.out.println("Observer 2: " + e + " Thread: " +
        Thread.currentThread.getName()));
}
```

Listing 4.10: Observable With Thread Definition Inside and 2 Observers

```
Observer 1: Hello Thread: Thread-0
Observer 1: World Thread: Thread-0
Observer 2: Hello Thread: Thread-1
Observer 2: World Thread: Thread-1
```

Listing 4.11: Observable With 2 Observers and Default Thread Execution – Output

As stated above, by default, Observables are designed for non-concurrent applications, where in one single thread all emissions are broadcasted to all subscribed Observers. Having multiple threads in Observables can cause emissions to be produced faster than the Observer can consume them. Consequently, when the situation occurs, it is recommended to use *Flowables and Backpressure* instead of Observables [65], [52].
In general, it is not always necessary to create threads in order to implement concurrency in RxJava. In the next subsection we will talk about schedulers available in RxJava for creation of concurrent applications.

4.4.1 RxJava Schedulers

Schedulers in RxJava are responsible for performing operations of Observable on different threads. Schedulers give the opportunity to specify where and when to execute tasks related to the operation of an Observable chain. Many of the default scheduler implementations can be found in the Scheduler's static factory class. For a given Observer, a Scheduler provides a thread from a pull that will push the emissions from the Observable. When *onComplete()* is called, the operation will stop performing and the thread will be given back to the pool, where it can be persisted and reused by another Observer [65]. The *subscribeOn()* operator suggests to the source Observable which Scheduler to use and how to execute operations on one of its threads.
In the following all the schedulers provided by RxJava will be discussed (based on [65], [81]):

- **Computation Scheduler:** [*Schedulers.computation()*] maintains a fixed number of threads based on processor availability, making it appropriate for computational tasks (math, algorithmic tasks, complex logic). It limits by default the number of threads running in parallel to the value of available processors, for this reason it is often used for CPU-bound tasks. It uses an unbounded queue in front of every thread, so if the task is scheduled, but all cores are occupied, it will be queued.

- **I/O Scheduler:** [*Schedulers.io()*] is often used for input/output tasks such as reading/writing from/to databases, web requests, etc. These tasks in general use little CPU power and often have idle time waiting for data to be received/sent. This allows creation of threads more liberally. It maintains as many threads as there are tasks and dynamically grows the number of threads, caches them, and deletes them if they are not needed any more. In general, the number of thread creations is unbounded, for this reason it is recommended to use it with caution.

- **New Thread Scheduler:** [*Schedulers.newThread()*] creates a new thread for each Observer and then destroys the thread when it is not needed anymore. The difference to I/O Scheduler is that this Scheduler does not create a cache of threads with an intent to reuse them. This Scheduler is used if it is needed to create, use, and immediately destroy a thread, so that it does not take up memory. If this Scheduler is used for complex, long tasks in non-trivial applications, there exists a risk of creating a high volume of threads, which could crash the application.

- **Single Scheduler:** [*Schedulers.single()*] is used with the intent to run tasks sequentially on a single thread. It is used primarily to isolate non-thread safe operations to a single thread.

- **Trampoline Scheduler:** [*Schedulers.trampoline()*] is primarily used in the RxJava internal implementation. This Scheduler is like default scheduling on immediate threads, but it prevents cases of recursive scheduling, where a task schedules a task while on the same thread. Instead of producing an error, this Scheduler allows the current task to finish the execution and lets the new scheduled task be executed afterwards.

- **ExecutorService Scheduler:** [*Scheduler.from()*] is used when a developer wants to build a Scheduler from standard Java ExecutionService. It is often used when more control over thread management policies is needed. For example, if there is a need to bind the number of threads to 20 threads executed in parallel, the developer can create an ExecutorService instance with this parameter and then use the created instance in *Scheduler.from()* just as shown in Listing 4.12.

```
public static void main(String[] args){
    ExecutorService executor =
            Executors.newFixedThreadPool(20);
    Observable.just("One", "Two", "Three")
            .subscribeOn(Schedulers.from(executor))
            .doFinally(executor::shutdown)
            .subscribe(System.out::println);
}
```

Listing 4.12: Example of Observable With ExecutorServie Scheduler

Each default Scheduler is lazily instantiated. Every Scheduler from above can be disposed at any time by calling the *shutdown()* method. This method stops all the created threads and forbids new tasks from coming in. The method *start()* is used to reinitialize the Scheduler so that it can accept tasks again [65].
Another useful operation often used by applying Schedulers is *observeOn()*. This method is used to reschedule notifications emitted by the source Observable. As a consequence, the *observeOn()* operation intercepts emissions at the point where it is defined in the Observable chain and switch them to a different Scheduler going forward. It is typically used in situations when, for instance, there is a need to get some data from a database (I/O Scheduler) and use the data afterwards for computation purposes.
In RxJava there is also another way how concurrency can be achieved. The next section discusses the use of the *flatMap()* with an intent to create concurrent applications.

4.4.2 Using *flatMap()* to Achieve Concurrency

In general, *flatMap()* transforms an Observable by applying a specified function to each item emitted by the source Observable, where that function returns an Observable that itself emits items. The method then merges the emissions of these resulting Observables, emitting these merged results as its own sequence.
Since it is not possible to run emissions concurrently on a single Observable, there exists another way how we can achieve concurrent emissions of elements from the Observable using a new Observable for every new emission. Observe Listing 4.13 as an example. Here an Observable with 5 emissions is defined. Afterwards, using *flatMap()* for each emission a new Observable is created and executed concurrently with the computation

Scheduler. Listing 4.14 shows the output of the program in Listing 4.13. It is visible that each emission was returned from a new thread [65], [27].

```
public static void main(String[] args){
  Observable.just("One", "Two", "Three", "Four", "Five")
  .flatMap(e -> Observable.just(e)
      .subscribeOn(Schedulers.computation())
      .map(str -> compute(str)))
  .subscribe(System.out::println);
}

public static String compute(String emission) throws
   InterruptedException {
return "Emission: " + emission + "; Thread: " +
      Thread.currentThread().getName();
}
```

Listing 4.13: Achieving Concurrency Using flatMap() in RxJava

```
Emission: Two; Thread: RxComputationThreadPool-3
Emission: Three; Thread: RxComputationThreadPool-4
Emission: Four; Thread: RxComputationThreadPool-5
Emission: Five; Thread: RxComputationThreadPool-2
Emission: One; Thread: RxComputationThreadPool-1
```

Listing 4.14: Achieving Concurrency Using flatMap() in RxJava – Output

Despite the fact that the method of using *flatMap()* to achieve concurrency exists, it is not recommended to use it because of some overhead for creation of new Observables for every new emission.
In the next chapter the concurrent application scenario will be presented and the implementation of this scenario using threads and RxJava will be discussed.

CHAPTER 5

Concurrent System Scenario and Implementation

It's important to have a sound idea, but the really important thing is the implementation.

— Wilbur Ross.

In this chapter the practical aspects of programming with threads and reactive programming concepts will be covered.

5.1 Concurrent System Scenario

In this section we describe a chat application which we will call *ChatApp*. This application serves as the reference for evaluating the implementations using Threads and RxJava.
In general, any chat application is a software system which enables sharing of data between people (*clients*) who are distanced apart. Such applications are often called *social media* [33].
As a rule, every chat application allows 2 forms of interaction for their users: users are able to compose and send messages, and they are also able to receive messages from other users. The content of a message can be different – from simple text to video or audio files. Some applications require registration with personal data of a user and creation of accounts, some allow the usage without mandatory registration.
Chats gained rather big popularity recently because of rush development of internet technologies and rather cheap (and in majority of countries not restricted) access to them. Another major factor which influences popularity of such applications is affordable gadgets and devices which are able to connect to internet and store these applications [33].

As a rule, chat applications are implemented using the *client-server paradigm*. This paradigm divides software into two categories: clients and servers. A client is software which initiates the connection to the server and sends requests, whereas a server is a software application that listens for connections and processes requests [64]. In general, a *server process* can be explained as a service which *client processes* want to use in order to solve their tasks. In case of the *ChatApp* application, the server is dedicated to manage access to the chat room and to distribute messages incoming from given clients to other clients. The server is also responsible for the login/logout logic, database connections, etc., providing network services and resources for clients in order to achieve their satisfaction. In *ChatApp*, client processes establish connections to the server process and use its resources and functionality in order to communicate with each other.
The communication between clients and server usually follows strict protocols. *ChatApp* will use Java Sockets[1] and TCP[2] in order to establish and maintain the connection between clients and server. In Section 5.2 system components and their interactions will be disclosed in more detail.
In the next subsection we will define the concurrent scenario.

5.1.1 Scenario Description

As described earlier *ChatApp* is the client-server software application which enables the communication between clients using the resources and logic the server provides. *ChatApp* will be implemented two times: using threads, and using RxJava in order to compare both implementations and answer research questions postulated earlier in this thesis. There is also a need to cover different possible implementations of threads: *stateful* and *stateless* implementations for the purpose of performance comparison with equivalent implementations in RxJava. During experiments we will change some classes in order to make them stateless or stateful. The general scenario for both implementations is presented below.
ChatApp is a chat application where each user (client) can create an account and be authorized. In order to create an account a potential user will need to enter his/her firstname, lastname, password, and nickname. The last string value must be unique for all users and will identify the user in the chat. After successful account creation, users can login using their nickname and password. It will be possible to change all personal information, except the nickname.
Authorized users will be able to send messages either to one other authorized user (unicast), or to a group of authorized users (blockcast), or to all users (broadcast). Authorized users will be able to receive unicast, blockcast, and broadcast messages.
Every message in ChatApp is a text message. The authorized sender of a text message can apply diverse restrictions to further message editing. The sender can qualify a message as *final*. All final messages cannot be edited after sending by everyone including the

[1]**Java Socket** is an endpoint of a two-way communication link between two programs running on the network.

[2]**Transmission Control Protocol (TCP)** provides reliable, ordered, and error-checked delivery of a stream of bytes between applications which communicate over the network.

sender. *Private* messages will be editable only for the sender, and *public* messages will be open for editing by every authorized user. These features are added for evaluation of performance and memory consumption by accessing critical sections through concurrent clients implemented with both strategies.
In order to perform the evaluation, test data will be created. Performance and memory consumption of two implementations of the scenario will be compared several times using different numbers of generated clients and different scenarios for sending and editing messages.
In the next section the design of the *ChatApp* software application will be explained.

5.2 Design of *ChatApp* Application

In this section the design of the *ChatApp* software application will be described and discussed. First, the system architecture will be presented, afterwards the focus will be on the database setup, and, in conclusion, the usage and the backend processes which will happen after submitting one or another request will be discussed.

5.2.1 System Architecture

The chat application adheres to the client-server paradigm, as stated above. Figure 5.1 depicts the idea behind the client-server paradigm.

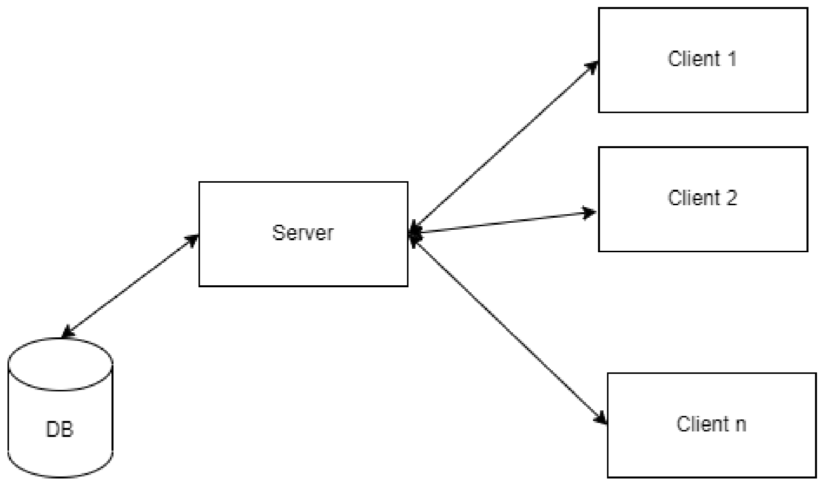

Figure 5.1: Client-Server Application.

The server is a software application that provides services, processes requests from clients, creates and computes the responses to clients and manages shared resources. It listens for incoming requests from clients, processes the requests, and sends appropriate responses to clients [64]. In *ChatApp*, server is responsible for routing messages between clients,

storing chat history to the database, managing user authentication and authorization, and ensuring that messages reach their intended recipients.
The client is a software application that initiates requests to the server and interacts with the user [64]. In *ChatApp*, a client will act as a user interface, allowing users to input their data, send messages, and receive them. Figure 5.1 depicts the client-server architecture used for *ChatApp* in general. It is visible, that server interacts with all the connected clients and with the database in order to enable storing the chat history and user authentication and authorization.
For the communication between clients and the server Java **Sockets** are used. Socket is a two-way communication link between two independent programs running on a network. The socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent. Sockets in Java are implemented via **java.net.Socket** and **java.net.ServerSocket** [75].
In *ChatApp*, the server application will create a socket, listen on a specific IP address[3] and port for incoming client connections. The client application will also create a socket and will specify the IP address and port to which it wants to connect. Consequently, each client will specify the IP address and port of a chat application server. When the server will receive a connection request from the client, it will accept the request creating a new socket that will be dedicated for communication with that specific client. This socket is called *client socket*. The server will continue listening for additional connection requests from other clients.
In *ChatApp*, clients and server will use TCP as communication protocol for the exchange of messages. TCP will ensure connection-oriented, reliable, and stream-oriented data transmission between communication partners [75].
The termination of a connection between a single client and the server can happen via an appropriate command the client can send. If the server terminates its work, all the clients will not be able to communicate any more. In case the server will reboot, all the clients need to reboot in order to be able to communicate further.
The server will also have a connection to a database, where the login data of all clients will be stored, and also all messages exchanged between clients will be persisted. The database itself will be a shared resource for clients, because, on the one hand, they will need to access it concurrently in order to register or login/logout from the application, and, on the other hand, all the messages clients send and modify will be concurrently stored in the database.
In the next subsection the database setup will be presented and discussed.

5.2.2 Database Setup

The *ChatApp* software application will have a database where persons (the users of the application) and text messages (written and sent by the users) will be stored. Figure 5.2 depicts the entity-relationship diagram of a database that will be created.

[3]**IP address** an *Internet Protocol* address is a unique string of characters which identifies each computer over network.

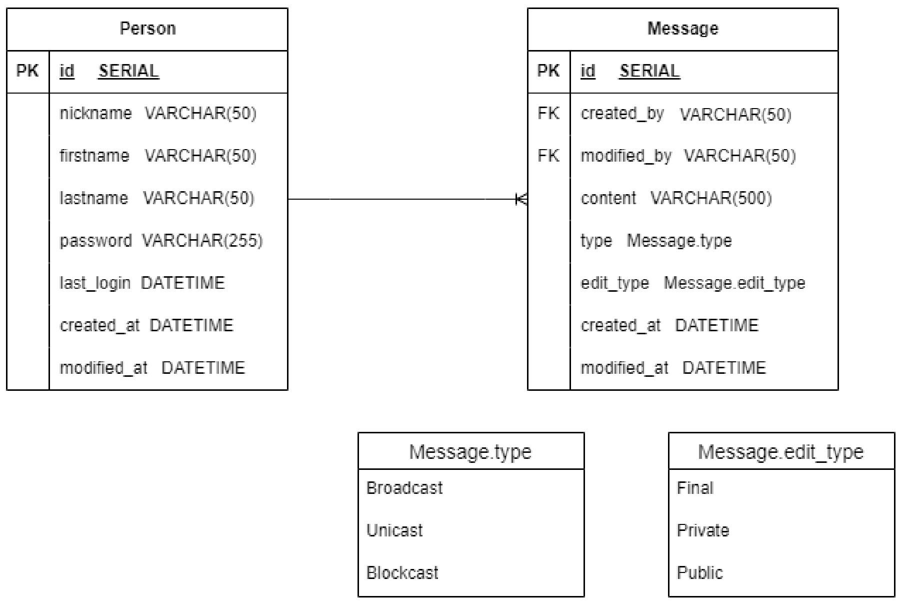

Figure 5.2: Entity-Relationship Diagram for the Database Setup of *ChatApp*

The *Person* table will have following attributes:

- **id** the private key of the table of type *SERIAL*[4];
- **nickname** is some selected name of a person which cannot be null;
- **firstname** is a first name of a person;
- **lastname** is a last name of a person;
- **password** is a password of a person[5];
- **last_login** is a time stamp showing when this person was last seen online in the *ChatApp*;
- **created_at** is a time stamp showing when this person was created;
- **modified_at** is a time stamp showing when this person was modified (for example, first name or last name change).

The *Message* table will consist of following attributes:

- **id** the private key of the table of type *SERIAL*

[4]SERIAL is an alias for BIGINT UNSIGNED NOT NULL AUTOINCREMENT UNIQUE used primarily for private keys in entity-relationship databases.

[5]Password will be stored securely using the SHA256 hashing algorithm

- **created_by** is a foreign key which references the person who wrote and sent the message, cannot be null;

- **modified_by** is a foreign key which references the person who last modified the message, can be null if this message is final or was not modified;

- **content** is the text of the message;

- **type** is a type of the message which can be one of the following: broadcast, unicast or blockcast. Message is of type *broadcast* if it is sent to all persons who are now online; the message has type *unicast* if it is sent to one single person (who is online); the message is of type *blockcast* if it is sent to every person online except those, specified by the sender;

- **edit_type** is a type of the message from the editing permission point of view, the value can be one of the following: final, private, public. The message is of edit type *final* if it is not editable; the message has edit type *private* if it can be edited only by the sender; the message is of edit type *public* if it can be edited by everyone;

- **created_at** is a time stamp when the message was created;

- **modified_at** is a time stamp when the message was modified.

In the next subsection the usage requirements will be presented and discussed.

5.2.3 *ChatApp* Use Cases and Utilization

Figure 5.3 depicts the Use Case UML Diagram[6] which represents the functionality of *ChatApp* utilizing the so-called use cases. In this subsection the use cases and some utilization agreements will be presented.

Table 5.1 explains each use case from the Figure 5.3 and contains also notation agreements for message compositions.

[6]**Use Case Diagram** in the Unified Modelling Language (UML) is a behavioural diagram which graphically depicts user's possible interactions with a system.

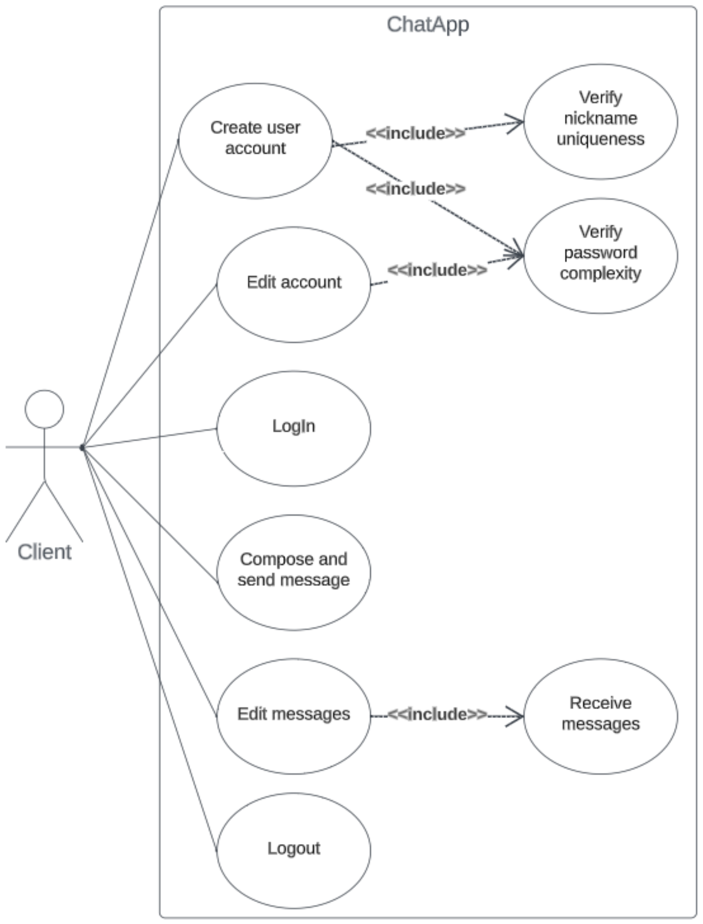

Figure 5.3: Use Case Diagram of *ChatApp* Software Application

Use Case	**Description**
Create User Account	User will be able to create an account. For this purpose the user will need to enter his/her nickname, first name, last name, and password. The nickname must be unique because it will be used in *ChatApp* as identification of a user, for this reason the nickname uniqueness verification will be done internally. Another internal function will verify the password complexity. The password should be at least 6 characters long, and consist of letters, numbers, and at least 1 special character. If one or both of these verifications will fail, the user will have an opportunity to re-enter nickname, password, or both of them.

Edit Account	The user will be able to edit personal information like firstname or password. The only not editable field is nickname. In case of a password change, the user must adhere the same rules described above, and the password verification procedure will start automatically before saving. If the password validation fails, the user will get an opportunity to reenter the password.
Login	In order to login the user will need to enter his/her nickname and password. If the combination will be found in the database, the login will be successful. Otherwise, the user will get an opportunity to enter the credentials once more.
Compose and send messages	The user will be able to compose and send messages to other registered users. In order to send a message, the user will need to specify by what type (broadcast, blockcast, or unicast) and what edit type (final, private, public) the message should be sent. By default all messages are sent using broadcast strategy and have final edit type. The user will need to adhere to the following message-prefix pattern: [[type][edit_type]:]<message> type := @<receiver> | !<receivers> edit_type := /private | /public The notation above is as following: square brackets indicate an optional use of content enclosed in these brackets. Everything enclosed in <...> symbols indicates a variable, which the user must set before sending the message. The *type* is defined by two indicators: *@<receiver>* means a unicast message with specified receiver; *!<receivers>* means blockcast message to all the users except the users from the *receivers* list. The edit type can be specified after the type by use of self-explanatory words with slash as prefix. By default, all messages are sent as broadcast final messages, in this case no separate type and edit type inputs are needed. For example, assume the wish to send the following message: *Hello world!*. By default, there are no prefixes in order to send broadcast final messages. Assume the wish to broadcast a private message. In this case the following notation can be used: /private:Hello World! In case the wish is to send as public unicast message to *user_1*, the following string will be expected: @user_1/public:Hello World!
Edit Message	The user will be able to edit all messages with edit type *public* and also own messages with edit type *private*. The user will see the id and the edit type of the message prefixed to the message content.

Logout	The user will be able to logout using a *quit* command.

Table 5.1: Use Cases Description for *ChatApp*

In the next subsection the technical stack and *ChatApp* test and execution environment will be defined.

5.2.4 Technical Stack

For the development of *ChatApp* the following technologies will be used:

- Programming Language: Java 23.0.2
- Reactive Programming Library: RxJava 3.1.9
- Database: PostgreSql 15.2-2
- Integrated Development Environment (IDE): IntelliJ IDEA 2024.3.4 Ultimate Edition.

5.3 Implementation Using Threads

This section will disclose the details and discuss the challenging parts of the concurrent scenario implementation using Java threads. As already stated earlier, in order to conduct the profound analysis and comparisson of performance and memory consumption between implementations using threads and RxJava, there is a need to implement both: stateless and stateful objects. In the following subsections implementation details for both strategies will be presented.

5.3.1 Stateless Implementation

As stated in Chapter 3 of this theses, stateless objects are considered thread safe because they do not rely on the external state. Creating a stateless implementation involves avoidance of shared mutable states and use of local variables and/or immutable data structures.
In the *ChatApp*, it is essential to design the class methods such that they operate solely on the provided input parameters and produce outputs without introducing side effects. Additionally, the system can use shared memory only for reading purposes and must avoid scenarios where the execution of one thread depends on the state or behaviour of another thread.
The literature ([24], [72]) advocates for the use of *immutable classes*, which remain unalterable after instantiation, as a foundational approach to implementing stateless classes. To achieve immutability, it is advised to declare all fields as *final* and refrain from providing setter methods for modifying internal variables. This concept is also

implemented for the stateless implementation of *ChatApp*.
To ensure the complete elimination of writing to a shared memory, we will refrain from saving any data in the database. Instead, we will utilize predefined messages and testing scenarios for all forthcoming experiments.
In the stateless implementation of *ChatApp*, all class methods exclusively utilize local variables or immutable fields to ensure safe concurrent invocation by multiple threads. Moreover, for methods comprising multiple steps, it is imperative to guarantee the atomicity of these operations to prevent the occurrence of race conditions.
In the next subsection the stateful implementation of the concurrent scenario for *ChatApp* will be discussed.

5.3.2 Stateful Implementation

This subsection covers, on one hand, all strategies applied to achieve thread safety and liveness in *ChatApp*, and on the other hand, problems, and solutions we met during the implementation of the *ChatApp* using threads.
As stated in Section 5.2.1 *ChatApp* is a client-server application which requires the implementation of two separate programs: the client and the server. For the client application, we use the socket class and initiate the connection to a server bypassing the IP address and port number. We use the *Scanner* to get the input from the user and send the data to the server using the *PrintWriter* object. For the server implementation, we use a ServerSocket class that binds to a specific port number. A server socket listens to the connection with the client and accepts it, in case it is present. A server thread is then initiated and started for each client. All the threads are added to the immutable list so that multiple clients can communicate with each other.
In the server thread, we receive sockets and a list of active threads from the main class using an constructor. When we start the thread from main, the run method is called. The *BufferedReader* helps us to receive information from the client. Any information that the server wants to send is sent using *PrintWriter*.
The server application implements all the features described in Section 5.2.3. Some features like account creation, login, or message saving for further editing possibilities require a database connection. Using a database in a multithreaded server application requires careful consideration to ensure proper handling of concurrent access and data consistency. In the following subsection the considerations regarding database usage in multithreaded environments will be presented.

Database Use in Multithreaded Server Application

When designing a multithreaded server application that interacts with a database, it is important to handle database operations in a thread-safe manner to prevent data corruption and ensure proper concurrency. The sequence of steps involved in a typical database connection life cycle includes:

- Opening a connection to a database using the database driver;

- Opening a TCP socket for data input or output;
- Reading or writing data over the socket;
- Closing the connection;
- Closing the socket;

It becomes evident that operations bound with database access involve several steps in a computer program, and as such, should be reduced to a minimum in every possible use case.
The first problem which arises is how to use the database connection in a multithreaded server in such a way that both conditions are satisfied:

1. Database operations are thread-safe.
2. Reduce the overhead in performing database connections and read or write operations.

Literature like [69], [26], [64], etc. introduce different solutions to the problem above: *Connection Pooling* or *thread-local database connections*.
Connection Pooling is a data access pattern, which at the most basic level is a database connection cache [26]. The idea is simple, all the database connections are placed in the so-called Connection Pool and are taken by threads, if they are needed, or released if they are not in use anymore. Each connection is then reusable for other threads. Connection pools can improve performance by reusing existing database connections instead of repeatedly opening and closing new connections; they are resource efficient because they limit the number of concurrent connections to the database; sometimes they also guarantee faster response times for database operations since the overhead of establishing a new connection is eliminated. On the other hand, connection pools may lead to connection leaks, where connections are not properly released back to the pool. This can result in resource exhaustion and degrade system performance over time. Furthermore, implementing and managing connection pooling can add more complexity to the application code. There are some well known full-featured connection pooling JDBC frameworks available on the market: *Apache Commons DBCP Component*, *HikariCP*, etc. [45].
Tread-local database connections is a situation where each thread has its own database connection. It is ensured that each thread operates on its local database connection. Among advantages it is worth to mention the simplicity compared with the management of connections pools; thread-local database connections provide a high level of isolation between threads, which minimizes the likelihood of data corruption. Each thread can manage its transactions independently without affecting other threads, providing more control over the transactional boundaries. Among disadvantages there are: increased resource consumption, as each thread maintains its own connection; connection

limits provided by database vendors; limited connection reuse [69] [26].
In order to remain simple and provide guarantees of thread safety while connecting to the database, the decision to use thread-local database connections in *ChatApp* was taken.
The positions in code where writing to the database or also reading from the database takes place become the critical sections for threads to access. Some sequence of events must be defined in order to prevent data corruption or data unavailability. On the one hand, it must not happen that some thread begins to write to the same place, where another thread is already editing at the same time. On the other hand, one thread must not begin to read data which another thread has not yet stored. For example, if two users want to edit the same message, the first user must first save his/her editing before the other user saves his/her changes. Another example can be: one user must be able to send a message to another user only after another user is successfully registered and logged in.
To be able to manage situations described above and to maintain data consistency there is a need to implement proper transaction management and synchronization between threads. For example, no two or more threads can access the procedure of writing to the database simultaneously, no thread can read before writing is complete, etc.
In addition, the database itself provides so-called **isolation levels** which can handle some unwanted concurrent side effects: dirty read, nonrepeatable read, phantom read, serialization anomaly. We speak about *dirty read* if one transaction is reading data which is being updated by an uncommitted transaction. *Nonrepeatable read* occurs when a transaction re-reads some data and finds that it was changed by some other transaction that has already committed between the two reads of the initial transaction. When a transaction re-executes a read query which satisfies a certain search criteria and gets a result set which is different from when the query was executed the first time, we talk about *phantom reads*. Finally, when the result of successfully committing a group of concurrent transactions is different from all possible combinations of running those transactions one after the other, then we know that *serialization anomaly* has been occurred [21] [46] [32].
Isolation is the ability of the transactions to operate concurrently without causing any kind of a side effect. Database theory defines 4 isolation levels: read uncommitted, read committed, repeatable read, serializable. Table 5.2 inspired by [83], [71], and [32] shows how different isolation levels make some concurrent side effects possible or not possible on the database site.

According to [71] PostgreSQL implements only three of four isolation levels: the first level from Table 5.2 - read uncommited - is not present among implementations. The default isolation level in PostgreSQL is set to *read committed*. If the isolation level is *repeatable read*, then *phantom read* is not possible in PostgreSQL [71].
In *ChatApp*, our primary objective was to ensure thread safety by employing traditional Java thread-locking mechanisms. To minimize side effects associated with database access, standard Java synchronization techniques were utilized. To prevent multiple redundant database access shielding, the isolation level was configured to the default value, *read committed*.

Isolation Level	**Dirty read**	**Nonrepeatable Read**	**Phantom Read**	**Serialization Anomaly**
Read un-committed	Possible	Possible	Possible	Possible
Read com-mited	Not possible	Possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible	Possible
Serializable	Not possible	Not possible	Not possible	Not possible

Table 5.2: Transaction Isolation Level and Possibilities for Side Effects in Database Theory.

Striving for Thread Safety and Liveness

In Chapter 3 we talked about safety and liveness hazards that can occur if, for example, threads are not properly synchronized between each other, or one thread blocks other threads from accessing the critical section, etc. Section 3.6 of this theses defines design rules which can be applied to the code of software applications in order to guarantee the absence of diverse hazards. Taking into account everything said previously, in this subsection we will discuss the concrete implementation of *ChatApp* and describe what safety and liveness hazards were possible and how they were mitigated during the implementation.
In *ChatApp* there are several crucial critical sections where either safety or liveness hazards, or even both of them can potentially occur. For the purposes of simplicity of describing and reading this section, the whole code written for *ChatApp* will be divided into logical blocks. For example, we will call all lines of code responsible for broadcasting a message the *message broadcast block*. In an analogue way, all lines of code responsible for message editing will be called *message edit block*.
The *user registration and login block* contains several input fields, where a user has to input his/her credentials and, in case of registration, the information will be stored in the database, or, in case of login, the information must be read from the database and compared to the input data made by the user. Afterwards the decision if the user input is valid for login must be taken. If the user is qualified for login, the online status of the user will be updated in the database and set to true. It is visible, that neither the registration nor the login procedure is atomic and contains numerous steps. Moreover, the steps are to be performed in strict logical order, meaning, the user must first input several lines containing his/her nickname, password, etc., (also in the given order) and only after all the input is typed in, the information can either be stored in the database or verified using the database.
In a concurrent environment user, registration and login can pose several problems. First of all, the consistency of user input data should be established. It can be violated due, for example, accessing the section where database saving of users login information happens by multiple threads. Because of a non-deterministic thread execution order, it cannot

be guaranteed that all data will be saved in the right cells in the database. Even more, it is crucial to lock input/output sections during the intake of user data, because the order and the integrity of data is important. All problems described above provoke the appearance of race conditions and introduce thread safety hazards. In order to mitigate this hazards, a locking mechanism around code sections described above is used. In addition, all database communication and text parsing external libraries used in *ChatApp* are thread safe (following rules 3, 4, and 5 from Section 3.6 of this thesis).
After solving thread safety problems from above another problem predicted in Rule 6 from Section 3.6 of this thesis occurs. Assume one thread where the user makes no input, or where the database does not acknowledge if saving is ready. In this case this thread, which is currently inside the critical section, will block all other threads from accessing the critical section and provoke the starvation, which is a liveness hazard. The solution used in *ChatApp* for this case is the introduction of a timer which measures how long the current thread remains idle in the critical section. If the threshold is reached (for instance, 90 seconds), the thread closes automatically and the next thread in a queue can access the critical section.
The next block worth to discuss is the message sending and editing block which itself is logically divided in 4 sub-blocks for each feature provided for messages in *ChatApp*: unicast sending, blockcast sending, broadcast sending, and message editing. During the process of message writing each user can decide how to send the current message and what editing rights to give to the message. Once the decisions are taken, the message must be sent according to the user wishes. Each messages sending includes not only the logic for the correct kind of sending, but also for storing the message content (and other meta-data bound to the concrete message) in the database. In case of message editing, the following steps are made: (1) message retrieval from the database using the message id; (2) validation if the message can be edited by the specific user according to the constraints presented in Section 5.2.3 of this thesis; (3) if the message is qualified for being edited by the particular user, the new content and new meta-data will be saved in the database and the message will be sent again using the same strategy as the previous message; (4) if the validation's verdict is that the message is not available for editing, the respective user will get a notification about that and the process ends. It is again clear that message sending and also message editing are not atomic operations and are composed of numerous different steps including writing and reading from the database with specific logic for message categorization and parsing in-between.
Here again data corruption due to improper thread synchronization is possible. If several threads try to write several new messages into a database, there is no guarantee that data will be written in a consistent way, which poses a thread safety hazard. Locking the critical section, namely the section where the database writing procedure takes place, can help to avoid this hazard.
Another special case which can theoretically happen is as follows: if one thread begins to write the new message into a database, but another thread wants already to read the message (for example, for editing purposes). This case leads to a so-called *dirty read*. For avoidance of such cases the isolation level of the database itself was set to *read committed*

(for more details, refer the previous sub-chapter).
There are also several edge cases possible, where appropriate error handling must be introduced. To understand the nature of this edge cases, there is a need to describe the principle how message sending using different strategies works.
First of all, all currently active threads are stored in a thread safe immutable list. The addition of each new thread to the list happens when a user logs in to the system. Thus, this list represents all users, which are currently online. As soon as the user logs out from the system, the thread will disappear from the list. The immutable list of threads is used during each message sending procedure.
In order to send a broadcast message from the current thread, the process goes through all threads from the list of active threads one by one and with use of *ObjectOutputStream* writes a message for the user. The *ObjectOutputStream* is in general not thread safe, for this reason synchronization while sending is needed. Similar synchronization happens in the process of blockcast sending, where the message is printed in every thread from the list except one thread. For unicast message sending we search one thread from the list and write the message into the output stream of that thread. Before sending each message is saved in the database.
Proper error handling is important in cases when a message is, for example, sent to a specific user, who in the moment of sending is already logged out from the system. In such a case the message cannot be sent and if it is already saved in the database, a rollback must happen.
One more scenario can happen when there is a communication problem (for instance, server delay, etc.) between, for example, thread and database, or between threads themselves, and message sending is for any reason not possible. In this case among other possible solutions, one solution is to resend a message after some time. If this solution is implemented, then a livelock can occur. Here threads continuously resend the messages to each other and they are not deliverable. So, threads are working (not a deadlock), but with no visible result. In order to avoid livelocks (liveness hazard) there must be a braking mechanism for resends, for instance, stop the thread after the third resend. In *ChatApp* in case of communication problems, a thread will remain idle and will automatically end after some period of time.
Another important feature of *ChatApp* is message editing. To perform message editing several steps should be done by the thread: first of all the message should be retrieved from the database and validated if the message can be edited by a particular user. In a second step, the new content of the message along with an update timestamp must be stored in the database. In the last step the new content of the messages must be published according to the decisions taken by the original message.
If two or more threads begin to edit the same message simultaneously and want to save the new content in the same time and if the concurrency conflict in this case is not resolved properly, a deadlock can occur. For example, if thread A and thread B want to save the modified message to the database, they can come to the conflict, because both of them will try to lock the database access. Threads A and B enter a state where they are both waiting for the other to release the lock on the message before they can proceed

with saving.
To prevent the deadlock of such kind a locking mechanism for message editing must be implemented, where only one single thread at a time is able to edit one single message. In addition a user notification can be also sent if some other user is already in an editing process of the desired message.
As we have observed, in order to provide enough level of thread safety and liveness, in many cases appropriate locking (synchronisation) is needed. Literature, for example, [76], [24] propose an alternative approach for threads locking, which sometimes is called the non-blocking approach. In the next subsection this approach will be compared to the traditional one.

Blocking vs. Non-Blocking Server

Previously, in Section 3.3, we talked about different methods how to ensure thread safety in Java in concurrent applications. Among them we discussed *atomic* and *volatile* variables. Much of recent research in the thread safety area concentrated on the so-called *non-blocking* algorithms, which use low-level atomic machine instructions instead of locking to guarantee thread safety. Such algorithms are called non-blocking because they pose an alternative to the blocking locking or synchronization approaches. In this subsection we want to compare both blocking and non-blocking approaches.
Before diving into non-blocking strategies, there is a need to provide sufficient motivation why it is needed and what are the disadvantages of using well-known Java locks. First of all, the standard locking mechanism is *blocking*, which means, that each thread inside a critical section blocks all other threads from accessing the critical section. For this reason, in Section 3.6, we provided a rule number 6, where we stated, that the amount of locking (or synchronization) must be kept as minimal as possible to avoid liveness and performance problems.
Another problem occurs if multiple threads try to lock a resource at the same time and some thread is suspended with an intent to run later. In this case JVM scheduling will insert the previously suspended and now ready to resume thread into the end of the waiting queue. The thread will wait till all other threads from the queue will be ready. Suspending and resuming a thread is in general very expensive and entails a lengthy interruption [24] [68].
If one thread is waiting for access to a critical section, it is not able to do anything else, and if a thread that is holding a lock is delayed, than no other thread in a waiting queue can make progress. "Even ignoring these hazards, locking is simply a heavyweight mechanism for fine-grained operations such as incrementing a counter. It would be nice to have a finer-grained technique for managing contention between threads." [24]
There are several approaches for non-blocking threads synchronization. Some of them are based on hardware support for concurrency, others utilize atomic and volatile variables.

Hardware support for concurrency. Starting from Java 5.0 some operations designed for processors which support multiprocessor operations are available for developers to use in their code. Today's processors mostly have some form of atomic read-modify-write

instruction like: compare-and-swap, load-linked/store-conditional, etc. Based on this operations some algorithms can be developed, which are then called non-blocking algorithms.
For example, the compare-and-swap operation has three operands: memory location, old value that is already written to memory location, new value which must be saved to the given memory location. The compare-and-swap operation first compares if the inserted old value corresponds the true value in memory, and if it is so, conducts an update with a new value, otherwise, no update happens and a notification about the wrong old value is sent.
Assuming the situation where multiple threads want to update the value simultaneously using the compare-and-swap operation, certainly only one thread will win the race and update the value. All other threads will be notified that the update is already done and then the decision can be made: either each other thread retries the update procedure, or does nothing. Since the operation is not blocking, no thread is blocked.
There are also significant disadvantages for using hardware support for concurrency. For instance, the performance of hardware based operations varies across processors, usage of such kind of operations can also be more complicated for developers, because there is a need to implement thread strategies in case the thread does not manage to conduct the operation. If the strategy proposed is a retry strategy, then a liveness problem can occur, where threads will constantly retry the operation without proceeding further. Thus, using such non-blocking operations must be accompanied with solid software design and planning [24].

Atomic and volatile variables. In Chapter 3 we have already discussed the notion of atomic and volatile variables. The atomic variable classes provide a generalization of volatile variables to support atomic conditional read-modify-write operations. This allows to regard these classes as an alternative to traditional locking mechanisms.
There are twelve atomic variable classes in Java. These classes are divided into 4 groups: scalars, fields, arrays, and compound variables. All these classes support the compare-and-swap method described above; *AtomicInterger* and *AtomicLong* support non-locking arithmetic as well. Interesting is also that the scalar classes extend *Number* and do not extend the primitive classes such as for example *Integer* or *Long*. It is because the wrapper classes are immutable, while atomic variable classes are mutable [24].

A combination of both strategies: hardware support and utilization of atomic variable classes, the so-called non-blocking algorithms, can be designed and implemented. Unfortunately, it is stated in literature ([58], [73], [24]) that the development of such algorithms is more complicated and time consuming. The complexity of code increases, the maintenance possibilities decrease. Nevertheless, the advantages are: no explicit locking of threads, no deadlocks, and better possibilities for scalability.
The choice between blocking and non-blocking synchronization depends on the specific requirements and characteristics of the application, as well as the trade-offs between simplicity and performance. It is recommended to use a blocking strategy if the applica-

tion logic is simple enough to be implemented using locks; if the deadlock risk can be mitigated or managed; if a predictable execution order is important. On the other hand it is recommended to use non-blocking synchronization if scalability is very important; if deadlocks avoidance is of high priority; if the complexity of non-blocking synchronization is manageable for the given application.
Because of priorities like simplicity, clarity and robustness of *ChatApp* and manageable deadlock risk the *ChatApp* is implemented with use of a traditional locking mechanism. The idea and motivation of this thesis was also to compare a traditional threads implementation with a reactive implementation. Locking mechanisms are established best practices in concurrency handling in many programming languages [24].
In the next section we will discuss implementation of *ChatApp* using RxJava.

5.4 Implementation Using RxJava

This section is devoted to implementation challenges of a concurrent scenario developed using reactive programming (RxJava). During the implementation of the client-server *ChatApp* using RxJava several challenging problems had to be solved. These problems and their solutions will be discussed in the next subsections.

Reactive Implementation for Client-Server Paradigm

The first problem which arrives immediately after starting the implementation is how to implement multithreading using sockets in a reactive world. From Chapter 4.4 we know that by default the observable executes on the immediate thread, which is also the thread to which the observer subscribes. Concurrency is achieved by using schedulers on the observer side. After all the emissions are consumed and all the necessary computations are done, the program terminates. Unlike an implementation using threads, where the server waits for client input and the client goes into a standby mode and waits for messages from the server, the program implemented using RxJava terminates immediately after the observable has emitted all its emissions.
Another important issue is the modifiability of defined observables. *ChatApp* is an application where during its run different users can login and logout, can delete existing accounts and create new ones, etc. These actions can lead to some sort of dynamic behaviour, where some user, for example, can join the chat after other users joined it, or one user can decide to leave the chat, etc. In a traditional implementation using threads we used a dynamic list of active threads, where each thread is a representation of a user in a system. In a reactive world we need another mechanism to implement such a dynamic procedure.
All the limitations described above are solved with **WebSockets**. WebSocket is a protocol that provides a full-duplex communication channel over a single, long-lived TCP connection. It allows for real-time, low-latency communication between a client and a server [42]. While sockets are a more general concept for communication endpoints, WebSocket is a specific protocol built on top of sockets, tailored for efficient and real-

time communication between web clients and servers. Once a WebSocket connection is established, the connection stays open until the client or server decides to close this connection. WebSocket communication can be enhanced by integrating it with reactive programming libraries like RxJava [42] [18] [34] [25].
When creating the connection to the WebSocket, the library requires us to write a listener that takes care of processing events like opening/closing a WebSocket, receiving messages, etc. Just as in Java Sockets we implement a client and a server for the *ChatApp*. A WebSocket client possesses the following predefined methods, which are called when one of these events occur:

- *onOpen()* for reacting when a WebSocket connection is opened;
- *onMessage()* for reacting when a WebSocket receives a message;
- *onClosing()* for reacting when a WebSocket is just before closing;
- *onClose()* for reacting when a WebSocket is closing;
- *onFailure()* for reacting when some kind of error in the connection occurs.

The WebSocket Server implementation must override the following methods, which are also, as for the client case, called when special events occur:

- *onOpen()* when a client connects to the server.
- *onMessage()* when a client sends a message to the server.
- *onError()* when during the communication with clients any error event occurs.
- *onClose()* when a particular client disconnects from the server.

The first limitation described above is solved by the design of WebSockets, where the connection between server and clients remains open and waits for events. The second limitation from above is also solved by introducing *sessions*. Each client connected to the server receives its session with defined id. A session is stored as an AtomicReference object. An example of session creation while connecting the client to the server is shown in Listing 5.1.
Here (Listing 5.1), method *getConnection()*, which is called in the *onOpen()* method not present in the listing, returns an opened session if *connect()* sets the session. Method *connect()* tries to register the client by the server and to get a valid session for a client. This process is called *client deployment*. If during the deployment the client gets no open session, the process will terminate with an exception. The registration of the client and getting of the session is implemented in the *webSocketContainer.connectToServer()* method, which takes *clientSocket* and *URI* as input parameters and returns a valid session or throws an exception if session could not be established.

```
....

import javax.websocket.Session;
import java.util.concurrent.atomic.AtomicReference;

...

private AtomicReference<Session> session = new AtomicReference<>();

...

private Optional<Session> getConnection(){
  try{
    this.connect();
    return
        Optional.ofNullable(session.get()).filter(Session::isOpen);
  } catch(ReactiveWebSocketException e) {
    // Handle Exception
  }
}

private void connect() throws ReactiveWebSocketException {
  if (session.get() == null) {
    try {
      session.set(webSocketContainer.connectToServer(clientSocket,
          uri));
    } catch (final DeploymentException | IOException e) {
      throw new ReactiveWebSocketException("Could not connect to
          websocket server!", e);
    }
  }
}
```

Listing 5.1: Clients Connects to Server Using WebSockets

On the server side, we implement *onOpen()* so that we store each new opened session to a hash set just as in Listing 5.2

```
private final Collection<Session> clientSessions = new HashSet<>();

@OnOpen
public void onOpen(final Session session) {
  clientSessions.add(session);
}
```

Listing 5.2: WebSocket Server Stores Client Sessions

The software program implemented up to this point conquered limitations like waiting for incoming messages and storing open and active sessions. Our goal is to create web socket clients and a web socket server, which will provide an observable for observing received messages and also be observed for sending messages to a client or server. But before we jump to these details, our next feature in the list of features from Subsection 5.2 is to enable registration and login/logout to and from the program respectively. Besides the logic for authorization, this feature also includes the necessity of saving and retrieving the data using a relational database. The connection to the database and also read and write operations will be implemented in a reactive way. In the next subsection the details for a reactive database connection will be discussed.

Reactive Database Connection and Operations

The database connection implementation using RxJava can be facilitated with help of the **VertX PostgreSQL Client** persistence library[7].

```
PgConnectOptions connectOptions = new PgConnectOptions()
  .setPort(8080)
  .setHost("localhost")
  .setDatabase("chatApp-db")
  .setUser("chatApp-user")
  .setPassword("secret");

PoolOptions poolOptions = new PoolOptions()
  .setMaxSize(10);

SqlClient client = PgBuilder
  .client()
  .with(poolOptions)
  .connectingTo(connectOptions)
  .build();

client
  .query("SELECT * FROM Person WHERE id=123")
  .execute()
  .onComplete(ar -> {
        if (ar.succeeded()) {
          RowSet<Row> result = ar.result();
          // end login with success
        } else {
          // begin registration functions
     }
```

Listing 5.3: VertX PostgreSQL Client Usage

[7]VertX PostgreSQL Client Persistence Library link: https://vertx.io/docs/vertx-pg-client/java/ (accessed: 19.03.2025)

VertX is an event-driven application framework that runs on the Java Virtual Machine (JVM) and was designed for building scalable reactive applications. It is worth to mention that VertX supports multiple programming languages including Java, Kotlin, JavaScript, Ruby, Groovy, Ceylon, etc. This feature allows developers to write different components in different programming languages and have them interoperate within the same application. The code written in boundaries of this thesis is written only in Java. The common use cases for VertX are real-time web applications, microservices architectures, Internet of Things (IoT) applications, etc. [66] [57].
VertX PostgreSQL Client is reactive, non-blocking and allows to handle many database connections with a single thread. Listing 5.3 demonstrates an example of VertX PostgreSQL Client usage.
First of all, the developer specifies connection options like host, port, database name, and login credentials to access the database. After this is done, the developer can define the so-called pool options. Pool options help us to manage and optimize the use of database connections. These options usually include settings for the maximum number of connections, idle timeouts, and more. In our case pool options define how many connections can be reused in SQL clients. The number of such connections must be selected carefully. Too few connections can lead to bottlenecks, while too many can overwhelm the database server or consume excessive resources.

```
// Create an Observable
Observable<Row> observable =
    pool.rxGetConnection().flatMapObservable(conn ->
  conn
    .rxBegin()
    .flatMapObservable(transaction ->
      conn.rxPrepare("SELECT * FROM Message;")
      .flatMapObservable(preparedQuery -> {
        RowStream<Row> stream = preparedQuery.createStream(50);
        return stream.toObservable();
      })
    .doAfterTerminate(transaction:commit)));

// Subscribe
observable
  .subscribeOn(Schedulers.io())
  .subscribe(row -> {
    // do something
  });
```

Listing 5.4: VertX PostgreSQL Client Streaming

After pool options are set, VertX allows to create an SQL client and to use it in order to insert, modify, or delete data in the database. In an example in Listing 5.3 the select query for person with id = 123 is shown. This query is used in *ChatApp* during the

login/registration procedure in order to check if a person is present in the database. If the person is present, the procedure is completed, otherwise, the registration form will be proposed.
If we want an Observable-like behaviour, where we can emit values from the database as we can do from the data stream, we can use Obeservables or Flowables. Listing 5.4 shows an exmaple constructed with use of Observables [60].
Listing 5.4 begins with the creation of an observable. The observable will emit all the messages users sent throughout the chat history which are stored in the database. For this reason, results of an SQL query execution are first transformed to a stream (using the createStream() function, which uses a limit of rows to be retrieved from the database as input) and afterwards, mapped to the Observable. The generic type used in Observable is a row, so Observable emits database rows. In order to subscribe to the observable we used an I/O Scheduler which orchestrates emission receiving on multiple available threads.
Using the VertX PostgreSQL Client, all critical database operations in *ChatApp*—such as login, logout, saving messages, retrieving messages, and updating personal data—can be efficiently implemented. The next step is to organize the duplex communication between clients and server in a reactive way. The client must be able to send messages to the server, and to receive messages from the server. The server in its turn must also be able to send and to receive messages from clients. So, both clients and the server must be able to observe messages in order to receive them and must be observed in order to be able to send them. In the next subsection we will depict the solution in more details.

Reactive WebSocket Communication

As stated earlier in this section WebSockets can be enhanced by integrating it into reactive programming. Having this in mind we implemented *ReactiveWebSocket*, a class which contains all features both clients and the server need in order to communicate with each other. An abstract *ReactiveWebSocket* class extends rxjava3.subjects.Subject which makes *ReactiveWebSocket* a sort of bridge between observable and observer. As stated in Chapter 4 of this thesis such classes are able to subscribe to one or more observables and pass through the items by reemitting them. Both *ReactiveClient* and *ReactiveServer* classes then extend the *ReactiveWebSocket* class.
The communication between different parties involves the exchange of messages, meaning that each party needs to listen to incoming messages and to send own messages to other communication participants. If we think about this in a reactive way, each process which communicates with any other process should be able to emit one or many messages (have the so-called observable ability), and also to subscribe (to observe) all incoming messages (we will call it observer ability). To implement the observable feature we need to extend the observable class and implement the abstract method *subscribeActual(Observer observer)*, which in its turn creates a new *WebSocketDisposable* and stores the observer into the list where all the observers are stored [79] [51]. Listing 5.5 demonstrates the idea in more details.

```
...
private Collection<WebSocketDisposable<T>> observers = new
    HashSet<>();
...
@Override
protected void subscribeActual(final Observer<? super T> observer)
    {
final WebSocketDisposable<T> disposable = new
    WebSocketDisposable<>(observer);
  observers.onSubscribe(disposable);
}
...
private final class WebSocketDisposable implements Disposable {
  private final Observer observer;

  private WebSocketDisposable(final Observer observer){
    this.observer = observer;
  }

  @Override
  public void dispose(){
    observers.remove(observer);
  }

  @Override
  public void isDisposed(){
    return !observers.contains(observer);
  }
}
}
```

Listing 5.5: Adding Observable Ability To Web Sockets

The collection of WebSocketDisposables with the name *observers* from Listing 5.5 can then be called in both clients and servers in *onMessage()*. Listing 5.6 shows such calling for broadcast, for example. If the message arrives, it will automatically be sent to all observers.

```
...
@OnMessage
public void onMessage(final String message){
  observers.forEach(observer -> observer.onNext(message));
}
...
```

Listing 5.6: Handling Incoming Messages With Observables

To implement an Observer feature we only need to implement the standard interface

Observer<T> with common methods *onNext(), onSubscribe(), onError(), onComplete()*, where *onSubscribe()* will store the disposable to the *WebSocketDisposable* collection [51]. Listing 5.7 shows an example of a client side implementation. In the implementation it is visible that *onSubscribe()* just adds the disposable to the list *observers*. The method *onNext()* sends the message to the server via a previously opened session.

```
...
@Override
public void onSubscribe(final Disposable disposable){
  observers.add(disposable)
}

public void onNext(final String message){
  try {
    if(session.get() != null) {
      session.get().getBasicRemote().sendText(message);
    }
  } catch (final IOException e) {
    //ignore
  }
}
...
```

Listing 5.7: Adding Observer Ability To Web Sockets

On the server side we need to distinguish what type of sending (broadcast, blockcast, or unicast) has to be applied to a message distribution. The logic for this is then implemented in *onNext()*. In the case of broadcast, the message will be sent to all open sessions; in the case of blockcast we will filter only sessions we desire before sending the message; in the case of unicast the message will be sent to the only session specified in the message prefix.
In the next section we will define important software product quality attributes, create various scenarios and metrics, and conduct the comparison and evaluation of both software products we described in this section.

CHAPTER 6

Evaluation

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

— Winston Churchill.

In Chapter 3 the concept of threads was introduced. Chapter 4 presented the notion of the reactive programming with help of RxJava and provided information about how RxJava can be utilized for solutions of concurrent problems. In Chapter 5 we created and discussed the concurrent system scenario and its implementation in Java using both threads and reactive programming, with an overall goal to compare both implementations and answer the research questions defined in Chapter 1. In this chapter we compare and evaluate the comparison results of both implementations relative to each other.
As we have demonstrated earlier, both concepts threads and reactive programming qualify for expressing concurrent computations. Moreover, their mechanisms and abstractions also support parallel multicore and distributed computations. The scope of this thesis is bounded to concurrent computations with no multicore and distribution features involved.
In Chapter 1 we defined the research questions, where we postulated the objectives based on which the comparison can be done. Namely, in scope of our interest we announced: *developing effort*, *performance*, and *memory consumption* comparisons. These objectives were deduced as being important based on following resources: [53], [78], [59], [44], [67].
In addition, the ISO/IEC 25010 norm (a revised version of the former ISO/IEC 9126) elects eight major attributes to evaluate the quality of a software system. They can be found alongside their respective sub-metrics in Figure 6.1. In a research about the importance of software quality attributes, the authors conducted several conversation with diverse stakeholders and were able to assess that *Modifiability, Performance,* and *Usability* are the top three attributes concerning business sustainability.
In this thesis the goal is not only to provide the evaluation of metrics important for the final business users (like *performance* or *usability* of the final product), but also to

evaluate the convenience of both strategies for developers and testers. The *maintainability* of a software product in our understanding includes not only metrics given in the sub-tree in Figure 6.1, but also, for example, the time a developer or tester must spend to master the technology in order to be able to extend or change the current application; or also the simplicity of code for those who will read it with an intent to add, modify, or delete some features; etc.

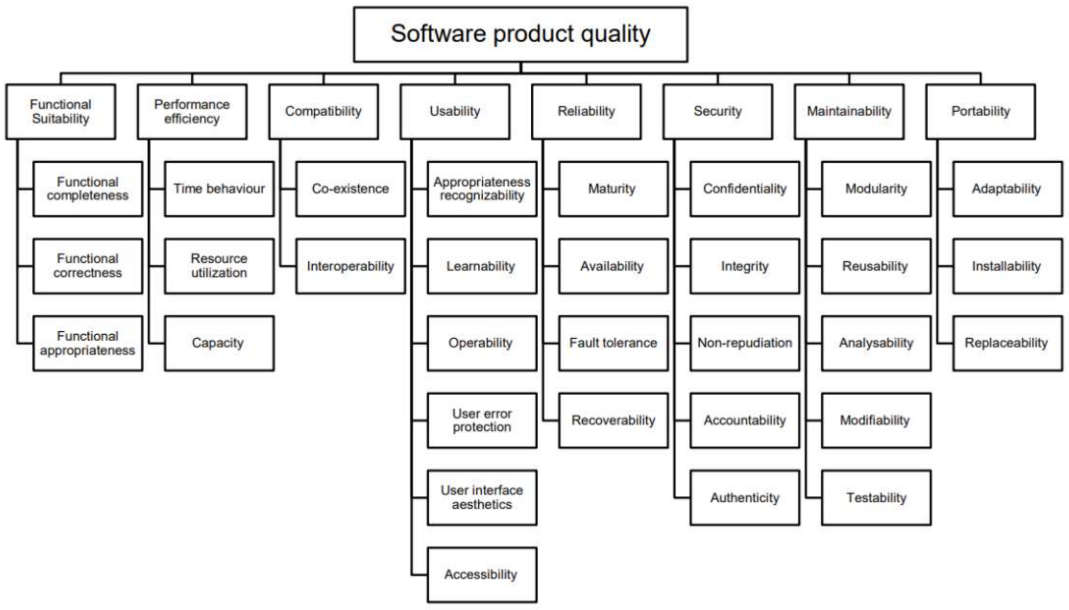

Figure 6.1: ISO/IEC 25010 Attributes for Software Quality Evaluation

In the next sections of this chapter we will conduct the comparisons for the important objectives, discuss the results and conclude each section with a summary.

6.1 Functional Sustainability

RxJava and threads offer different approaches to handle concurrent operations in Java. In this section we discuss the functional sustainability of both concepts for solving concurrent problems and will figure out what benefits and drawbacks are there for usage of both technologies. Here we do not aim to provide formal proofs of behavioural or functional equivalence. Instead we revert to informal discussions about the observations we made during the development phase. Ultimately, we are interested to understand if RxJava has the equivalent expressive power to convey the same ideas as can be implemented with threads and if there are some benefits of RxJava usage.

6.1.1 Programming Paradigms

Under *Programming Paradigm* we understand a style of building the structure and elements of computer programs.
A computer program written with use of threads has to include all the code for managing threads creation, execution, synchronization, and lifecycle. All the operations must be implemented manually which leads to a significant amount of boilerplate code. We define the concurrent computer program written with threads as a program written in an *imperative* programming paradigm. Code written in this style can become complex and difficult to manage especially as the project grows. The need to explicitly define and develop each step can lead to convoluted code, which becomes harder to read and to understand. The code itself can contain more hidden bugs (just because the number of code lines is large), but it is also more prone to bugs. Changes once made in one code part can affect other parts leading to unpredictable behaviour and difficult-to-track errors. On the other hand, the imperative style can provide developers with all the needed mechanisms in order to fully control and understand the code.
On the contrary, reactive programming (including programming with RxJava) encourages a *declarative* programming paradigm where a developer defines and implements how data flow in the system, and the framework is able to handle execution details. The developer writing code with RxJava describes what should happen when some event occurs (i.e., data emission, error, completion, etc.) allowing him or her to focus on what the system should do, rather then how to do it. As consequence the amount of code the developer has to write can be smaller compared to imperative programming paradigm. This can decrease the likelihood of bugs to occur. The program becomes more concise and more goal-oriented (each line of code leads to some functional goal, either message validation, or sending). On the other hand, the code becomes less intuitive and many functions happen rather as implemented in some "black box". The declarative programming style can be rather hard to understand and to manage for developers with relatively low level of experience.
The implementation of *ChatApp* using threads comprises a total of 1,291 lines of code.[1] Specifically, the server implementation accounts for 698 lines, the client implementation consists of 179 lines, while the remaining lines are allocated to database operations and entity mappings.
The RxJava-based implementation of *ChatApp* comprises a total of 1005 lines of code. This includes the ReactiveWebSocket class with 190 lines, the Server class with 184 lines, the Client class with 154 lines, and various auxiliary classes responsible for database operations and entity representations, collectively accounting for 477 lines.
Consequently, it is evident that the RxJava-based implementation reduces the codebase by nearly 300 lines (precisely, 286 lines) compared to the thread-based implementation. As a rather involved example for the benefit of RxJava with a declarative style over the threads with an imperative style we can use backpressure handling. RxJava provides

[1]Here and in the RxJava-based implementation the total amount of lines includes only the logic needed to implement features from Chapter 5. Total amount of lines does not include code used for the benchmark and unit tests.

Flowables and diverse strategies how to handle backpressure. Using threads there is a need to manually handle the situations where data is produced faster then it can be consumed. It really increases complexity and sometimes requires additional technologies like queues etc.

6.1.2 Concurrency Handling

In the literature ([12], [35], [84], [19]), researchers identify and classify various concurrency models. A concurrency model defines the mechanisms by which threads in a concurrent system interact and collaborate to accomplish assigned tasks. Java threads and RxJava are founded on distinct concurrency models, which fundamentally shape their approaches to handling concurrent computations.
Threads are based on the so-called *parallel workers* concurrency model. The name of the model is derived from organizational agencies, where each new task is delegated to a distinct worker. In Java it is a traditional concurrency model in which each request is processed on a separate thread. It is used in a wide variety of servers including Tomcat, Apache, etc. This model is straightforward and easy to implement; however, it has limitations when managing a large volume of simultaneous requests. Specifically, it can be resource-intensive, as it requires the creation of a new thread for each incoming request. Furthermore, this model may exhibit reduced performance efficiency due to the overhead associated with thread start-up and termination, as well as the potential for blocking and bottlenecks. When a high volume of simultaneous requests is received, the system may exhaust its available thread resources, rendering it incapable of processing additional requests.
RxJava concurrency is based on the *event driven* concurrency model (sometimes also called *assembly line* or *reactive* concurrency model). It is used in a number of event driven platforms and servers like: Netty, Akka, Node.JS, etc. In this model, a pool of threads collaborates to handle connections and requests in a non-blocking manner. Each thread manages a finite number of connections and utilizes an event loop to efficiently process I/O operations and network events. When an event occurs, the thread registers it in its event loop and processes it in sequence when it reaches its turn. This approach enables the system to manage a high volume of connections efficiently without degrading performance. In terms of previous explanation for parallel worker concurrency model: not all the work in event driven concurrency model is completed by a single worker, rather the work is divided into small parts and is executed by each and every worker in the queue line by line in a single direction.
A key advantage of the event driven concurrency model is its ability to handle a large number of connections using only a small number of threads, thereby optimizing resource utilization. Furthermore, the event driven model demonstrates greater resilience to bottlenecks and blocking, as threads avoid start-up and shut-down overhead and are limited to managing a predefined number of connections. The number of threads utilized in this model is typically determined based on the processor's core count and overall computational capacity, ensuring efficient utilization of system resources.
The primary disadvantage of the event driven concurrency model lies in the distribution

of task execution across multiple threads and locations in code, often spanning several classes within the project. This fragmentation can make it more challenging to trace and understand the specific code being executed for a given task.
In addition the code written in the event driven concurrency model can have many nested callback handlers and it can get complicated to track what the code is really doing.
In the Section 6.2 we will investigate performance and memory consumption differences for both implementations of *ChatApp*.

6.1.3 Error Handling

The execution of each program can be faulty and lead to an error. Some errors can occur due to bugs in source code, other, can result from unexpected events like big latency in some communication lane, or sudden shut-down of an important service. For example, in *ChatApp* if a server stops providing of service for the clients, all the clients must also eventually stop with an error, because the communication of users without active server is impossible.
Error handling in Java threads has to be done using traditional try-catch blocks inside the thread execution. The definite behaviour in case of an error can be implemented by the developer manually. There are possibilities to interrupt or to terminate the thread. In thread's context an interrupt is an indication to a thread that it should stop what it is doing and do something else. The most common developer's decision is to terminate the thread in case of an interruption. Another decision might be the implementation of interruption policies. The interruption policy determines how a thread interprets an interruption request – what it is supposed to do if the interruption occurs, what units of work are considered to be atomic with respects to interruption, etc. The interruption itself is a risky and erroneous option, because if each thread has its own interruption policy, the interruption should be conducted only if the developer knows what it means for a current thread [24].
Errors in threads can also go unnoticed for developers. For example, if there is no proper error handling and if there are hundreds of threads running and one particular thread has an error, in this case the stack trace may be printed somewhere on the console, but no one may be watching the console. When one thread of hundreds fails, the application may appear to continue to work, because other threads are running (and maybe also producing some log outputs). The consequence of unnoticed thread termination can depend on the thread's role in the application. If some worker thread terminates and the application was running well with 100 threads, it may also run well with 99-threads. On the other hand, if several threads will fail, there might be performance problems visible for the end customers of the software product [24].
RxJava offers a built-in mechanism for graceful error handling. Errors can be caught and handled at any point in the data stream using operators like *onErrorResumeNext()*, *onErrorReturn()*, *doOnError*, etc. The developer is encouraged to implement error handling in each observer using *onError()* methods regardless of concurrent execution or not. In this way RxJava provides clear separation between normal and exceptional flows [49].

Disadvantages of the RxJava error handling strategy include following considerations ([15], [34] [49]):

- **Lack of centralized error management.** Unlike traditional exception handling mechanisms in Java (like try-catch), error handling in RxJava is distributed across streams. This can lead to inconsistent error handling strategies across different parts of the application, making it harder to track and manage errors effectively.
- **"Silent" errors.** Certain error handling strategies (i.e. *onErrorReturn*) depending on their implementation can lead to unnoticed errors where errors are swallowed and not logged or communicated properly. This might result in hard-to-diagnose bugs, especially when the error is critical but not properly handled.
- **Complexity of error handling logic.** RxJava provides a variety of operators to handle errors, but choosing the right one for a particular use case can be complex. There may be cases where multiple levels of error handling are required, leading to messy and hard-to-maintain code.
- **Potential for memory leaks.** Improper error handling can result in subscriptions that are not properly disposed of, leading to memory leaks. If an error occurs and a subscriber is not disposed of correctly, it can hold references to resources that would otherwise be garbage collected, which is especially common in complex asynchronous streams.

In the *ChatApp*, the thread-based implementation employs *try-catch* blocks to verify the availability of the database connection and the partnering client during unicast message transmission. In the RxJava-based implementation, error handling was implemented in analogous locations, supplemented by the use of the *onError()* pathway for the client. This additional mechanism primarily addresses scenarios where the server unexpectedly terminates its operation.
Additionally, validation blocks with error propagation to the user were implemented to handle incorrect user input, particularly during the login and message composition phases.

6.1.4 Composability

We denote components of software *composable* if components can be selected and assembled in various combinations in order to satisfy functional and non-functional requirement of users in reasonable short time with use of minimal personal and computational resources. When comparing RxJava and threads in terms of composability, the key differences lie in the way of handling the creation, combination, and coordination of concurrent tasks.
As in all other comparisons above, with threads, composing multiple asynchronous tasks requires manual management of their execution and coordination. Combining results from multiple threads usually involves shared resources like shared variables, locks, etc.

For example, if two threads run concurrently and after they finished the results should be merged together, the *join()* method can be used. In this case, thread one will wait till thread two completes, and vice versa, thread two will wait till thread one terminates. The results can be then combined manually. This approach can quickly become complex and error-prone when scaling up to many tasks or threads.
RxJava excels at composing asynchronous tasks through its functional reactive programming model. It provides a rich set of operators (e.g., *zip()*, *merge()*, *concat()*, *flatMap()*, etc.) that make it easy to combine, transform, and synchronize the execution of multiple asynchronous streams. Multiple observables can be also combined into one, the execution order in this case can be controlled via special dedicated methods.
RxJava provides built-in support for reactive streams, making it highly composable for scenarios that involve continuous data flows. Multiple streams of data can be easily merged, filtered, combined, and transformed. The reactive operators (i.e., *merge()*, *combineLatest()*, *flatMap()*, etc.) allow developers to create complex data pipelines by combining streams in a declarative and efficient manner.
In the case of using threads, handling and combining continuous streams of data can be implemented manually, but it needs high skills of developers, because unexpected hazards can occur and the probability of occurrence is very high.

6.1.5 Summary of Functional Sustainability Section

In previous subsections we discussed and compared Java threads and RxJava in terms of functional sustainability for handling concurrent tasks. The differences between both technologies are not only in program syntax and usage of distinct functions, but are much more fundamental. This summary provides an answer for the research question RQ1.

RQ1. How and why does reactive programming qualify itself for treating concurrent problems in programming? What are major differences to traditional threads?

RxJava qualify itself for treating concurrent problems. In the following the major differences to threads will be discussed. Further the suggestions when to use threads and when RxJava will be given.
Java threads are rooted in the imperative programming paradigm, where all operations must be explicitly defined and implemented. While this approach may lead to an increase in code volume, which in turn can raise the likelihood of bugs, it offers developers simplicity and convenience by providing full control and a clear understanding of the program's flow.
In contrast, RxJava adopts the declarative programming paradigm, which emphasizes describing what the program should accomplish rather than how to accomplish it. This paradigm often results in more concise code, reducing boilerplate code, but it may also make the code less intuitive and more challenging for developers to comprehend and

maintain.
Threads operate on the parallel workers concurrency model, where each new request is handled by a dedicated thread. Despite its simplicity, this approach can introduce performance bottlenecks due to the overhead associated with thread creation, termination, and their inherently blocking nature.
RxJava, on the other hand, is built on the event loop concurrency model. In this model, requests are broken into smaller tasks that are distributed across different threads for execution. This design minimizes thread management overhead and avoids blocking, improving scalability. However, it comes with its own challenges, such as the fragmentation of task execution across multiple classes or components, which can complicate tracing and debugging.
RxJava offers superior composability of software components due to its extensive library of built-in reactive operators, which simplify the combination and transformation of data streams. Additionally, RxJava includes built-in mechanisms for error handling, ensuring a clear separation between normal and exceptional flows within the application. In contrast, threads lack native utilities for both composability and structured error handling.
Summing it all up, solving concurrency problems with threads is more effective (compared to RxJava) if:

- the developer needs precise control over thread behaviour, such as setting custom thread priorities, creating daemon threads, or managing their lifecycle explicitly. Example: implementing a real-time application where thread priorities directly impact performance;
- tasks are independent and do not require complex inter-task communication or coordination. The amount of concurrent tasks is relatively small. Example: processing a file in a separate thread or performing a background computation;
- operations persist over long durations, such as server listeners, file watchers, or continuously running services where thread management is explicit and predictable.

Solving concurrency problems with RxJava is more effective (compared to threads) if:

- the application involves processing, transforming, or combining multiple asynchronous streams of data. Example: processing of some live data streams, such as stock prices, sensor data, etc.;
- workflows involve multiple asynchronous operations that need to be chained, combined, or synchronized in a readable and declarative way. Example: managing user interactions and data updates without blocking the main thread;
- handling transient failures in requests with retry and/or fallback strategies is required.

6.2 Performance Evaluation

One of the most crucial software product quality attributes is performance. A software program with acceptable performance is able to increase trust of customers, their desire to buy and continue to use the program, and to increase satisfiability of users. In this thesis we will not go into the discussion of what performance can be acceptable for users, but rather discuss what performance have the software solutions from above under different conditions.
In this thesis we compare two different solutions for the chat application *ChatApp* implemented with the client-server paradigm using Java Threads and RxJava. The technologies used for implementation offer different approaches for chat communication: threads give an ability to coordinate the communication via synchronization of parallel tasks, whereas RxJava enables asynchrony and stream-oriented communication. The overall performance comparison will be based on *load* (number of parallel available online users) and time constraints under which the communication must happen.
This section will be structured as follows: in the next subsection the performance metrics for the comparison of both implementations will be defined; in the following subsection we will define the experimental setup; the final subsection of this section will discuss the results of experiments.

6.2.1 Performance Metrics

This subsection is devoted to the definition of performance metrics. In general, under a performance metric we understand the measurable and comparable success indicator that demonstrates how well the program performs under some strictly defined conditions. In this thesis we will define metrics for measurements of time performance and of memory consumption of the two implementations of *ChatApp*.
There are numerous ways of time performance measurements. Some of them aim to answer the "how fast" question which means, if there is some unit of work, the interest is then to observe how fast this unit can be processed under different conditions. Examples for such measurements are service time, latency, efficiency, etc. Other performance management ways concentrate on answering the question "how much" work can be done during given time under given conditions. For instance, here we can name following measurements: throughput, capacity, etc [4] [24].
Comparing threads and RxJava implementations involves analysing the time and memory performance across several key metrics. First of all, we need to distinguish between different types of tasks the programs can have: I/O-bound and CPU-bound tasks. I/O-bound tasks are operations that spend the majority of their execution time waiting for external resources, such as file systems, databases, or network services, rather than using the CPU extensively. CPU-bound tasks are computations that heavily utilize the CPU and spend most of their execution time performing calculations or processing data. We will additionally perform measurements for mixed workloads, incorporating varying proportions of I/O-bound and CPU-bound tasks.
We are also going to compare time and memory performance of implementations using

threads and RxJava in scenarios where threads maintain their states and where threads are stateless. Stateless threads can provide several benefits in terms of time performance. These benefits arise from the fact that stateless threads do not need to retain or access past state, which simplifies their execution.
J. Pong et. al. [59] developed a benchmark using JMH (a subproject of OpenJDK) used for comparison of performance metrics of different reactive programming libraries (inclusive RxJava). In their article they compare the performance for individual CPU-bound operations, multiple-operator pipelines, and I/O-bound pipelines. According to the authors, there is neither an established standardized benchmark for reactive programming libraries in Java, nor a benchmark for reactive streams. The benchmark authors provided in the article can be adopted for performance measurements for this thesis.
For performance evaluation of both implementations of *ChatApp* a custom benchmark is implemented.
The next subsection will disclose what experiments will be conducted in order to measure performance.

6.2.2 Experiment Setup

In this section we will discuss what experiments were conducted in order to measure time and memory performance for both implementations of the concurrent scenario. We gather the experiments under 3 groups: I/O bound tasks, CPU-bound tasks and mixed workload (under which we understand the normal operation of the software, where both I/O bound and CPU-bound tasks are mixed).
In all experiments described below, there is a need of a user to communicate with the system by means of providing login data, composing, sending, editing messages, etc. For the purposes of simplicity and equality of test conduction we abstract this communication off by providing predefined immediate user responses as soon as the server requires it. This statement is valid for both scenarios and all experiments described below.
All the experiments will be executed on the following environment:

- Operating System: Windows 10 Home (Version: 22H2)
- System type: 64-bit operating system, x64-based processor
- Device: Dell Inspiron 16 7610
- Processor: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
- Installed RAM[2]: 16,0 GB

In all experiments we conducted time measurements with use of standard Java *System.currentTimeMillis()*. We made a measurement before and after the experiment execution. The total time needed for the experiment was obtained as the difference

[2]Random Access Memory (RAM) is a type of electronic computer memory. It allows to read and write data items in any order in almost the same amount of time irrespective of physical location of data inside the memory.

of measured times.
In all experiments we used the difference between free available memory before and after the execution of the experiment (in Java: *Runtime.getRuntime().freeMemory()*) as total memory consumption value.
If additional measurements were incorporated into the experiment, they are detailed in the corresponding discussion of the experimental results.

Experiment A – I/O Bound Tasks

In this experiment our intent is to show time and memory performance for the tasks where I/O operations are central. As stated above, we want to distinguish between two variants of threads in scenario implemented with threads: stateless and stateful threads. Both experiments from below will be conducted.

Experiment A.1 – I/O Bound Tasks – Stateless Threads The login operation of the user includes the user's input of credentials, a lookup (read operation) in the database, comparison of received data, and an update of a thread-local login status. In the scenario implemented with threads, each user is represented with a separate thread and each thread during the login phase accesses the database to read credentials of its certain, distinct user (it is a non-blocking access). In our implementation, in the case of the login operation, no state is shared between threads.

Experiment A.2 – I/O Bound Tasks – Stateful Threads A message editing operation will be employed for this experiment. Each user will try to edit the same message received earlier. Each new edition of the message will be stored in the database (and eventually resend to all other users, making the state of the threads shared).

Experiment B – CPU-bound Tasks

In this experiment our intent is to measure time and memory performance for the tasks, where the operations require some CPU capacity for some calculations. As in Experiment A we will distinguish between stateless and stateful threads implementations.

Experiment B.1 – CPU-bound Tasks – Stateless Threads In this experiment we utilize the feature of message broadcasting. We define one user as sender of a message and other users as receivers. The sender will send a line of 1000 random words of different length (with maximal length of 50 characters), where each word is a random sequence of letters from the English alphabet. Each word in a line is separated by space from other words. The task for all receivers will be to check if at least one of the words sent by sender is a palindrome, and if it is so, calculate the total number of palindromes in a list received from the sender. Receivers will not share any data, the calculation will happen strictly inside each thread.

Experiment B.2 – CPU-bound Tasks – Stateful Threads For this experiment we will utilize unicast and broadcast message sending with message editing features. As in the previous experiment, there will be one sender and the other threads will be receivers. The sender will unicast send for each receiver a new generated line of 1000 random words of different length (with maximal length of 50 characters). After completion it will broadcast an empty line to each receiver. The task of receivers will be to validate the words for the palindrome property and to find all palindromes, afterwards edit the broadcasted message and store all found palindromes from each list sent in the unicast way.

Experiment C – Mixed Workloads (Normal Operation)

Here we do not distinguish between stateless and stateful threads, but just simulate the normal operation of *ChatApp*. The workload can be mixed in different ways: having more I/O bound tasks, or having more CPU-bound tasks, or strive to the equality of both groups cardinalities in the mixture. In this experiment we will conduct several tests with different constellations of I/O bound and CPU-bound tasks and different amounts of users with a goal to understand which implementation gives better results in each case. We will utilize all the features *ChatApp* provided for user management and for message sending and editing.
The experiment will be conducted two times, for 100 users and for 1000 users, where the proportion of I/O bound and CPU-bound will be variable. Our goal will be to measure a throughput for both implementations of the concurrent scenario. Some users will be busy with pure I/O operations, other ones with CPU-bound operations.
In all runs of the experiments, users will have the same amount of messages to send or the same amount of computational tasks to compute (both will be equal to 200 tasks per user). Details like the art of message sending (unicast, blockcast, broadcast), the next message to be edited, etc. will be decided randomly. Just as in previous scenarios, we will reuse a polynomial finding algorithm for CPU-bound tasks with the same conditions as in the experiment B.1. The messages to be analysed will be predefined and stored in the database.
The workload mix will be made to 70/30 percent portions in cases where either I/O bound or CPU-bound tasks should prevail, or we will strive to 50/50 percent portions, where I/O bound and CPU-bound tasks amounts should be equal.

6.2.3 Experiment Results: Platform Threads & RxJava

Each experiment was conducted in a static system configuration to reduce the effects of structural scalability like mobility or elasticity. Each experiment was conducted 3 times in order to reduce the effect of outliers. The data points in the graphs represent the mean of three measured values. Each experiment was started with the so-called "cold start". We restart all involved services, clear the whole database content, generate each message or person new before each experiment. In the following subsections the results of experiments will be presented and analysed.

Experiment A.1

Figure 6.2 shows the plot for the experiment A.1. The plot indicates that when threads do not share state, the application developed with threads demonstrates significantly better time performance compared to the application where RxJava is applied.

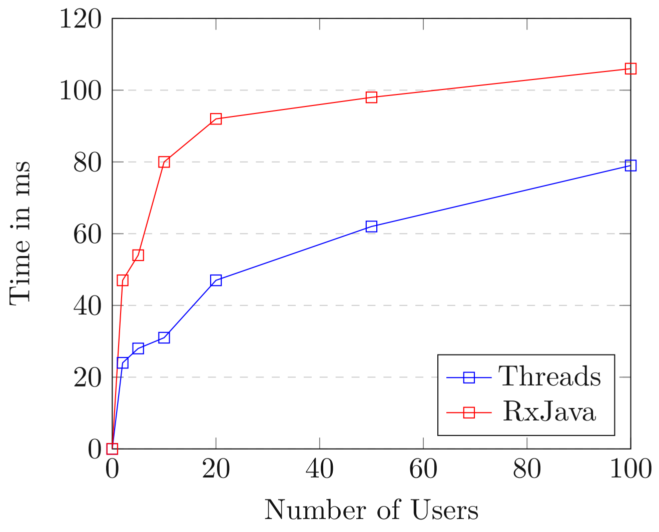

Figure 6.2: Experiment A.1 – I/O Bound Tasks – Stateless Threads. Time Performance Comparison

Memory consumption for both implementations is nearly identical if the number of tasks is small. For example, RxJava implementation uses 15 MB (for 100 users), and in threads case it is 16 MB for the same number of users. But as soon as the number of tasks increases, RxJava shows better memory performance. For 1000 concurrent tasks RxJava memory consumption is 16 MB, and threads have 22 MB.

Experiment A.2

Figure 6.3 shows the plot for the experiment A.2. In this case we can observe the following:

- Java threads have better performance compared to RxJava when the number of threads is relatively small;
- RxJava outperforms Java threads when the number of concurrent tasks increases.

Reactive systems are optimized for I/O-bound tasks. Since I/O operations do not block threads, reactive frameworks can handle hundreds of thousands of concurrent I/O-bound tasks without the need for a proportional increase in threads. Threads in its turn, can become inefficient due to the blocking nature of I/O operations. They can spend a significant amount of time waiting for input in order to complete the task. When the number

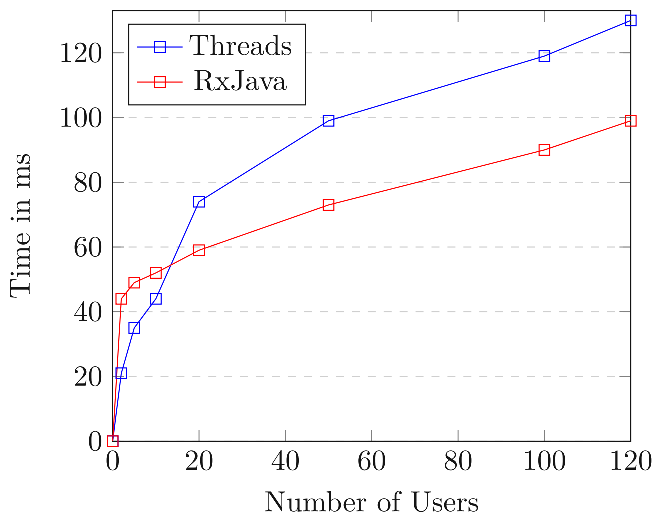

Figure 6.3: Experiment A.2 – I/O Bound Tasks – Stateful Threads. Time Performance Comparison

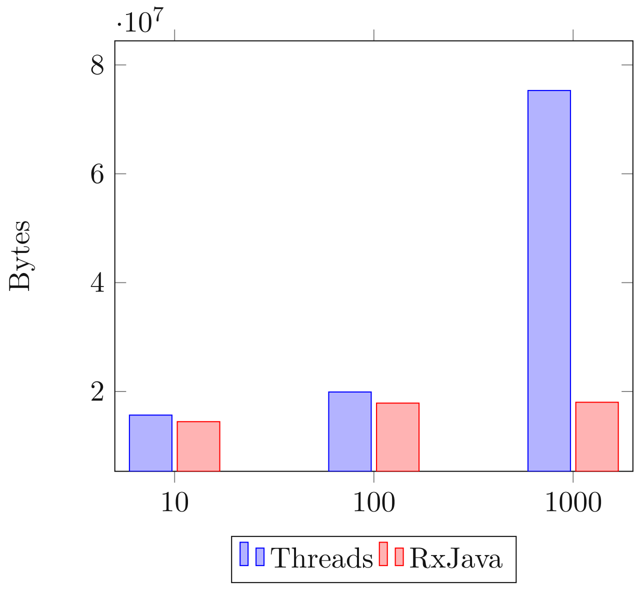

Figure 6.4: Experiment A.2 – I/O Bound Tasks – Stateful Threads. Memory Performance Comparison

of concurrent users is small, the overhead of RxJava's dynamic thread management and abstraction can outweigh its benefits, making a simple thread-based implementation more efficient.
Figure 6.4 depicts the memory consumption comparison for Threads and RxJava implementations. On the X-axis the number of users is shown, and on the Y-axis the amount of bytes each program has consumed during the execution. It is visible from the bar plot,

that RxJava memory consumption is comparable to threads if the number of threads is not very big. But as soon as the number of threads increases, it is evident, that RxJava consumes far less memory for the program execution. Such behaviour of RxJava is due to the reuse of threads defined via Schedulers. If a thread is no longer used, in RxJava it will be reused for further tasks.

Experiment B.1

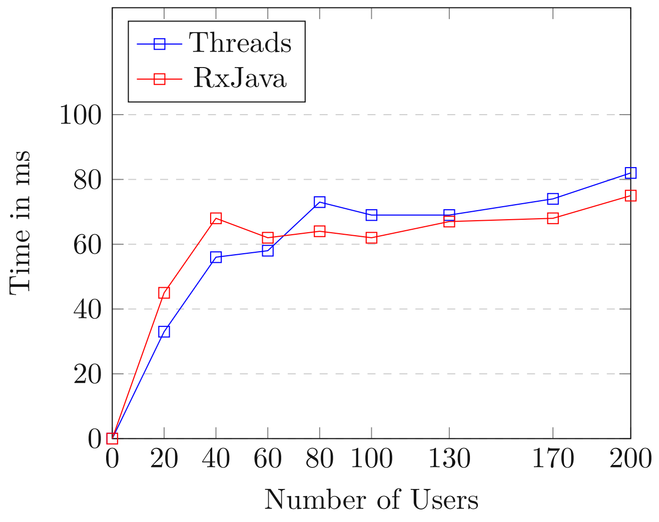

Figure 6.5: Experiment B.1 – CPU-bound Tasks – Stateless Threads. Time Performance Comparison

Figure 6.5 depicts the time performance for a stateless threads implementation. From the figure it is visible that time performance between threads and RxJava implementations is nearly equal. For some cases there are outliers, but in general both implementations have nearly equal efficiency.
The results can be explained as follows. For stateless threads the time performance is bound only to the number of CPU-cores and the computational time needed for providing the solution to the algorithmic task. Threads allow parallel processing by fully utilizing multiple cores of the machine, where the experiment was conducted. There was no context switching, no shared resources, so the performance was relatively stable and bound only to the number of threads. Some outliers (as in case the of 80 threads) are there only because of the "cold start" of each test defined previously. It means that for each new experiment, new randomized messages were generated. In case of RxJava, the performance was dependent to the organization of concurrency. Effective parallelization with an appropriate scheduler (in the case of an algorithmic CPU-bound task, the best choice is *compuation()* a scheduler) can lead to better performance. If the number of users is relatively small, RxJava can get worse time performance compared to threads implementation due to the overhead of RxJava's internal mechanisms like memory allocation, introduction of schedulers, etc.

Figure 6.6 depicts the memory consumption performance for both implementations where threads were stateless. Threads require memory for each new thread, whereas RxJava Schedulers are optimized for threads reuse. For this reason, memory consumption of RxJava even with many concurrent tasks is better compared to threads.

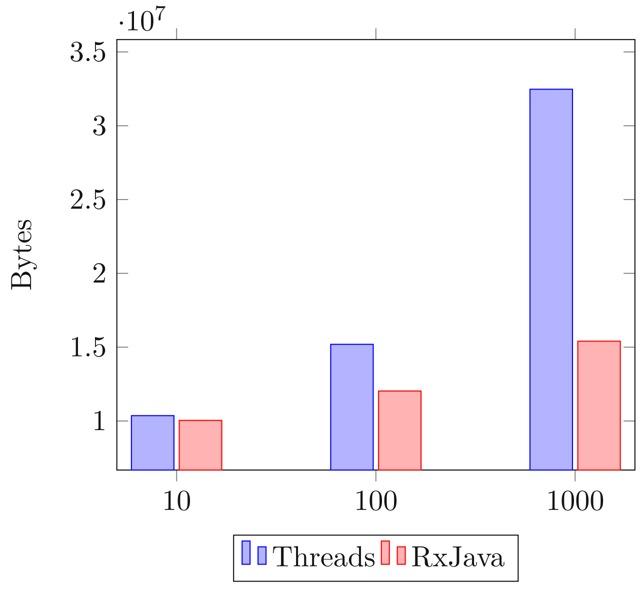

Figure 6.6: Experiment B.1 – CPU-bound Tasks – Stateless Threads. Memory Performance Comparison

Experiment B.2

Figure 6.7 depicts the time performance comparison for the implementation where threads maintain their states. It is visible from the plot that threads are the better choice for computational tasks in means of time if the expected number of tasks is rather small. If the number of tasks increases, RxJava has better performance.
For this experiment we used *Schedulers.computation()* because it can better handle concurrent CPU-bound tasks. If other schedulers are implied, the performance can vary and show worse results. Repetitive tests showed that for the RxJava implementation it is crucial to have effectively parallelized tasks with proper schedulers.
Figure 6.8 depicts the memory consumption bar-chart for both strategies. It is clearly visible, that the RxJava implementation requires far less memory then the threads implementation.

Experiment C

Figure 6.9 depicts a time performance comparison for the normal operation of *ChatApp* with 100 users. Here and in next experiments, the results are shown as follows: the first

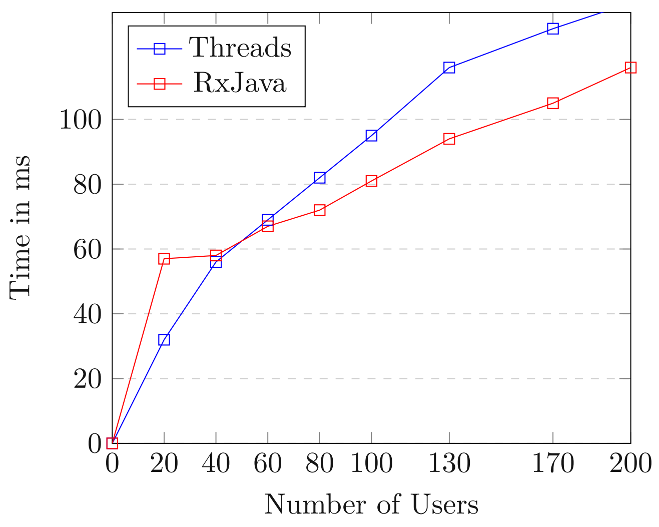

Figure 6.7: Experiment B.2 – CPU-bound Tasks – Stateful Threads. Time Performance Comparison

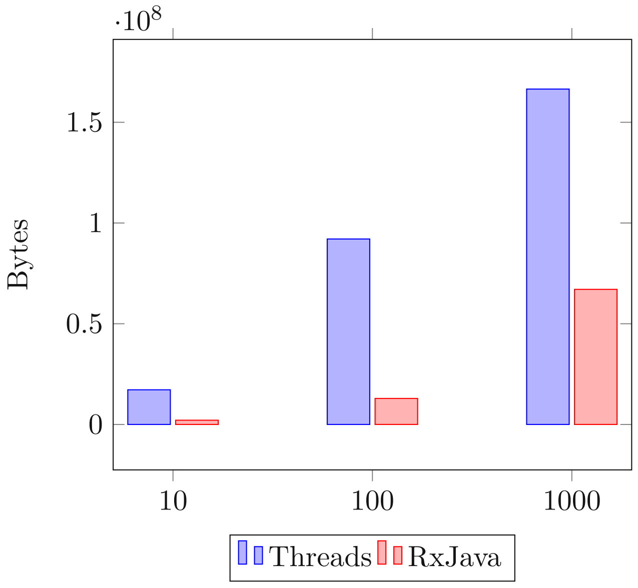

Figure 6.8: Experiment B.2 – CPU-bound Tasks – Stateful Threads. Memory Performance Comparison

two bars denoted as "I/O" in the Figure 6.9 represent the throughput for concurrent users, where more tasks were merely I/O-tasks; next two bars denoted as "CPU" represent the throughput for concurrent users, where more tasks were CPU-bound; and the third two bars represent the throughput for concurrent users, where the number of I/O bound and CPU-bound tasks were equal.

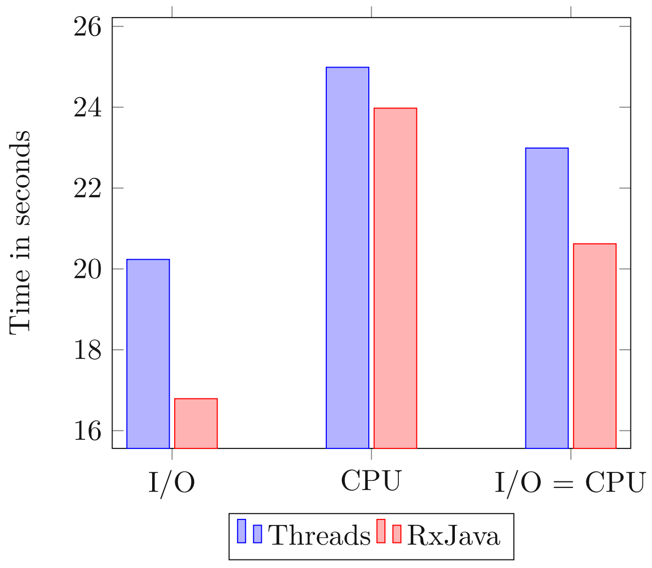

Figure 6.9: Experiment C – Time Performance Comparison for Normal Operation of *ChatApp* for 100 Users.

Figure 6.10 depicts a time performance comparison for normal operation of *ChatApp* with 1000 users.

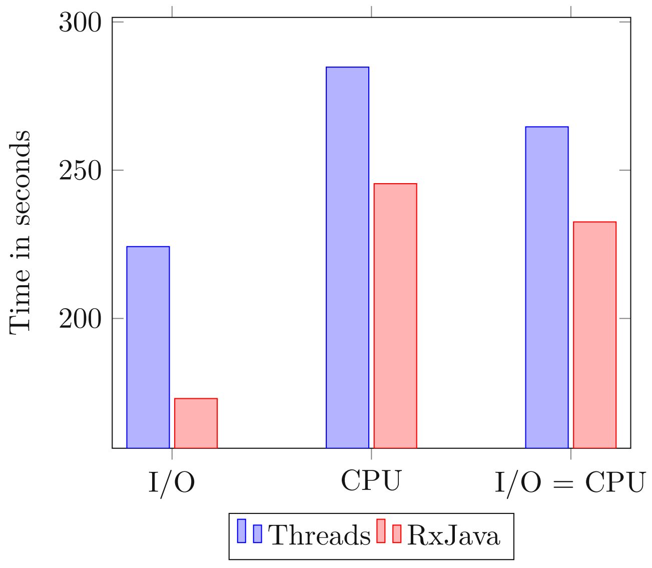

Figure 6.10: Experiment C – Time Performance Comparison for Normal Operation of *ChatApp* for 1000 Users.

From both plots it is visible that the RxJava implementation has better performance compared to threads if there are more I/O bound operations. Performance of both implementations is comparable (nearly equal) if the number of threads is relatively

small and there are more CPU-bound tasks. Increasing the amount of threads makes performance of RxJava better but only if the *Schedulers.computation()* is used for CPU-bound tasks.
In the case of an equally-mixed workload, the time performance of RxJava is slightly better mostly because of non-blocking I/O operations and asynchrony features for messages editing.
As in all other experiments the memory performance of the RxJava implementation is much better compared to the implementation using threads. For 1000 users threads used, nearly 857 MB of memory, while RxJava consumed only 126 MB.
To explain such big difference in memory consumption, we need to know that each Java thread typically consumes a significant amount of memory for its stack space. On the JVM, the default stack size for each thread can vary depending on the platform, but is often in the range of 256 KB to 1 MB per thread.[3] This stack memory is allocated even if the thread is idle [24]. Even more, threads block when waiting for I/O or other synchronization events, leading to inefficient memory use if they remain idle but still occupy system resources.
On the contrary, RxJava uses an event-driven, non-blocking architecture and doesn't create a separate thread for each operation. Instead, it relies on worker threads from a thread pool, reducing the overall memory footprint. Depending on the Scheduler, RxJava reuses threads or threads from a pool, further reducing memory consumption compared to creating new threads.

6.2.4 Additional Considerations on Memory Consumption of Java Threads

In the preceding subsection, we analysed the experimental results, including, among other parameters, the measurements pertaining to memory consumption. Although all the memory consumption measurements presented in the previous section are accurate and valid, an alternative approach exists for estimating the usage of memory.
Each thread in Java has its own memory allocation pattern, divided between the **heap** and the **stack**, which are key components of the Java memory model [40].
The **heap** is a shared memory area used to store objects and class-level variables. All threads within a Java application share the same heap, allowing objects to be accessed across threads. Every object and every data structure used by the thread consume heap memory. The management of the heap's memory allocation is entrusted to the *Garbage Collector* (GC), which autonomously reclaims memory occupied by objects deemed unreachable, thereby enhancing the efficiency of heap utilization [5]. The heap memory size allowed per application can be configured when launching the application with the following parameters:

- **-Xms** – set initial Java heap size

[3]In the subsection 6.2.4 some additional considerations regarding memory consumption of Java threads will be discussed in more detail.

- **-Xmx** – set maximum Java heap size

Too small heap memory can, on the one hand, lead to an *OutOfMemoryError*, and, on the other hand, can force frequent GC runs especially with a large amount of threads, which can lead to throughput and performance degradation. Large heap, in its turn, can lead to underutilization of available resources. Reducing heap size might not significantly lower the actual memory consumption [40] [31].
In contrast, the **stack** is a private memory area for each thread, used to manage method calls, local variables, and references to objects in the heap. The Java Virtual Machine (JVM) includes a configuration parameter, *-Xss<size>*, which determines the stack size allocated for each thread. In 64-bit JVMs, the default stack size is typically set in range up to 1 MB [63] [41], [8]. This does not necessarily imply that a thread actively utilizes the entire memory allocated to its stack. It simply means that the JVM will allocate (or reserve) memory for each thread's stack. With use of *-Xss<size>* parameter the stack size of each thread can be increased or decreased. Too small stack size can cause an *StackOverflowError* [63].
The relationship between the heap and stack is critical for efficient memory management in a multithreaded environment. Objects created inside a method are stored in the heap, while references to these objects are stored in the stack. For example, if a thread creates an instance of a class inside a method, the object resides in the heap, and the reference is stored in the thread's stack [40] [31].
Now we consider the *ChatApp*. Changes to heap size only impact the portion of the memory allocated for objects. Reducing the heap size will not significantly diminish the propensity of threads to create objects; instead, it will introduce additional challenges and may lead to exceptions, as previously discussed. However, recognizing that a significant portion of allocated stack memory often remains unused, we can consider adjusting the stack memory size and attempting to reduce it.
We run Experiment C and set the stack sizes of all threads to 512 KB[4] (*-Xss<512K>*). With this restriction application uses nearly 50 MB for 100 threads and approximately 500 MB for 1000 threads. The experiment runs without any issues. We can conclude that the spare space allocated in the default configuration for each thread was never utilized by any of the threads.
We further reduce the stack size to 256 KB. The experiment terminates successfully when I/O-bound tasks prevail but throws a *StackOverflowError* when CPU-bound tasks dominate. If the number of I/O-bound and CPU-bound tasks is equal, the experiment may either produce the exception or terminate successfully, depending on the specific conditions.
I/O-bound threads spend much of their time waiting for external events. These threads does not execute complex code and do not involve deeply nested method calls. As a result, they require less stack memory and can better cope with stack memory reduction. On the contrary, CPU-bound threads perform intensive computations and actively use the stack to store variables, method calls, and temporary data for computation. CPU-bound

[4]The default value set for all previous experiments was 1 MB.

threads are more prone to hitting the stack size limit because their active stack usage is significantly higher.

6.2.5 Experiment Results: Virtual Threads & RxJava

During the writing stage of this thesis, developers finalized the implementation of a new feature in the Java programming language —- virtual threads. In this subsection, we enhance the implementation of the concurrent scenario by utilizing virtual threads instead of platform threads. Afterwards we repeat all the previously conducted experiments, comparing the implementation using RxJava with the one utilizing Java virtual threads. We conducted all the experiments under the same preconditions defined in Subsection 6.2.3.

Experiment A.1

Figure 6.11 shows the plot for the experiment A.1. From the plot we deduce that virtual threads outperform RxJava significantly when threads are stateless and operations are I/O-bound.
Figure 6.12 depicts memory consumption comparison for both implementations. From the figure we observe nearly equal memory consumption.

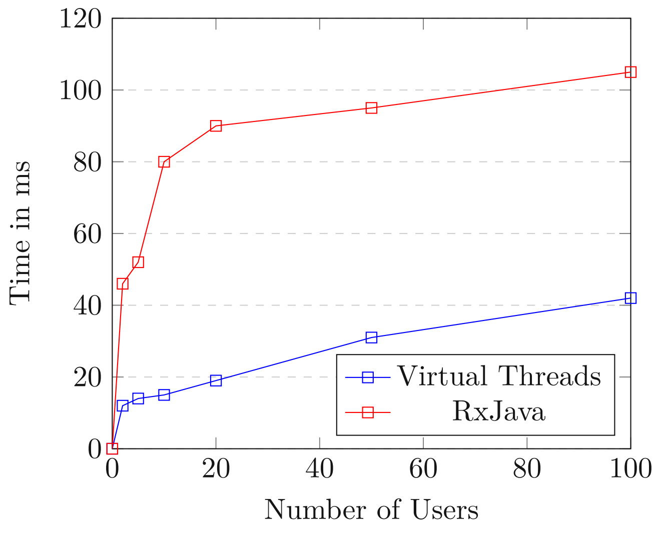

Figure 6.11: Experiment A.1 – I/O Bound Tasks – Stateless Virtual Threads. Time Performance Comparison

Experiment A.2

Figure 6.13 presents the plot for the experiment A.2. From the plot we can observe that virtual threads have better time performance compared to the RxJava implementation when number of concurrent tasks is small. With the increase of concurrent users the time

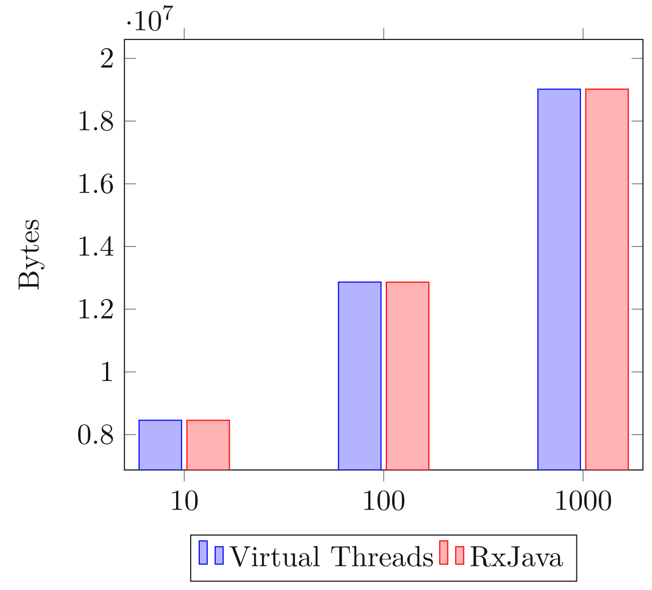

Figure 6.12: Experiment A.1 – I/O Bound Tasks – Stateless Virtual Threads. Memory Performance Comparison

performance of both implementations is comparable and for some runs of the *ChatApp* was nearly equal with very insignificant difference.

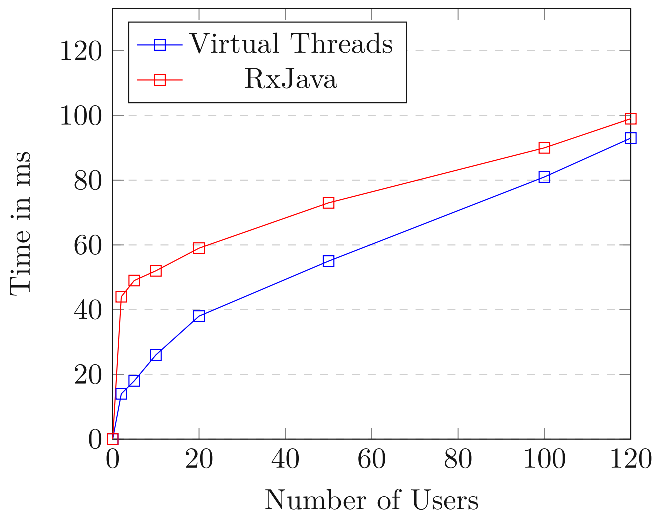

Figure 6.13: Experiment A.2 – I/O Bound Tasks – Stateful Virtual Threads. Time Performance Comparison

Figure 6.14 shows the memory consumption comparison between 2 implementations. We can observe that memory consumption is nearly equal and in case of 10 threads, RxJava implementation consumes even a little bit more memory.

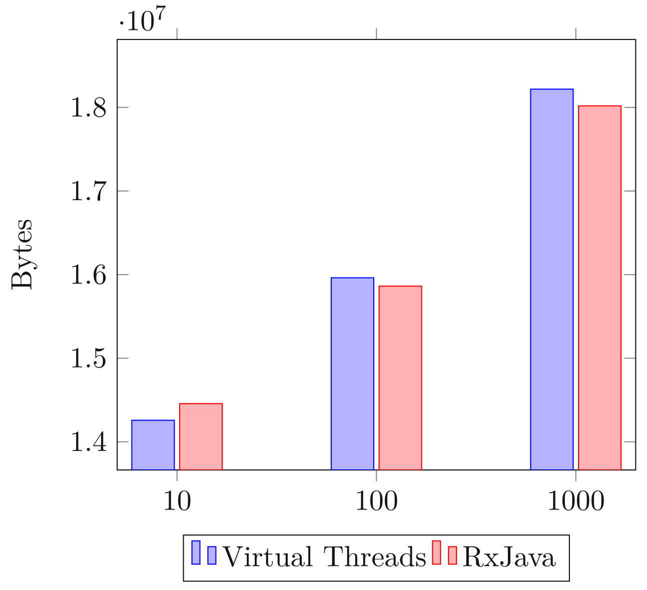

Figure 6.14: Experiment A.2 – I/O Bound Tasks – Stateful Virtual Threads. Memory Performance Comparison

In general, experiments indicate that virtual threads exhibit superior or, in some cases, comparable performance to RxJava in I/O-bound workloads. This can happen due to efficient scheduling, and lower complexity of virtual threads. RxJava's reactive model introduces additional processing costs through, for example, event propagation. This explains the moderate performance especially in cases when the number of concurrent tasks is small.

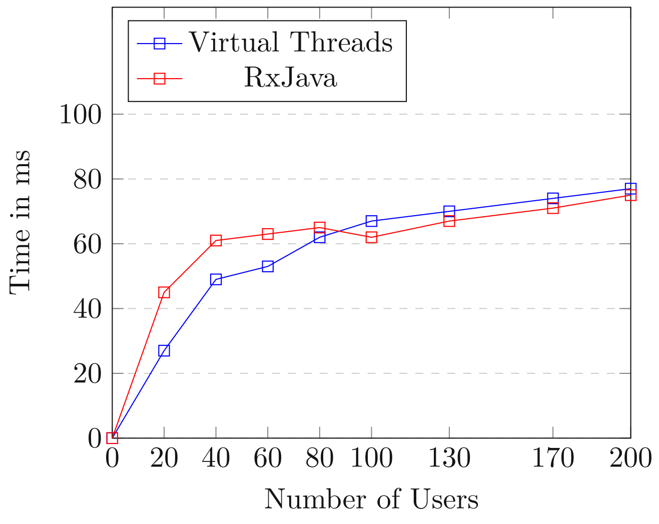

Figure 6.15: Experiment B.1 – CPU-bound Tasks – Stateless Virtual Threads. Time Performance Comparison

Experiment B.1

Figure 6.15 depicts the time performance comparison for a stateless case of virtual threads and RxJava implementations. The results indicate that the execution times are nearly equivalent, with virtual threads demonstrating a slight performance advantage when the number of threads is low.
The results are similar to Figure 6.5 where we compared implementations using platform threads and RxJava. When threads do not share state, both virtual threads and platform threads have minimal synchronization overhead. Virtual threads are lightweight and designed for frequent scheduling, while platform threads are typically heavier but also handle independent tasks efficiently. In this case the scheduling is handled similar for both platform and virtual threads, leading to comparable performance.
Figure 6.15 presents the memory performance comparison for a stateless case of virtual threads and RxJava implementations. Similar to the previous experiments, the results reveal that both implementations exhibit nearly identical memory usage.

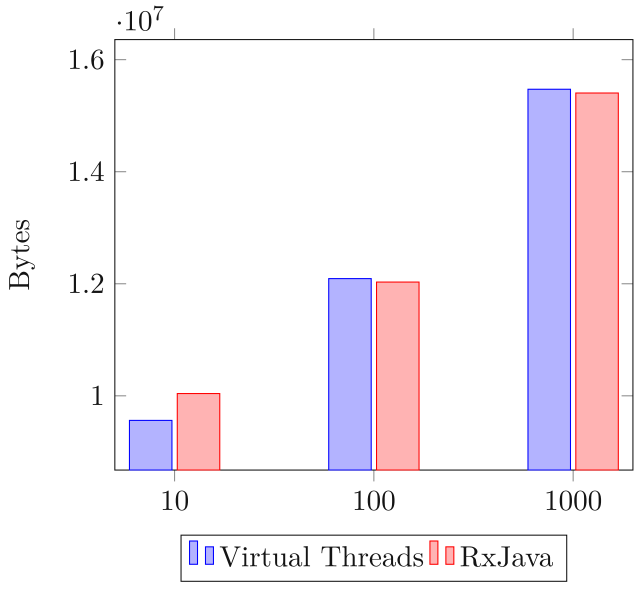

Figure 6.16: Experiment B.1 – CPU-bound Tasks – Stateless Virtual Threads. Memory Performance Comparison

Experiment B.2

Figure 6.17 presents the time performance comparison for the case of stateful vurtual threads and RxJava implementations. The plot indicates that virtual threads demonstrate superior performance when the number of concurrent tasks remains relatively low. However, as the number of threads increases, the RxJava implementation surpasses the virtual threads implementation in performance. Nevertheless, the difference in execution times between the two approaches remains relatively small.

By comparing the time performance of platform threads, virtual threads, and RxJava implementations (Figure 6.7, Figure 6.17), we can conclude that virtual threads do not provide significant performance advantages in the stateful scenario compared to platform threads. However, their performance remains comparable to that of the RxJava implementation.

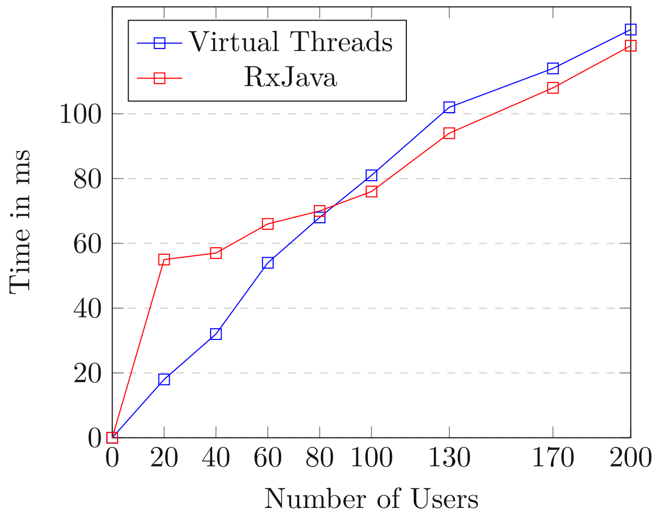

Figure 6.17: Experiment B.2 – CPU-bound Tasks – Stateful Virtual Threads. Time Performance Comparison

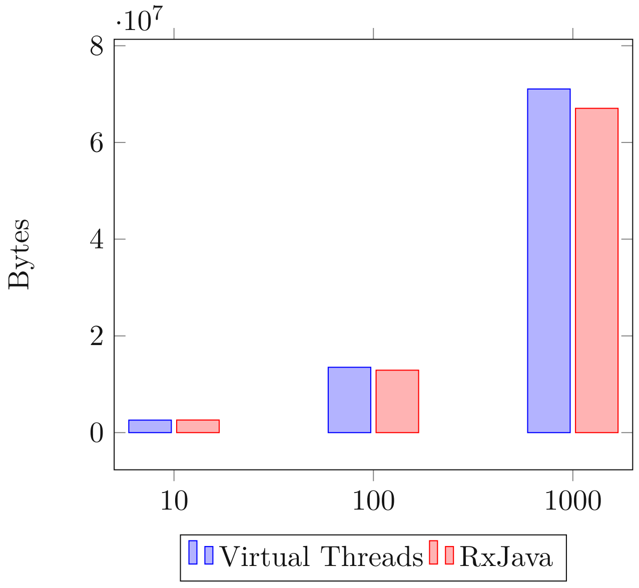

Figure 6.18: Experiment B.2 – CPU-bound Tasks – Stateful Virtual Threads. Memory Performance Comparison

Figure 6.18 depicts the memory consumption comparison for both implementations. It is visible from the figure that RxJava exhibits a little bit more efficient memory consumption; however, the difference is not substantial.

Experiment C

Figure 6.19 depicts a time performance comparison for the normal operation of *ChatApp* with 100 users. Figure 6.20 depicts a time performance comparison for normal operation of *ChatApp* with 1000 users.
From both figures we can obtain the following:

- Virtual threads have significantly better time performance when concurrent tasks are I/O-bound
- RxJava has slightly enhanced performance when concurrent tasks are CPU-bound.
- Both virtual threads and RxJava are viable options for the normal operation of ChatApp, offering comparable performance while outperforming standard platform threads.

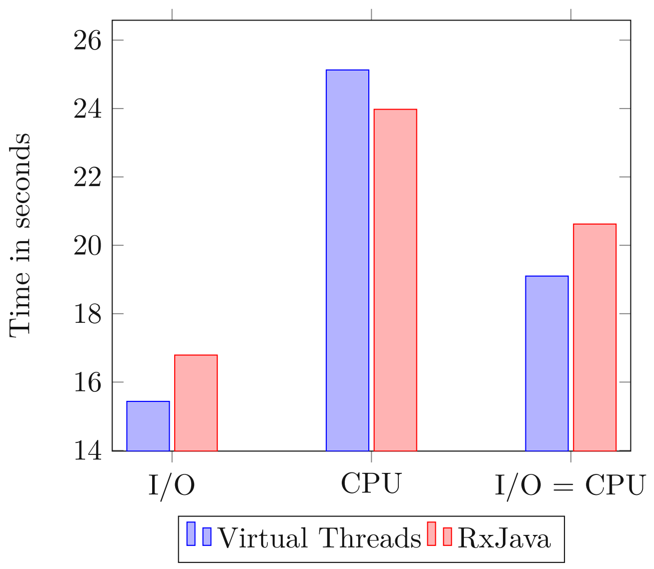

Figure 6.19: Experiment C – Time Performance Comparison for Normal Operation of *ChatApp* for 100 Users (Virtual Threads and RxJava).

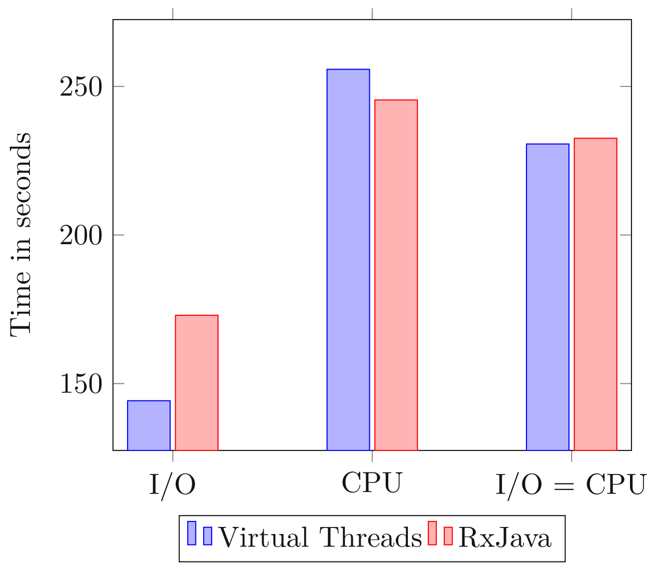

Figure 6.20: Experiment C – Time Performance Comparison for Normal Operation of *ChatApp* for 1000 Users (Virtual Threads and RxJava).

Similar to previous cases where we compared the memory consumption of virtual threads and RxJava, the results here exhibit the same trend: memory consumption remains nearly identical for both implementations.
From a time and memory performance perspective, virtual threads serve as strong competitors to RxJava. Experiments show that virtual threads have advantages when the majority of concurrent tasks are I/O-bound. RxJava has still advantages by handling asynchronous streams of data and by providing built-in backpressure handling.

6.2.6 Considerations on Experiment Results Validity

Every experiment comes across some factors which can influence its validity and have direct or indirect impact on the results. In different sources like [48] and [43] authors distinguish between two kinds of validity threats: external and internal threats. The external threats concern how much the results of experiments are generalizable. The internal threats, in their turn, concern the accuracy of data provided for experiments.
The domain-specific actions of the scenario have influence on performance as an external threat. The utilized technologies and the database can provide delays and damp the throughput of tasks being executed. Variability in developer skills, experience, and coding styles can affect the performance and quality of software experiments can influence the repeatability of experiments especially if the scenario or the benchmark will be implemented again by other software engineers. New versions of libraries used in the implementation may fix some software vulnerabilities and bugs in the future which can lead to an unpredictable increase or decrease of performance for both implementations of concurrent scenarios. In order to mitigate external threats we used the latest versions for

all external assets we used during the development, conducted the experiments on the same multicore machine, provided a documentation of all steps, decisions and configuration details to facilitate the replication of experiments in the future.
Internal threats to validity in software engineering experiments are factors within the study itself that can lead to incorrect conclusions about causal relationships between the independent and dependent variables. These threats can stem from issues such as study design, participant behaviour, or implementation errors, making it essential to identify and mitigate them effectively.
Due to non-deterministic behaviour of threads and randomized nature of the experiments, some tests can have other surprising results. The expectations or beliefs of the experimenter can unconsciously influence the interpretation of results. In order to mitigate the internal threats, we conducted experiments several times and used the statistical mean as final result, used no-human-intervention tests to exclude unintended altering of experiment data or results of human nature.

6.2.7 Summary of Performance Evaluation Section

In this subsection we will answer research questions number 2 and 3.

RQ2. How does the performance differ between threads and reactive programming approaches in different forms of concurrency organization?

RxJava tends to provide superior time performance compared to Java platform threads in the majority of real-world applications, especially those that involve asynchronous, I/O-bound, or highly concurrent tasks. RxJava minimizes time-consuming operations like thread creation and context switching while ensuring non-blocking execution through its reactive model. By efficiently managing a small pool of threads and processing tasks as they arrive, RxJava excels in scenarios requiring low latency and high scalability.
However, for CPU-bound tasks or simple systems with minimal concurrency requirements, traditional threads may still offer competitive performance, making them a better fit in specific cases where the overhead of RxJava is unnecessary (especially if the expected number of threads is relatively small).
One of the most significant benefits of RxJava over traditional threads is its ability to scale while maintaining low latency. In a thread-based approach, as the number of threads grows, so does the overhead of managing them, leading to increased task latency as the system struggles to juggle active threads. The problem becomes more pronounced when thousands of threads are required, such as in high-concurrency environments like web servers or real-time systems. RxJava, by efficiently reusing threads and processing tasks asynchronously, maintains low latency even under heavy load. Its event-driven architecture ensures that tasks are processed as soon as they are ready, minimizing delays. This makes RxJava highly scalable and well-suited for modern applications where high

concurrency and low response times are critical.
In applications that involve long-running, CPU-bound tasks, where each thread has a significant amount of work to do, the overhead of RxJava's operators and schedulers might not provide any real benefit. In such cases, the direct execution model of threads may offer better time performance, as each thread can work independently without the additional layers of abstraction RxJava introduces.
Java virtual threads, introduced recently, excel in I/O-bound workloads by efficiently handling huge amounts of lightweight threads and shows even better time performance compared to RxJava. However, for CPU-bound tasks platform threads remain preferable since virtual threads do not significantly improve performance, and RxJava's scheduling overhead may not provide benefits especially for small amount of concurrent tasks. In mixed workload cases, where I/O-bound and CPU-bound tasks are both present in a mixture, virtual threads and RxJava show both comparable and better performance than platform threads.

RQ3. How does memory consumption differ between threads and reactive programming approaches in different forms of concurrency organization?

Traditional platform threads create a new thread for each concurrent operation, which increases memory usage, especially when managing large numbers of tasks. RxJava, by contrast, manages concurrency through a small set of worker threads from pools, using much less memory. With platform threads, each created thread occupies memory, even when idle. RxJava doesn't suffer from this issue, as it efficiently reuses a few threads, making it more memory-friendly.
For traditional Java threads exists a possibility to reduce (and also to increase) stack memory usage via special parameter (*-Xss<size>*). Too large stack memory reduction can lead to *StackOverflowError* when allocated stack memory capacity is not enough for handling a task associated with a thread.
RxJava generally consumes less memory than threads, especially in applications with high concurrency, due to its use of thread pools and a non-blocking, event-driven concurrency model.
Java virtual threads consume significantly less memory compared to platform threads because they are lightweight and are managed by the JVM rather than the OS. Unlike platform threads, which require a dedicated OS thread and a large stack, virtual threads use a small, dynamically growing stack, reducing overall memory usage. This allows the JVM to handle millions of Virtual Threads efficiently without excessive memory overhead. The overall memory consumption of Java virtual threads is less or equal to RxJava memory consumption.

6.3 Maintainability

In the Macmillan English dictionary the term "Maintainability" is defined as ease with which a software developer can repair, improve, and understand software code written by himself or by another software developer. In general, the software maintenance phase in the software development life cycle starts after the customer has received the software product. As a rule, developers take care of maintenance by continuously adapting software to meet new customer requirements and address problems faced by customers.
The maintainable software product can be defined as software which is relatively easy to analyse, modify, and test. Another important feature is the time and ease of the learning process of the technology with which the software is written. It is quite natural, because if the developer cannot understand the technology behind the code in a feasible time, then corrections of software bugs will also take longer and this will influence the trust of customers.
This section is devoted to the comparison of maintainability features for both implementations made while writing this thesis.

6.3.1 Analysability and Modifiability

We define *Analysability* (based on [29], [1]) as the ease with which a system, codebase, or component can be understood, examined, and assessed by a developer or a maintainer. It is one of the critical software quality attributes that determines how easy it is to:

- Understand the functionality and behaviour of the code;
- Trace the flow of execution or data through the system.
- Identify issues such as bugs, performance bottlenecks, or errors.
- Verify correctness to ensure the system meets its intended design and requirements.

Modifiability refers to how easily a system or software component can be changed, adapted, or extended without introducing errors or breaking existing functionality [29], [1]. It is an important quality attribute in software development, as it affects the ease with which a system can evolve over time to meet new requirements, fix bugs, or improve performance.
When discussing the analysability of code that uses threads, there are several challenges and factors to consider due to the complexity of multithreaded programming. As discussed earlier, threads give an ability of low-level control over task executions, but in many cases it requires more code and increases its complexity. Nevertheless, there is no hidden "magic" and all the steps during the thread lifecycle can be analysed having the source code.
One key factor of impacting the analysablity of code written with threads is the synchronization complexity. On the one hand, no synchronization can lead to diverse hazards like race conditions, deadlocks, etc. Analysing the code to detect potential hazards can be very challenging, especially in large codebases. On the other hand, synchronization

mechanisms can ensure safe access to shared data. However, it is often difficult to analyse whether the synchronization was properly implemented. Incorrect synchronization can cause either data corruption (if too little synchronization is applied) or performance degradation (if too much synchronization is used). For developers it is rather complicated to find a balance. This issue also strongly influences the modifiability, because each modification must also have this balance and especially after the modification is deployed, no new hazards must appear [24].

As a case study, we examine the *ChatApp* application, specifically its registration functionality. During the registration process, the user provides multiple input fields, which are subsequently stored in the application's database. In this context, it is essential to ensure data integrity, meaning that all user-provided inputs must be accurately stored in the database without any alterations. Without proper synchronization, simultaneous registration attempts by multiple users may result in data being stored in a disordered or unpredictable manner, leading to potential data corruption or mixing of user inputs. If the entire registration process, including user input and subsequent database storage, is locked while one user is interacting with the system, all other threads will be blocked and forced to wait until the process is completed. This will prevent any concurrent interaction with the system during that time. It is evident that both a lack of synchronization and excessive synchronization are problematic.

In order to analyse if there is enough synchronization we can identify critical sections. In our example it is writing to the database. We can define user input variables as thread local and synchronize only the database access. In this way we guarantee that other users will be able to make their inputs.

Another factor worth to mention while describing the analysability of code written with threads is the non-deterministic behaviour during execution. Since the timing and order in which threads are scheduled by the operating system can vary, it is difficult to predict the exact sequence of actions, which complicates the identification of bugs and performance issues. The so-called *Heisenbugs* can appear. These are bugs that seem to disappear when developers try to debug or observe them. Thread-based code is notorious for producing such bugs because the timing of thread execution can change when debugging tools are used.

Such unpredictability is also present in *ChatApp*. We can never guess the order of clients who will receive a broadcast message. Another example, if two or more users try to login into the system simultaneously, it is absolutely unpredictable who will be first logged in. When multiple threads are executing simultaneously, error logs and stack traces can become cluttered with information from different threads. Analysing stack traces and logs to pinpoint the source of an error or issue can be overwhelming, especially in systems with many threads. To address this issue, we implemented thread names in the logs of *ChatApp* as a means of identifying the origin of each log entry. This allowed us to track which thread generated each log. However, as the number of simultaneous users increases, the volume of log entries grows significantly. The rapid creation of logs results in a rate that overwhelms the developer, making him/her difficult to detect bugs or warnings in real-time.

Threads often share state through shared variables or objects. Analysing how the state changes across multiple threads and ensuring that data is not being corrupted by concurrent accesses (due to improper synchronization) is also a challenging task.
All the points listed above indirectly impact the modifiability of software with threads. If the analysis of the program flow cannot be done properly, it is rather complicated to modify the software without producing errors. Low-level control makes modifying the code difficult because changes (e.g., introducing new threads or adjusting thread behaviour) can lead to unforeseen issues such as performance bottlenecks or thread contention. When modifying thread-based code, it's essential to ensure that new code is thread-safe. This requires careful consideration of how new variables or objects are accessed by multiple threads. Any change in shared state or business logic can require significant adjustments to ensure thread safety.
For example, consider a scenario where the requirement is to add a key-value database (i.e. Redis) for storing some metadata of users. In this case we will need to rethink the synchronization concept with an intent to provide exclusive access to relational and key-value databases. We will need, for instance, to properly define minimal critical sections, because again, too large synchronization blocks can lead to problems which user can notice.
Now we consider the analysability and modifiability of code written with RxJava. RxJava promotes the declarative paradigm of programming where developers describe what should happen instead of how it should happen. On the one hand this can improve the analysability of code because many low-level operations are hidden which makes logic easier to follow. On the other hand, this feature makes code less intuitive and in complex systems, where a variety of operators for composing different streams and observables are used, it can be very complicated for developers to analyse what the code is intended to produce.
One of RxJava's strengths is its abstraction over concurrency. Instead of explicitly managing threads, developers use schedulers to control where and how operations are executed. This simplifies the analysis of how asynchronous code is executed because there is no need to track threads explicitly. But in the same time the analysability can be reduced because developers cannot understand or follow the flow of data. Since RxJava handles streams asynchronously, it can be difficult to analyze exactly when certain events are emitted and how they interact with other parts of the system, especially when multiple observables are combined.
In *ChatApp* we use the abstract class ReactiveWebSocket and both client and server extend it. For this reason there are many nested callbacks for some functions (i.e., onMessage()) which can be hard to analyse. The *ChatApp* is a relatively small application, but in bigger applications the code analysis time can significantly increase.
Just as with threads, when errors occur in RxJava streams, the stack traces can be difficult to analyse because they often do not show the exact source of the problem. Different threads can produce various log or error information simultaneously making stack traces messy and hard to read and understand. Logs produced for *ChatApp* are not an exception because they are barely readable and it is nearly impossible to understand

what really happened in case of an error.
Talking about the maintainability of code written with RxJava, we need to keep in mind that the developer should be familiar with reactive programming and its concepts, because otherwise the code may seem unmaintainable "magic". Nevertheless, RxJava simplifies management of concurrency via abstracting all low-level operation making code more precise and elegant. This in its turn can help to increase maintainability, because developer can rely on basic synchronization and lifecycle objectives and concentrate on the feature itself.
Another advantage for analysability of RxJava compared to threads is a use of generic types. Types serve as documentation for developers, making code easier to understand. When the type of a variable or function return value is explicit, it is immediately clear what kind of data is being dealt with, reducing the cognitive load for anyone reading or maintaining the code. In *ChatApp* generic types made diverse refactorings easier and safer. Changing of a type immediately highlights all locations where type is used incorrectly.
RxJava provides a reach set of powerful operations for transformation and composition of reactive streams. This makes the codebase also more maintainable due to reduction of boilerplate code. However, the amount of operations available can lead to misuse and unnecessary complication of software code, which can influence maintainability in disadvantageous direction.
As we have shown above, threads can make analysability difficult due to issues like race conditions, deadlocks, non-deterministic behaviour, and synchronization complexity. Modifiability is also not so simple because of hidden bugs and potential bottlenecks if additional threads will be implemented. RxJava usage compared to threads can lead to better analysability and modifiability especially if the codebase does not contain complex code.

6.3.2 Testability

In general, testing of concurrent programs is more complicated compared to sequential ones due to non-determinism provided by concurrency. The same techniques for testing correctness and performance in sequential programs can be applied to concurrent programs, but in the last ones more things can go wrong.
The development of unit tests for concurrent classes starts with the same analysis as for sequential classes – identifying invariants and postconditions that must hold during and after program execution. Once this information is identified, it is advisable to test the code by isolating it from any concurrent behaviour. This statement is valid for both implementations: using threads, and using RxJava. In this way the logic of the code and the behaviour of the program for one single thread can be tested and fixed if problems occur. In the case of *ChatApp*, the same tests can be reused for both implementation strategies (because both software products have the same set of features).
Testing the essential concurrent behaviour requires more than one thread and can be performed after the developer is sure that the program behaviour is correct for a single thread.

Testing the concurrent behaviour of *ChatApp* implemeted using Threads.
Testing concurrency properties requires the introduction of additional threads. From the previous chapters we are aware that threads are non-deterministic and can be scheduled by operation systems unpredictably. This brings us to the point where issues like race conditions and deadlocks can arrive. It is also worth to mention, that sometimes the tests are working with two or three threads, but as the number of threads increases, the tests begin to fail. Non-determinism implies that given the same code and shared resources, the program can follow different execution paths, resulting in different outputs or no results.

Testing for thread safety in concurrent classes is often called a chicken-and-egg-problem: the test program itself is a concurrent program. Developing good concurrent tests can be more difficult than developing the classes they test [24]. In literature there is no perfect method for thread safety tests available. Some authors like for example Brian Goetz ([24]) describe a method which uses a producer-consumer-design. The idea behind the method is to check that everything put into a "queue" comes out of it, and does nothing else. The author proposes to compute checksums of elements that are enqueued and dequeued using an order-sensitive checksum function, and compare them. Unfortunately, in multi-producer and multi-consumer situations there will be a need to synchronize the shared checksums. This brings a bottleneck situation and can dramatically extend the time for running the tests.

Other authors like Matthew Sottile [72] propose a method where the so-called *thread interleavings* are analysed. Thread interleavings can be best described based on comparison with footprints in the snow. Each thread accesses shared data, modifies it and leaves its footprints called *thread interleavings*. In this approach the analysis of these interleavings can guarantee that no race condition occurs and consequently, the thread safety is present. Unfortunately, the analysis can take a huge amount of time if the number of threads increases. The control of interleavings can be guaranteed only by a relatively small amount of threads.

Testing for liveness poses a complex problem for researchers from all over the world. The overall decidability of deadlock/starvation occurrence and fair termination for concurrent problems is discussed in some articles, for example, in [6]. Nevertheless, there are several approaches that allow to test if some thread blocks when it is needed, and some ideas how to check if this thread continues its execution after the blocking state.

One relatively simple approach is to test the concurrent software for blocking methods. Under some defined condition one thread should be blocked from the execution because another thread is in a critical section, then a test for such behaviour should succeed only if the original thread does not proceed. This test is comparable to classical tests for throwing an exception, if the method proceeds normally, the test has failed.

For instance, in *ChatApp* only one thread at a time can broadcast a message to other threads. If during the test all the threads complete and no thread comes into a blocking state, then the test fails, because in this case data integrity cannot be guaranteed.

But testing if the method blocks does not test the code for liveness. Once the method successfully blocks, there is a need to be convinced that it will eventually unblock and continue processing. The intuitive way is to insert an interruption into a running thread. It works as follows: assume some thread is inside the critical section and another thread is blocked and waits for execution. If we insert an interruption into a running thread, and then assert that the blocking thread completes, in this case the test is positively finished. This method works fine in theory, but in practice, the developer has to make an arbitrary decision about how long the interruption should be, how long the instructions inside the critical section will take to complete, and wait a little bit longer than that. For many cases this is not a trivial task [24].

Testing the concurrent behaviour of *ChatApp* implemeted using RxJava. RxJava provides a developer with special classes used for testing: *TestObserver* and *TestSubscriber*. With use of both of them the developer is able to assert that certain events have occurred or specific values were received. They also provide blocking methods (i.e. *awaitTerminalEvent()* which stop the calling thread until the reactive operation finishes its work. As a rule *TestObserver* is used for *Observer*, *Single*, *Maybe*, and *Completable* classes, while *TestSubscriber* is used for *Flowables* [49].

The use of *TestObserver* and *TestSubscriber* for testing *ChatApp* can, for example, facilitate tests where sent and received messages are compared, and also can help in tests where the intent is to count the users which are connecting or disconnecting to/from the chat. Unfortunately, due to the fact that sometimes we had to stop the main thread from racing past an *Observable* or *Flowable* that operates on a different thread, some operations can take a long time to complete.

The *TestScheduler* is a Scheduler implementation that allows actions to be executed either immediately or in the future. The *TestScheduler* controls how the *Observable* interprets time and pushes emissions. It provides several methods for the convenience of developers: *advanceTimeTo()* will execute all actions that are scheduled for up to a specific moment in time; *advanceTimeBy()* advances time relative to the current moment in time; *triggerActions()* does not advance time. It only executes actions that were scheduled to be executed up to the present.

Unfortunately, *TestScheduler* is not a thread-safe Scheduler, for this reason the so called *scheduling collision* can occur, where actions are scheduled for the same time [49]. The order in which two simultaneous tasks are executed is the same as the order in which they where scheduled.

The combination of classes and methods mentioned above gives an opportunity to test reactive concurrent applications written in RxJava and assert almost all possible invariants and postconditions that must hold during and after the program execution.

Testing of concurrent applications is not trivial and requires profound knowledge and understanding of the software product itself. Successfully performed tests written for an application developed using threads does not guarantee thread safety and liveness of software. This means, race conditions and other hazards can potentially occur even if all the tests are successfully executed. Such tests are also concurrent and sometimes the developer is not sure if the test fails because the concurrency in the program or in test itself contains an error.

For RxJava, developers need to have profound knowledge in reactive programming and

fully understand reactive concepts to be able to effectively test the reactive software application. Special classes and schedulers provided in RxJava for tests sometimes give a feeling of more control while testing.
Unfortunately, in both cases (RxJava and Threads implementation) tests cannot fully guarantee the absence of problems caused by concurrency. As a well-known wisdom says, the safest multi-threaded code is the single-threaded one.

6.3.3 Effort for Developers

This subsection is not devoted to complex estimations of software development effort in time units. The intention here is to summarize the experience gained during the development of the concurrent scenario using both strategies and to compare the difficulty level for implementing the same software using threads and RxJava.
Threads are classical and well-known by many developer tools for handling concurrent problems. Almost every developer who deals with concurrency has to understand threads. The parallelisation of tasks (in the sense of simultaneous execution) requires some understanding and mastering. Threads provide fine-grained control over concurrency and give a good opportunity for practising in order to "gain the feeling" of concurrent tasks executions.
Writing thread-safe code requires careful consideration of thread hazards and a proper application of synchronization mechanisms. Developers must provide sound error handling and resource cleanup after software execution.
The author of this thesis has already had some experience using threads in some university projects in different programming languages. The *ChatApp* development effort for the application implemented with threads is approximately equal to 50 hours, where 15 hours were used for the knowledge recapitulation; 20 hours were used for software development; and the remaining 15 hours were used for refactorings and bug fixes.
RxJava simplifies concurrency by abstracting away low-level operations, introduces built-in error handling, increases scalability, and hides thread management making readability of code easier. At the same time, RxJava requires profound understanding of reactive concepts, introduces a large amount of operations which take time to understand, and makes code less intuitive due to abstraction.
RxJava introduces a higher initial learning curve because developers need to understand reactive programming principles, observables, and operators. Since RxJava uses a functional, declarative style of programming, developers unfamiliar with these paradigms may initially invest more time and effort in learning the technology.
For the author of this thesis RxJava was completely new technology. The *ChatApp* development effort for the application implemented with RxJava is approximately equal to 130 hours, where 60 hours were spent for knowledge gain and practice; another 60 hours were spent for program development and fixing of bugs; the remaining 10 hours were spent for diverse refactoring tasks.[5]

[5]The approximate spent time values provided in this subsection does not include the development of benchmark and software tests.

Carlos Zimmerle et al. [85] performed a data mining analysis of ReactiveX-related projects on GitHub and StackOverflow to explore the challenges encountered by developers working with reactive programming. Their findings reveal that the most significant difficulties faced by developers globally are the introduction to reactive programming, stream abstraction, and dependency management. Moreover, testing and debugging of reactive programs emerged as areas of substantial concern for developers.
Conclusively, due to its benefits skilled developers can have less effort in implementation of *ChatApp* using RxJava compared to traditional threads. Unexperienced developers have to invest more time for learning all the features RxJava provides and may implement the *ChatApp* using threads more promptly.

6.3.4 Summary of Maintainability Section

In this subsection we will summarise the knowledge gained in this section and answer the research question number 4. RxJava improves the analysability of code compared to traditional thread-based approach by providing a higher-level abstraction over concurrency, and powerful operators for data transformation. An additional advantage of RxJava over traditional threading is its utilization of generic types, which enhances code analysability and maintainability. However, its flexibility and complexity can make it harder to analyse the code in certain situations, particularly when dealing with deeply nested streams, complex operator chains, or subtle concurrency issues.
The analysability of code involving threads is inherently complex due to the non-deterministic nature of thread execution. Synchronization issues, such as race conditions and deadlocks, often arise and are challenging to detect and resolve. Debugging multi-threaded code is further complicated by the interleaving of thread execution, which can produce different outcomes under varying conditions. Analyzing the flow of control and data dependencies requires specialized tools and techniques. Despite these challenges, proper design patterns, thorough testing, and the use of abstractions like thread-safe libraries can significantly enhance the clarity and maintainability of threaded code.
RxJava enchances testability with built-in support for operators and schedulers that allow easier mocking and controlled execution. Threads often require additional steps and, in some cases, more complex algorithms for testing, particularly when ensuring the avoidance of concurrency hazards.

RQ4. How does the developing effort of the same scenario differ while developing the solution using threads and using reactive programming?

Unfortunately it is quite problematic to measure the developer's effort due to various levels of experience, willingness to learn, ability to learn, readiness to change of thinking patterns, etc by developers around the globe. Development effort can also differ in terms of pure developing and further code analysis in order, for example, to fix a bug or to

extend code with new features. Consequently, an effort consists not only from resources needed to learn the technology and to develop the runnable code, but also emphasizes time needed to understand already written code, time needed to insert changes and to test the software product.
Threads require a high development effort, especially for managing concurrency, synchronization, and error handling. The code can become verbose and prone to errors, and the debugging process is more complicated due to common concurrency issues like deadlocks and race conditions. RxJava, while abstracting much of the complexity of thread management, introduces its own challenges in terms of learning the reactive paradigm, understanding how streams and operators work, and managing flow control (like backpressure). However, RxJava is able to reduce effort for concurrent applications development for experienced developers by providing enhanced composability, built-in error handling and improved testing.

CHAPTER 7

Conclusion and Future Work

He knew that all the hazards and perils were now drawing together to a point: the next day would be a day of doom, the day of final effort or disaster, the last gasp.

— J. R. R. Tolkien, *Lord oh the Rings*

This chapter is devoted to conclusions of the research conducted in boundaries of this thesis. We give answers to research questions, summarize gained knowledge and give ideas for the future work.

7.1 Conclusion

In this thesis, we compared the traditional Java concurrency approach, which uses threads, and the most recent Java virtual threads with the reactive programming paradigm implemented through RxJava. Our focus was on their suitability for concurrency, performance, memory consumption, and development effort. By differentiating between stateless and stateful threads, as well as I/O-bound, CPU-bound, and mixed workloads, this research provides a nuanced understanding of when and why reactive programming is advantageous compared to traditional thread-based approaches. To achieve this, we implemented a concurrent scenario, developed, and conducted a suite of experiments.
We conclude that RxJava is well-suited for addressing concurrency challenges due to its reliance on an event-driven concurrency model, which efficiently manages a high volume of concurrent tasks using a limited pool of reusable threads. Furthermore, RxJava aligns with the declarative programming paradigm, enabling developers to specify what the system should accomplish without being burdened by the intricacies of how it is implemented. However, programs written with RxJava are often less intuitive and require a deep understanding of reactive concepts. The primary disadvantage of the event-driven concurrency model is the distribution of task execution across multiple threads and

locations in the code.
RxJava provides users with built-in mechanisms for error handling and software testing. Moreover, it offers a rich set of built-in operations that simplify combining, transforming, and synchronizing the execution of multiple asynchronous streams. On the other hand, RxJava suffers from a lack of centralized error management, the potential for hidden errors (depending on the implementation of error handling), and relatively complex code analysis and modification.
Java threads rely on the imperative programming paradigm, where all operations (creation, execution, synchronization, and deletion of threads) must be implemented manually. While this leads to a significant amount of boilerplate code, it also gives developers full control over the entire lifecycle of threads.
Threads are based on the parallel workers concurrency model, where each request is handled by a new thread. This model is well-known, reliable, and easy to implement, but it can be resource-intensive and may impact performance, especially as the number of threads increases.
Analysability, modifiability, and testability of thread-based programs depend on the complexity of synchronization mechanisms, often requiring additional methods and techniques. Testing and analyzing whether a concurrent program implemented with threads is free of safety and liveness hazards remains a topic of discussion in books and articles. While several approaches exist, none guarantees the complete absence of such hazards.
Java virtual threads were introduced recently in Java 19 with an aim to provide lightweight threads to minimize disadvantages of Java platform threads. Unlike platform threads, virtual threads are managed by the JVM and do not require a dedicated OS thread, allowing for efficient execution of millions of threads. They are particularly well-suited for I/O-bound workloads, as they eliminate thread-blocking inefficiencies without consuming excessive system resources.
RxJava generally demonstrates superior time performance compared to traditional threads in scenarios involving asynchronous operations, I/O-bound tasks, or high levels of concurrency. However, in the applications that involve long-running CPU-bound tasks, where each thread is occupied with a significant amount of work, the overhead of RxJava's operators and schedulers might not provide any real benefit.
Virtual threads provide superior performance compared to RxJava for scenarios that include I/O-bound tasks. However, they do not significantly improve performance in CPU-bound tasks, as they do not provide additional parallelism beyond what the available CPU cores allow. Virtual threads show similar time performance as RxJava in scenarios where CPU-bound and I/O-bound tasks are both present in a tasks pool.
RxJava generally consumes less memory than platform threads due to its non-blocking nature and reuse of threads. However, virtual threads consume either less or equal amounts of memory than RxJava in all of scenarios above.
The development effort associated with threads is challenging to quantify, as threads have become the de facto standard for addressing concurrency issues. Consequently, many developers acquire familiarity with threads early in their careers. Threads typically require minimal time to learn due to their widespread use and familiarity among

developers. However, implementing a concurrent application with threads necessitates explicit management of various lifecycle mechanisms, including synchronization and error handling, which can significantly increase development complexity. In contrast, RxJava demands a significant investment of time for developers to learn due to its steep learning curve. However, it abstracts much of the complexity associated with thread management, resulting in more concise and declarative code.

Platform threads are well-suited for simple, low-concurrency tasks or scenarios requiring fine-grained control over thread lifecycle and resource management. They are ideal for developers already familiar with their mechanisms and for systems where CPU-bound, long running tasks prevail. Virtual threads are best for highly concurrent I/O-bound applications. Conversely, RxJava excels in high-concurrency environments, particularly for asynchronous workloads, where its event-driven model efficiently handles a large number of tasks with minimal resource usage, though it requires developers to have a deep understanding of reactive programming concepts. The choice between the two depends on the specific requirements for control, performance, and ease of development in the given context.

RxJava and virtual threads offer similar and superior time and memory efficiency compared to platform threads in cases where the workload and tasks simulate real-world conditions.

7.2 Ideas for Future Work

In this master thesis we made a comparison of concurrent problem solutions implemented with traditional Java threads and a reactive approach, namely RxJava. There are many other alternatives to RxJava available, for example, Reactor Core, Reactive Streams, RSocket, mitiny, Reactor, etc. There are also plenty alternatives in other programming languages (especially in Kotlin). For the future work it can be interesting to make a comparison with other reactive approaches [15].

For the real-world scenario it would be interesting to investigate performance and memory consumption in cloud and distributed systems. Testing RxJava and Java threads within cloud environments or distributed microservice architectures can provide insights into real-world performance under network latency, load balancing, and fault tolerance. Analyzing memory and CPU usage in such distributed systems can highlight how each threading approach scales in production environments with complex, dynamic workloads.

Another idea for future work is a comparative study on error handling and debugging strategies of both RxJava and threads. The intent is to investigate the debugging and error handling differences in detail, specifically in cases of nested tasks or long task chains, and analyse how RxJava's error handling operators compare to traditional try-catch methods. Debugging of code is not a simply task for both strategies either.

Since energy efficiency and sustainability is an increasingly important metric in large-scale applications, a future study could evaluate the power consumption and efficiency of RxJava versus traditional Java threads. This research would help to understand the sustainability implications of using reactive frameworks, especially in mobile and IoT

environments [36] [14].
As serverless computing and edge computing become more prevalent, testing RxJava and Java threads in these environments would be insightful. Understanding how well each model adapts to serverless architectures, where functions are short-lived and stateless, could drive the adoption of optimal concurrency patterns in edge devices and cloud functions [62].

APPENDIX A

Observer Pattern

The following explanations are inspired by the book "Design Patterns" written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. [23]

The observer pattern is a behavioral design pattern that establishes a one-to-many relationship between objects. When the state of one object (the subject) changes, all its dependent objects (observers) are automatically notified and updated. It focuses on managing interactions and communication between objects, particularly how they respond to changes in each other's state.
A real-world example of the Observer pattern is the newsletter subscription system. The newsletter service is the subject (the publisher). Users who subscribe to the newsletter are observers (subscribers). When the newsletter service publishes a new article (state change), all subscribed users automatically receive an email notification (an update).

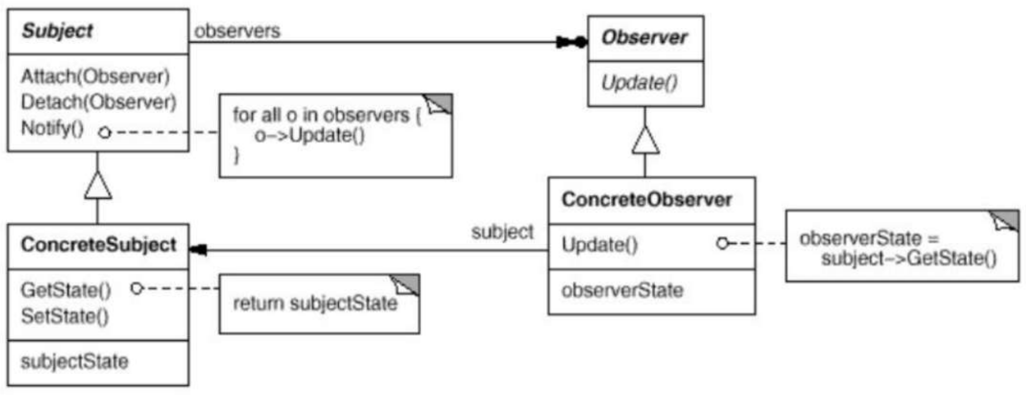

Figure A.1: Observer Pattern Structure. [23]

Structure. The structure of the observer pattern is depicted in Figure A.1.
The observer pattern contains following elements (based on [23]):

- **Subject** defines an interface for observers to subscribe. The subject always maintains awareness of its observers.
- **Observer** establishes an interface for notifying objects that need to be updated when the subject changes.
- **ConcreteSubject** maintains the state relevant to ConcreteObserver objects and notifies them whenever the state changes.
- **ConcreteObserver** holds a reference to a ConcreteSubject and implements the observer update interface to synchronize its state with the subject's state.

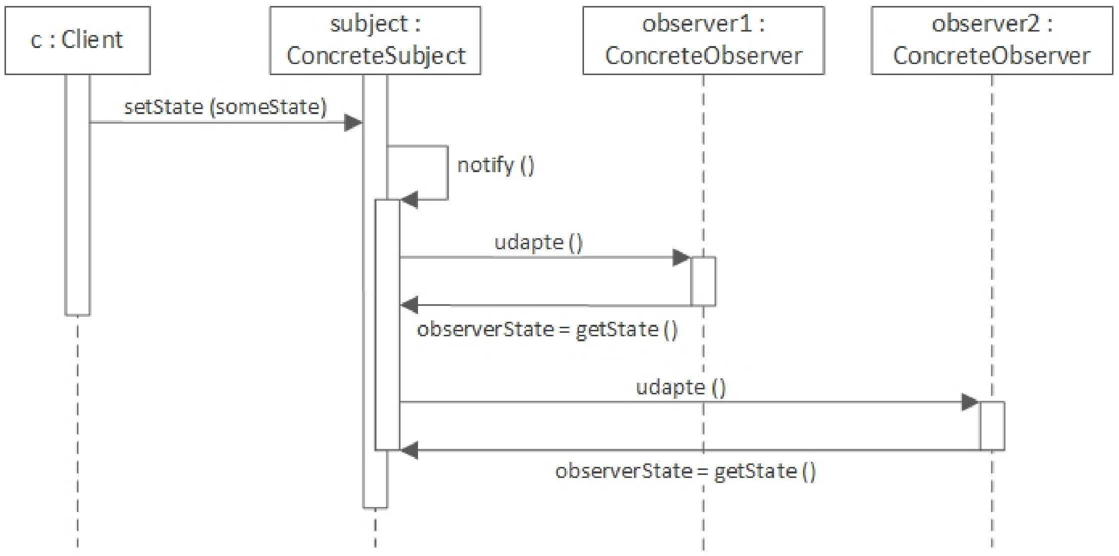
c : Client
subject : ConcreteSubject
observer1 : ConcreteObserver
observer2 : ConcreteObserver
setState (someState)
notify ()
udapte ()
observerState = getState ()
udapte ()
observerState = getState ()

Figure A.2: Sequence Diagram for Observer Pattern Interaction. [23]

Interactions. Observe the following sequence diagram depicted in Figure A.2, where a subject collaborates with two observers.
Note, after a client initiated the state change, the Subject is notified and sends an update message to each observer. In this way each observer gets notified of an update to the subject and obtains a new state of the subject. This flow is visualized in the sequence diagram presented in Figure A.2.
Applicability. The observer pattern is widely applicable in scenarios where multiple objects need to stay updated with changes in another object without tightly coupling them. It is particularly useful in event-driven systems, GUI frameworks, real-time data streaming, and publish-subscribe mechanisms. For instance, in a stock market application, multiple investor clients (observers) must receive real-time updates when stock prices

change (subject). Similarly, in UI design, button clicks can notify multiple event handlers dynamically. This pattern enhances modularity, allowing subjects and observers to evolve independently. It also improves maintainability and scalability, as new observers can be added without modifying the subject's core logic.

APPENDIX B

Amdahl's Law

Amdahl's Law (or also known as *Amdahl's argument*) is a formula that gives an opportunity to compute the theoretical speedup in latency of the execution of a task at a fixed workload that can be expected of a system whose resources are improved. It is named after the computer scientist Gene Amdahl[1].
Let P be a proportion of the system (or program) that can be theoretically made parallel. Then *(1-P)* is the proportion that remains serial. Then, according to Amdahl's Law, the maximum speedup *S(N)* that can be achieved using N processors can be computed as follows:

$$S(N) = \frac{1}{(1-P) + \dfrac{P}{N}} \tag{B.1}$$

As N grows, the maximum speedup converges to *1/(1-P)*. This means, that a program in which fifty percent of the processing must be executed serially can be sped up only by a factor of two, regardless of how many processors are available, and a program where ten percent must be executed serially can be sped up by at most factor ten.
Observe Figure B.1 where the ratio between the speedup and number of processors in dependency to parallelisation of programs is depicted.

[1]**Gene Amdahl** (November, 16. 1922, Flandreau, South Dakota, USA - November, 10. 2015, Plato Alto, California, USA) was an American computer scientist well known for his work on mainframe computers at IBM.

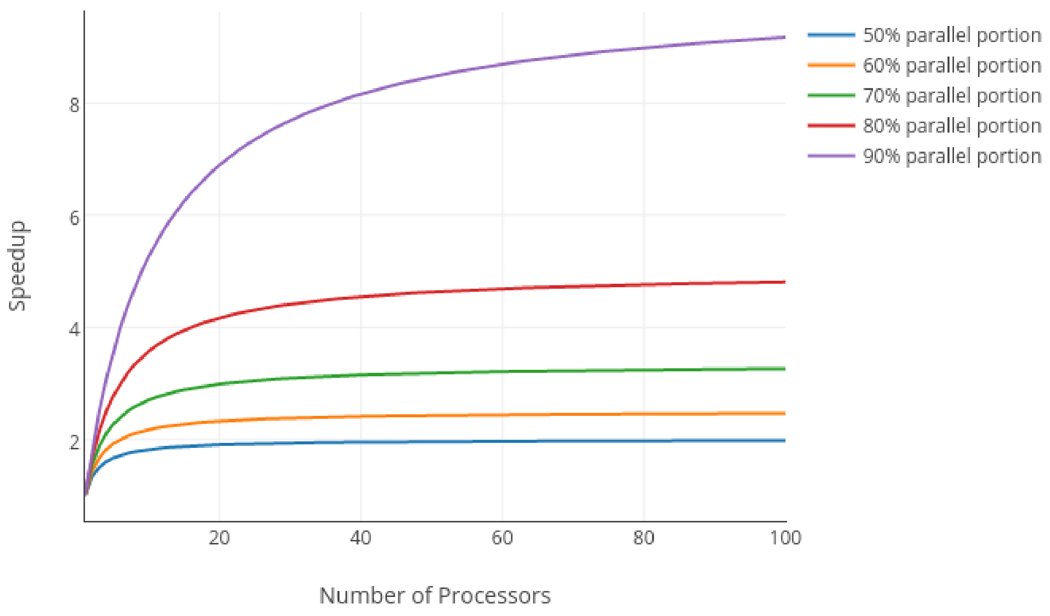

Figure B.1: Amdahl's Law. [28]

APPENDIX C

Overview of Generative AI Tools Used

List of all generative AI tools that were used in the creation of this work:

- **ChatGPT**, version 2 (released on November 6, 2024).
  This generative AI tool was used for the following cases:
  - generating short outlines of scientific papers with an intent to extract key insights and conclusions. This information was then used in order to understand if a scientific paper is relevant for this thesis and if it can be used for quotations or inspiration;
  - enhancing writing quality by improving sentences, especially those translated into English;
  - exploring novel perspectives on the topic of this thesis, which helped to write "Ideas for Future Work" section.

Bibliography

[1] IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990. `doi:10.1109/IEEESTD.1990.101064`.

[2] G. Agha, S. Frolund, W.Y. Kim, R. Panwar, A. Patterson, and D. Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(2):3–14, 1993. `doi:10.1109/88.218170`.

[3] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–43, 1983. `doi:10.1145/356901.356903`.

[4] Johannes Manuel Artner. Agile software performance engineering. Master's thesis, Wien, 2016. `doi:10.34726/hss.2016.39923`.

[5] Fede Barcelona. Monitoring java applications: Memory usage, threads and other JRE metrics. URL: `https://sysdig.com/blog/monitoring-java-jre/` (accessed: 19.03.2025).

[6] Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. Context-bounded verification of liveness properties for multithreaded shared-memory programs. *Proceedings of the ACM Programing Languages*, 5, 2021. `doi:10.1145/3434325`.

[7] M Ben-Ari. *Principles of Concurrent and Distributed Programming*. Pearson Education, Limited, Harlow, United Kingdom, 2006. ISBN: 9780321312839.

[8] Peter Bouda. How much memory does a java thread take? URL: `https://dzone.com/articles/how-much-memory-does-a-java-thread-take` (accessed: 19.03.2025).

[9] Clay Breshears. *The art of concurrency*. O'Reilly, Sebastopol, California, USA, first edition, 2009. ISBN: 0596551150.

[10] Peter A Buhr. High-level concurrency constructs. In *Understanding Control Flow*. Springer International Publishing AG, Switzerland, 2016. ISBN: 3319257013.

[11] David W. Bustard. Concepts of concurrent programming. SEI-CM-24. Software Engineering Institute; Carnegie-Mellon University, 1990. doi:10.1184/R1/6572699.v1.

[12] Paul Butcher. *Seven concurrency models in seven weeks : when threads unravel.* Pragmatic Programmers. Pragmatic Bookshelf, Dallas, Texas, first edition, 2014. ISBN: 1680504673.

[13] Egon Börger and Vincenzo Gervasi. *Structures of Computing: A Guide to Practice-Oriented Theory.* Springer International Publishing, Cham, 2024. ISBN: 3031543572.

[14] Coral Calero, Mª Ángeles Moraga, and Mario Piattini. *Software Sustainability.* Springer International Publishing Imprint: Springer, Basel, Switzerland, first edition, 2021. ISBN: 3030699706.

[15] Rivu Chakraborty. *Reactive programming in Kotlin: design and build non-blocking, asynchronous Kotlin applications with RXKotlin, Reactor-Kotlin, Android, and Spring.* Birmingham, England; Mumbai, India, first edition, 2017. ISBN: 1788470257.

[16] Oracle Corporation. Java platform, class thread, (2017). URL: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html (accessed: 19.03.2025).

[17] Oracle Corporation. Java platform, standard edition 7 api specification, (2017). URL: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.State.html (accessed: 19.03.2025).

[18] Danny Coward. *Java WebSocket programming.* McGraw-Hill, New York, first edition, 2014. ISBN: 0071827188.

[19] Adam L Davis. Existing models of concurrency in java. In *Reactive Streams in Java.* Apress L. P, USA, 2018. ISBN: 9781484241752.

[20] Adam L. Davis. *Reactive Streams in Java : Concurrency with RxJava, Reactor, and Akka Streams.* Imprint: Apress, Berkeley, CA, USA, first edition, 2019. ISBN: 1484241762.

[21] Ramez Elmasri and Shamkant B. Navathe. *Grundlagen von Datenbanksystemen.* Pearson Studium, München [u.a.], 3 edition, 2005. ISBN: 3827371538.

[22] Tatiana Fesenko. *Learn Java concurrency and multithreading in practice: harness the power of mulitthreading in Java.* Packt Publishing, first edition, 2019. URL: https://learning.oreilly.com/course/java-concurrency-and/9781789806410/ (accessed: 19.03.2025).

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Boston, USA, 1994. ISBN: 0201633612.

[24] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java concurrency in practice*. Pearson India Education Services Pvt. Ltd, Uttar Pradesh, India, 2017. ISBN: 9789332576520.

[25] Michael Goll. Websocket. In *JavaServer Faces*, pages 201–208. Springer Fachmedien, Wiesbaden, Germany, 2020. ISBN: 9783658318024.

[26] Jan Graba. *An Introduction to Network Programming with Java: Java 7 Compatible*. Springer Nature, London, United Kingdom, third edition, 2013. ISBN: 1447152549.

[27] Ulrik Bech Hald. Using flatmap() to parallelize streams and delay individual errors. URL: `https://medium.com/tech-at-unwire/parallize-streams-using-flatmap-and-delay-any-errors-5f194a56009a` (accessed: 19.03.2025).

[28] Vikas Hazrati. Digital transformation with Amdahl's and Gunther's Law, (2019). URL: `https://blog.knoldus.com/digital-transformation-with-amdahls-gunthers-law/` (accessed: 19.03.2025).

[29] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, pages 30–39. IEEE, 2007. `doi:10.1109/QUATIC.2007.8`.

[30] Jakob Jenkov. Java memory model. URL: `https://jenkov.com/tutorials/java-concurrency/java-memory-model.html` (accessed: 19.03.2025).

[31] Jakob Jenkov. Virtual threads. URL: `https://jenkov.com/tutorials/java-concurrency/java-virtual-threads.html` (accessed: 19.03.2025).

[32] Alfons Kemper and André Eickler. *Datenbanksysteme: eine Einführung*. R. Oldenbourg Verlag, München, Germany, 8 edition, 2011. ISBN: 3486598341.

[33] Manish Kolambe, Saurabh Sable, Venkatesh Kashivale, and Prajkta Khaire. Chat application. *International Journal for Research in Applied Science and Engineering Technology*, 10(4):1426–1428, 2022. `doi:10.22214/ijraset.2022.41548`.

[34] Jakob Korherr. Restful web applications with reactive, partial server-side processing in Java EE. Master's thesis, Wien, 2015. `doi:10.34726/hss.2015.32225`.

[35] Jeff Kramer and Jeff Magee. *Concurrency: State Models and Java Programs*. Wiley, New Jersey, USA, 2006. ISBN: 0470093552.

[36] Artem Kruglov and Giancarlo Succi. *Developing Sustainable and Energy-Efficient Software Systems*. Springer International Publisher, Switzerland, first edition, 2023. ISBN: 3031116585.

[37] Roland Kuhn, Brian Hanafee, Jamie Allen, and Jonas Bonér. *Reactive design patterns*. Manning, Shelter Island, New York, first edition, 2017. ISBN: 1638354057.

[38] Leslie Lamport. The computer science of concurrency: the early years. In *Concurrency: the Works of Leslie Lamport*, pages 13–26. ACM, 2019. doi:10.1145/3335772.3335775.

[39] Leslie Lamport. *A new solution of Dijkstra's concurrent programming problem*. Massachusetts Computer Associates, Inc., 2019. ISBN: 9781450372701.

[40] Angelika Langer and Klaus Kreft. *Java-Core-Programmierung : Memory Model und Garbage Collection*. entwickler.press, Frankfurt am Main, Germany, 2011. ISBN: 3868020756.

[41] Andreas Lochbihler. Making the Java memory model safe. *ACM transactions on programming languages and systems*, 35(4):1–65, 2013. doi:10.1145/2518191.

[42] Andrew Lombardi. *WebSocket*. O'Reilly, Sebastopol, CA, USA, first edition, 2015. ISBN: 1449369235.

[43] Ruchika Malhotra. *Empirical research in software engineering : concepts, analysis, and applications*. CRC Press, Taylor and Francis Group, Boca Raton, Florida, USA, first edition, 2016. ISBN: 0429183674.

[44] Alessandro Margara and Guido Salvaneschi. On the semantics of distributed reactive programming: The cost of consistency. *IEEE Transactions on Software Engineering*, 44(7):689–711, 2018. doi:10.1109/TSE.2018.2833109.

[45] Predrag Marić. Java connection pooling. URL: https://www.baeldung.com/java-connection-pooling (accessed: 19.03.2025).

[46] Andreas Meier. *Relationale und postrelationale Datenbanken*. Springer, Berlin [u.a.], 7 edition, 2010. ISBN: 364205255X.

[47] Marvin Minsky. *The society of mind*. Simon Schuster, Inc., New York, USA, 1986. ISBN: 0671607405.

[48] Amadeu Anderlin Neto and Tayana Conte. A conceptual model to address threats to validity in controlled experiments. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, page 82–85, New York, USA, 2013. Association for Computing Machinery. doi:10.1145/2460999.2461011.

[49] Thomas Nield. *Learning RxJava*. Packt Publishing, Limited, Birmingham, United Kingdom, 2017. ISBN: 9781787120426.

[50] Christoffer Noring. *Architecting angular applications with Redux, RxJS, and NgRx: learn to build Redux style high-performing applications with Angular 6*. Packt Publishing, Birmingham, UK, 1st edition, 2018. ISBN: 9781787122406.

[51] Rostislav Novak. Reactive websocket. URL: https://www.reactiveworld.net/2018/04/01/Reactive-web-socket.html (accessed: 19.03.2025).

[52] Tomasz Nurkiewicz and Ben Christensen. *Reactive programming with RxJava : creating asynchronous, event-based applications.* O'Reilly, Beijing, China, first edition, 2017. ISBN: 1491931604.

[53] Scott Oaks and Henry Wong. *Java threads.* O'Reilly, Sebastopol, California, USA, 3rd edition, 2004. ISBN: 144936666X.

[54] Ernst-Rüdiger Olderog. Correctness of concurrent processes. In *Mathematical Foundations of Computer Science 1989*, pages 107–132. Springer, Berlin, Germany, 2005. ISBN: 3540514864.

[55] Oracle. JDK release notes. URL: https://www.oracle.com/java/technologies/javase/jdk-relnotes-index.html (accessed: 19.03.2025).

[56] Oracle. Virtual threads documentation. URL: https://docs.oracle.com/en/java/javase/20/core/virtual-threads.html (accessed: 19.03.2025).

[57] Ahmad Oussous and Fatimah al-Zahra Bin Jilun. A comparative study of different search and indexing tools for big data. *Jordanian Journal of Computers and Information Technology*, 8(1):72–86, 2022. doi:10.5455/jjcit.71-1637097759.

[58] Ouya Pei, Zhanhuai Li, Hongtao Du, Wenjie Liu, and Jintao Gao. Dependence-cognizant locking improvement for the main memory database systems. *Mathematical problems in engineering*, 2021:1–12, 2021. doi:10.1155/2021/6654461.

[59] Julien Ponge, Arthur Navarro, Clément Escoffier, and Frédéric Le Mouël. 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*, page 51–60, New York, USA, 2021. Association for Computing Machinery. doi:10.1145/3486605.3486788.

[60] Julien Ponge and Martijn Verberg. *Vert.x in action : asynchronous and reactive Java.* Manning, New York, USA, 2020. ISBN: 1638353352.

[61] Herbert Prähofer. *Funktionale Programmierung in Java.* dpunkt, Heidelberg, Germany, 1st edition edition, 2020. ISBN: 3960889844.

[62] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing-where we are and what lies ahead. *IEEE internet computing*, 27(3):50–64, 2023. doi:10.1109/MIC.2023.3260939.

[63] James Ravn. JVM thread actual memory usage. URL: https://jsravn.com/2019/05/01/jvm-thread-actual-memory-usage/ (accessed: 19.03.2025).

[64] David Reilly and Michael Reilly. *Java network programming and distributed computing*. Addison-Wesley, Boston, Massachusetts, USA, 2002. ISBN: 0201710374.

[65] Nick Samoylov and Thomas Nield. *Learning RxJava: build concurrent applications using reactive programming with the latest features of RxJava 3*. Packt Publishing, Birmingham, United Kingdom, second edition, 2020. ISBN: 0201710374.

[66] Cleber Santana, Leandro Andrade, Flávia C. Delicato, and Cássio Prazeres. Increasing the availability of iot applications with reactive microservices. *Springer Nature: Service Oriented Computing and Applications*, 15(2):109–126, 2021. `doi:10.1007/s11761-020-00308-8`.

[67] Abhiroop Sarkar, Robert Krook, Bo Joel Svensson, and Mary Sheeran. *Higher-Order Concurrency for Microcontrollers*, page 26–35. Association for Computing Machinery, New York, USA, 2021. `doi:10.1145/3475738.3480716`.

[68] Michael L. Scott and Trevor Brown. *Shared-Memory Synchronization*. Springer, Switzerland, second edition, 2024. ISBN: 9783031386831.

[69] Baji Shaik and Avinash Vallarapu. *Beginning PostgreSQL on the Cloud: Simplifying Database as a Service on Cloud Platforms*. Imprint: Apress, Berkeley, CA, USA, first edition, 2018. ISBN: 1484234472.

[70] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating system concepts*. Wiley, Hoboken, New Jersey, USA, 8th edition, 2008. ISBN: 0470128720.

[71] Gregory Smith. *PostgreSQL 9.0: high performance*. Packt Publishing, Birmingham, United Kingdom, first edition, 2010. ISBN: 1282896474.

[72] Matthew J. Sottile, Timothy G. Mattson, and Craig E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman and Hall/CRC, Boca Raton, Florida, USA, 2009. ISBN: 9781420072136.

[73] Michael Spear, Arrvindh Shriraman, Luke Dalessandro, Sandhya Dwarkadas, and Michael Scott. Nonblocking transactions without indirection using alert-on-update. In *ACM Symposium on Parallel Algorithms and Architectures: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures; 09-11 June 2007*, pages 210–220. ACM, 2007. ISBN: 159593667X.

[74] William Stallings. *Operating systems*. Macmillan, New York, USA, first edition, 1992. ISBN: 0029464919.

[75] Maarten van Steen and Andrew S. Tanenbaum. *Distributed systems*. CreateSpace Independent Publishing Platform, third edition, 2017. ISBN: 1543057381.

[76] Riccardo Terrell. *Concurrency In . NET: Modern Patterns of Concurrent and Parallel Programming*. Manning Publications Co. LLC, New York, USA, 2018. ISBN: 9781617292996.

[77] Dionysios C Tsichritzis and Philip A Bernstein. *Operating systems*. Academic Press, New York, USA, 1974. ISBN: 012701750X.

[78] Nickolay Tsvetinov. *Learning reactive programming with Java 8: learn how to use RxJava and its reactive Observables to build fast, concurrent, and powerful applications through detailed examples*. Packt Publishing, Birmingham, United Kingdom, first edition, 2015. ISBN: 1785282506.

[79] Timo Tuominen. *RxJava for Android developers*. Manning Publications, Shelter Island, New York, 1st edition edition, 2019. ISBN: 1638351252.

[80] Alejandro Ugarte. What is thread safety and how to achieve it? URL: `https://www.baeldung.com/java-thread-safety#bd-synchronized-collections` (accessed: 19.03.2025).

[81] Helder Vasconcelos. *Asynchronous Android Programming - Second Edition*. Packt Publishing, Birmingham, UK, 2nd edition, 2016. ISBN: 9781785883248.

[82] Ron Veen and David Vlijmincx. *Virtual Threads, Structured Concurrency, and Scoped Values : Explore Java's New Threading Model*. Apress Media LLC, New York, USA, first edition, 2024. ISBN: 8868805006.

[83] John Clayton Worsley and Joshua D. Drake. *Practical PostgreSQL*. O'Reilly, Beijing, China ; Sebastopol, California, USA, first edition, 2002. ISBN: 1449310109.

[84] Yang Zhang and Jun-liang Chen. Constructing scalable internet of things services based on their event-driven models. *Concurrency and computation*, 27(17):4819–4851, 2015. `doi:10.1002/cpe.3469`.

[85] Carlos Zimmerle, Kiev Gama, Fernando Castor, and José Murilo Mota Filho. Mining the usage of reactive programming APIs: a study on GitHub and stack overflow. In *Proceedings of the 19th International Conference on Mining Software Repositories*, page 203–214, New York, USA, 2022. Association for Computing Machinery. `doi:10.1145/3524842.3527966`.