Technische Hochschule
Brandenburg
University of
Applied Sciences

Vergleich von reaktiver Programmierung und virtuellen Threads in Bezug auf Skalierbarkeit und Ressourcennutzung

Masterarbeit

zur Erlangung des Grades Master oft Science

des Fachbereichs Informatik und Medien der

Technischen Hochschule Brandenburg

vorgelegt von:

Maximilian Milz, B.Sc.

Betreuer: Prof. Dr.-Ing. Thomas Preuss
Zweitgutachter: Jonas Brüstel, M.Sc.

Brandenburg an der Havel, 24. Januar 2024

Kurzfassung

Das Ziel der Arbeit besteht darin, reaktive Programmierung und virtuelle Threads hinsichtlich ihrer Skalierbarkeit und Ressourcennutzung zu vergleichen. Hierfür werden drei Spring-Anwendungen entwickelt. Zwei dieser Anwendungen nutzen Spring MVC und unterscheiden sich in der Verwendung von nativen und virtuellen Threads. Die dritte Anwendung basiert auf Spring WebFlux, welches das reaktive Programmierparadigma umsetzt. Das Ziel dieser Arbeit ist es, die Skalierbarkeit und Ressourcennutzung verschiedener Anwendungen unter variierenden Lastbedingungen zu analysieren. Durch experimentelle Vergleiche und quantitative Auswertungen wurden signifikante Unterschiede in der Handhabung hochskalierbarer Anwendungen festgestellt. Die Ergebnisse zeigen, dass virtuelle Threads und reaktive Ansätze in bestimmten Szenarien eine höhere Durchsatzrate und effizientere Ressourcennutzung bieten. Diese Erkenntnisse erweitern das Verständnis der besten Praktiken für die Anwendungsentwicklung im Spring Framework und bieten wertvolle Einsichten für die Auswahl von Nebenläufigkeitsmodellen in der Java-basierten Webentwicklung.

Abstract

The objective of this work is to compare the scalability and resource utilization of reactive programming and virtual threads. To achieve this, three Spring applications were developed. Two of these applications use Spring MVC and differ in their use of native and virtual threads. The third application is based on Spring WebFlux, which implements the reactive programming paradigm. The objective of this thesis is to analyze the scalability and resource usage of various applications under different load conditions. The study found significant differences in the handling of highly scalable applications through experimental comparisons and quantitative analyses. The results indicate that virtual threads and reactive approaches offer higher throughput and more efficient resource utilization in certain scenarios. These findings expand the understanding of best practices for application development in the Spring Framework and offer valuable insights for selecting concurrency models in Java-based web development.

Inhaltsverzeichnis

1 Einleitung

In einer Welt, in der die Anforderungen an Software ständig wachsen, ist die effiziente Verarbeitung von Anfragen ein kritischer Faktor für den Erfolg von Anwendungen (vgl. Sommerville 2018, Kap. 10.1). Laut einer Umfrage von Stack Overflow aus dem Jahr 2022 hat sich das Spring Framework als eines der führenden Backend-Frameworks für Java-Anwendungen etabliert (StackOverflow 2022). Es bietet verschiedene Modelle zur Handhabung von Nebenläufigkeit. Mit der Einführung von Projekt Loom und der zunehmenden Popularität reaktiver Programmierungsparadigmen stehen Entwickler vor der Herausforderung, den richtigen Ansatz für ihre Anwendungen zu wählen. Diese Masterarbeit widmet sich der experimentellen Untersuchung der Auswirkungen von Projekt Loom und reaktiver Programmierung im Spring Framework auf die Skalierbarkeit und Ressourcennutzung. Zielsetzung ist der Vergleich der Herangehensweisen an die Nebenläufigkeit. Dabei werden folgende Forschungsfragen berücksichtigt:

1. Wie unterscheidet sich die Leistung von Spring MVC-Anwendungen hinsichtlich Antwortzeiten und Ressourcennutzung, wenn native Java-Threads im Vergleich zu virtuellen Threads des Loom-Projekts verwendet werden?
2. Inwieweit verbessert die Anwendung reaktiver Programmierungsparadigmen mit Spring WebFlux die Skalierbarkeit und den Durchsatz von Webanwendungen unter variierenden Lastbedingungen im Vergleich zu synchronen Ansätzen?
3. Welche spezifischen Szenarien und Anwendungsfälle in der Java-basierten Webentwicklung profitieren am meisten von den virtuellen Threads, die durch das Projekt Loom eingeführt wurden, und wie können diese Erkenntnisse zur Optimierung bestehender Anwendungen eingesetzt werden?

Dieses Kapitel gibt den Rahmen für die Arbeit vor, indem es die Bedeutung der Nebenläufigkeit in modernen Anwendungen und die Notwendigkeit einer effizienten Anfragebearbeitung hervorhebt. Es werden die zentralen Forschungsfragen vorgestellt und die Zielsetzung der Untersuchung skizziert.

Kapitel zwei stellt relevante Studien und Arbeiten vor, die helfen sollen, diese Arbeit in einen wissenschaftlichen Kontext einzuordnen und mögliche Forschungslücken aufzuzeigen.

Das dritte Kapitel beschäftigt sich mit der Theorie und den Grundlagen zu dem Forschungsthema. Es schafft eine Grundlage für das Verständnis der theoretischen Konzepte, die für die nachfolgende Analyse entscheidend sind. Es werden die Prinzipien der Nebenläufigkeit in der Java-Programmierung, die Architektur des Spring Frameworks sowie die Konzepte hinter Projekt Loom und reaktiver Programmierung erläutert.

Im vierten Kapitel wird die Methodik vorgestellt, welche das Design des Experiments, die Systemarchitektur der entwickelten Anwendungen sowie die spezifischen Aufgaben im Rahmen des

Experiments beschreibt. Es werden die Messinstrumente und Bewertungskriterien dargelegt, die zur Datenerhebung und Datenauswertung verwendet werden.

In den folgenden Kapitel fünf und sechs werden die Ergebnisse vorgestellt und analysiert. Sie präsentieren die gesammelten Daten im Kontext der Forschungsfragen und bieten einen direkten Vergleich der Skalierbarkeit und Ressourcennutzung der verschiedenen Nebenläufigkeitsmodelle sowie eine Diskussion der Implikationen für die Praxis.

Schließlich werden im siebten Kapitel die Erkenntnisse zusammengefasst, die Limitationen der Studie reflektiert und ein Ausblick auf zukünftige Forschungsgebiete gegeben. Das Ziel ist es, einen Beitrag zum Verständnis der besten Praktiken für die Anwendungsentwicklung im Spring Framework zu leisten und fundierte Entscheidungen über die Auswahl von Nebenläufigkeitsmodellen zu unterstützen.

Diese Arbeit hat zum Ziel, unter Verwendung einer methodischen Herangehensweise und einer detaillierten experimentellen Untersuchung das komplexe Thema der Nebenläufigkeit im Kontext moderner Java-Anwendungen objektiv zu beleuchten. Die Ergebnisse sollen Entwicklern helfen, die Leistungsfähigkeit ihrer Anwendungen zu maximieren und die Ressourcennutzung zu optimieren.

2 Verwandte Arbeiten

In diesem Kapitel werden relevante Arbeiten und Studien vorgestellt, die im Zusammenhang mit dem Thema dieser Masterarbeit stehen. Der Zweck dieses Kapitels besteht darin, bestehendes Wissen und Forschungsergebnisse zu präsentieren, die als Grundlage und Kontext für die vorliegende Arbeit dienen. Es ist wichtig zu betonen, dass die präsentierten Ergebnisse eine solide Basis für die weitere Forschung in diesem Bereich bieten und somit einen wichtigen Beitrag zur Wissenschaft leisten. Darüber hinaus sollen potenzielle Wissenslücken oder Forschungslücken identifiziert werden, die die Motivation für die vorliegende Arbeit unterstützen.

2.1 Reactive WebFlux und Spring Boot

Iwanowski und Kozieł führten eine umfassende Untersuchung durch, in der sie die imperativen und reaktiven Ansätze in der Entwicklung von Java-Webdiensten mithilfe von Spring Boot und Reactive WebFlux verglichen (Iwanowski und Kozieł 2022). Dabei analysierten sie verschiedene Aspekte der Leistung. Ein herausragendes Ergebnis dieser Studie war die Feststellung, dass der reaktive Ansatz eine signifikant niedrigere Latenz aufwies und weniger Anfragen mit langer Bearbeitungszeit hatte. Anfragen wurden schneller verarbeitet und Benutzer mussten weniger lange auf eine Antwort warten. Das ist besonders wichtig für die Skalierbarkeit und die Bereitstellung reaktionsfähiger Dienste. Trotz der Herausforderungen bei der Verwendung des reaktiven Ansatzes, erforderte dieser Ansatz in der Regel mehr Codezeilen im Vergleich zum imperativen Ansatz, was auf eine erhöhte Komplexität hinweist. Es wurde festgestellt, dass die Implementierung und das Testen des reaktiven Ansatzes mehr Zeit in Anspruch nehmen kann. Diese Erkenntnisse sind jedoch wichtig, um das Verhältnis zwischen Leistungsgewinn und Entwicklungsaufwand zu verstehen. Mit diesem Ansatz können Entwickler die Leistung ihrer Anwendungen erheblich verbessern und gleichzeitig den Entwicklungsaufwand minimieren. (vgl. Iwanowski und Kozieł 2022)

Karl Dahlin führte eine separate Fallstudie durch, um die Leistung von Reactive WebFlux speziell im Kontext von Datenbankoperationen im Vergleich zu Spring Boot MVC zu bewerten (Dahlin 2020). Die Ergebnisse waren äußerst vielversprechend und zeigen das enorme Potenzial von Reactive WebFlux auf. Die Ergebnisse dieser Studie betonen die Vorteile des reaktiven Ansatzes-Reactive WebFlux zeigte deutlich bessere Antwortzeiten. Dies deutet darauf hin, dass die Verarbeitung von Datenbankanfragen effizienter erfolgte. Dies ist besonders relevant, da Datenbankoperationen oft eine kritische Komponente in Webanwendungen darstellen. Darüber hinaus wies der reaktive Ansatz einen deutlich niedrigeren CPU-Verbrauch und eine signifikant geringere Heap-Nutzung auf. Das bedeutet, dass die Ressourcennutzung äußerst effizient war, was in skalierbaren und ressourceneffizienten Anwendungen von unschätzbarem Wert ist. (vgl. Dahlin 2020)

Zusammenfassend beleuchten die beiden Studien die Vor- und Nachteile des reaktiven Ansatzes in der Entwicklung von Java-Webdiensten mit Spring Boot und Reactive WebFlux. Die Ergebnisse

zeigen, dass der reaktive Ansatz eine gute Wahl für die Entwicklung von Webanwendungen ist. Es ist jedoch wichtig, die damit verbundenen erhöhten Entwicklungsanforderungen und Komplexität zu berücksichtigen, obwohl der reaktive Ansatz in Bezug auf Leistungsvorteile glänzt. Die Vorteile des reaktiven Ansatzes überwiegen jedoch bei weitem. Dies macht den reaktiven Ansatz zu einer geeigneten Wahl.

2.2 Leistung von Java Threads und virtuellen Threads

Pufek et al. haben eine umfassende Untersuchung durchgeführt, in der sie die Latenz pro Anfrage zwischen Java virtuellen Threads und traditionellen Java Threads verglichen haben (P. Pufek u. a. 2020). Die Studie ergab, dass die Implementierung von virtuellen Threads eine signifikant niedrigere Latenz aufwies, insbesondere bei einer Wartezeit von 10 Millisekunden pro Anfrage. Diese Erkenntnis verdeutlicht den entscheidenden Vorteil von virtuellen Threads in Situationen, in denen niedrige Latenzzeiten entscheidend sind. Virtuelle Threads sind besonders nützlich in Echtzeit-Anwendungen oder bei der Verarbeitung von Benutzeranfragen in Webdiensten. (vgl. P. Pufek u. a. 2020)

Beronić, Modrić, Mihaljević und Radovan haben in einer vergleichenden Studie die Anzahl der gestarteten Java Threads und die Menge des verwendeten Heaps in verschiedenen Prototypen analysiert. Dabei wurden normale Java- und Kotlin-Threads, Kotlin-Koroutinen und Java virtuelle Threads untersucht. Die Ergebnisse dieser Untersuchung legen nahe, dass strukturierte Nebenläufigkeitsansätze wie Kotlin-Koroutinen und Java virtuelle Threads erhebliche Leistungsverbesserungen im Vergleich zu ihren normalen Gegenstücken aufweisen. Insbesondere wurde festgestellt, dass die Anzahl der gestarteten OS-Threads bei der Verwendung von virtuellen Threads erheblich reduziert wurde. Das hat nicht nur positive Auswirkungen auf die Ressourcennutzung, sondern trägt auch zur besseren Skalierbarkeit von Anwendungen bei. Diese Erkenntnisse sind sehr vielversprechend und könnten dazu beitragen, die Entwicklung von Anwendungen zu verbessern. Darüber hinaus wurde eine geringere Heap-Nutzung beobachtet. Dies weist auf eine effizientere Verwaltung des Arbeitsspeichers hin. Diese Ergebnisse sind von großer Bedeutung, da sie auf die Potenziale von virtuellen Threads in Bezug auf Effizienz und Ressourceneinsparungen hinweisen können. (vgl. D. Beronić u. a. 2022)

Zusammenfassend zeigen diese Studien, dass virtuelle Threads im Vergleich zu traditionellen Java Threads Vorteile aufweisen, insbesondere in Bezug auf niedrige Latenzzeiten und verbesserte Ressourcennutzung. Diese Erkenntnisse sind relevant für die Entwicklung von Anwendungen, die eine hohe Leistung und Effizienz erfordern.

2.3 Forschungslücken und Motivation

Die vorliegende Masterarbeit zielt darauf ab, wichtige Forschungslücken zu schließen und fundierte Erkenntnisse in Bezug auf die Leistung von Java-basierten Webanwendungen zu liefern. Eine der zentralen Fragen, die diese Arbeit adressiert, betrifft die Leistung von Spring MVC-Anwendungen unter Verwendung von nativen Java-Threads im Vergleich zu den virtuellen Threads des Loom-Projekts. Bisherige Studien haben einige Hinweise auf die Vorteile der virtuellen Threads gegeben, aber eine umfassende Analyse ihrer tatsächlichen Auswirkungen auf Spring MVC-Anwendungen fehlt noch. Die Motivation hierbei liegt in der Notwendigkeit, die besten Ansätze für die Implementierung von Threads in Java-Webanwendungen zu verstehen, um die Leistung zu optimieren und Ressourcen effizient zu nutzen.

Eine weitere entscheidende Forschungsfrage konzentriert sich auf die Wahl zwischen virtuellen Threads und reaktiver Programmierung mit Spring WebFlux, insbesondere in Bezug auf die Verbesserung der Skalierbarkeit und des Durchsatzes von Webanwendungen unter variierenden Lastbedingungen. Obwohl beide Ansätze vielversprechend sind, fehlen umfassende Vergleichsanalysen, insbesondere unter realen Lastbedingungen. Die Motivation hinter dieser Frage besteht darin, Entwicklern und Architekten fundierte Entscheidungshilfen zur Verfügung zu stellen, um die richtige Technologie für ihre Anforderungen auszuwählen und die Leistungsfähigkeit ihrer Webanwendungen zu maximieren.

Schließlich werden spezifische Anwendungsszenarien und -fälle in der Java-basierten Webentwicklung untersucht, um herauszufinden, in welchen Kontexten die Verwendung von virtuellen Threads des Projekt Looms den größten Nutzen bringt. Diese Untersuchung soll Entwicklern und Architekten praktische Empfehlungen bieten, wie sie virtuelle Threads am effektivsten einsetzen können, um die Leistung ihrer Anwendungen zu optimieren. Die Forschungslücken in diesem Bereich bestehen in der begrenzten Verfügbarkeit von Leitlinien und bewährten Praktiken zur effektiven Nutzung dieser Technologie in der Praxis.

Insgesamt wird diese Masterarbeit dazu beitragen, das Verständnis für die Leistung von Java-basierten Webanwendungen zu vertiefen und konkrete Erkenntnisse für die Entwicklung und Optimierung von Anwendungen in diesem Bereich zu liefern, einschließlich der Wahl zwischen virtuellen Threads und reaktiver Programmierung.

3 Theorie und Hintergrund

Das vorliegende Kapitel bildet die Grundlage für ein umfassendes Verständnis der theoretischen Grundlagen, die im weiteren Verlauf dieser Masterarbeit von entscheidender Bedeutung sind. Die Arbeit konzentriert sich auf die Thematik der Nebenläufigkeit im Kontext des Spring Frameworks, einem modernen Java Backend Framework.

Der Kern dieses Kapitels befasst sich zuerst mit der Nebenläufigkeit in der Java-Programmierung. Dabei wird eine Basis gelegt, indem die Grundprinzipien der Nebenläufigkeit und in diesem Kontext die Besonderheiten der Programmiersprache Java erläutert werden. Insbesondere wird auf das Thread-Per-Request-Modell, seine Definition sowie seine Vor- und Nachteile eingegangen und die damit verbundenen Herausforderungen erläutert.

Im nächsten Abschnitt wird als Lösungsansatz für die Herausforderungen des Thread-Per-Request-Modells die reaktive Programmierung betrachtet. Dabei wird das Konzept, die dahinterstehende Idee sowie die Implementierung von reaktiven Frameworks ausführlich erläutert. Des Weiteren werden die Herausforderungen diskutiert, denen Entwickler bei der praktischen Anwendung dieses Paradigmas gegenüberstehen.

Um eine Verbindung zwischen traditioneller Thread-Programmierung und modernen Anwendungen herzustellen, wird auch das Projekt Loom von Oracle vorgestellt. Oracle beschreibt dieses Projekt als bahnbrechende Initiative zur Einführung von virtuellen Threads in die Java-Plattform. Virtuelle Threads versprechen eine erhebliche Vereinfachung bei der Entwicklung, Wartung und Überwachung von hochdurchsatzfähigen, nebenläufigen Anwendungen. Die technische Funktionsweise virtueller Threads wird im Detail besprochen und deren Implementierung in der Arbeit vorgestellt.

Darüber hinaus wird das Spring Framework als wichtiger Bestandteil moderner Java-Anwendungen behandelt. Sowohl das etablierte Spring Web MVC Framework, das dem Threads-per-Request Ansatz folgt, als auch das innovative Spring Webflux, das reaktive Programmierung als Lösung für Skalierungs- und Ressourcenprobleme in den Vordergrund stellt, werden vorgestellt.

Dieses Kapitel bietet eine theoretische Grundlage, auf der die praktischen Erkenntnisse und Anwendungen in den folgenden Kapiteln aufbauen können.

3.1 Nebenläufigkeit

Nebenläufigkeit ist ein grundlegendes Konzept in der Softwareentwicklung, dass es einem System ermöglicht, mehrere Aufgaben oder Operationen gleichzeitig auszuführen, oft auf verschiedenen Prozessoren oder Kernen. Es ermöglicht verschiedenen Teilen oder Einheiten eines Programms, eines Algorithmus oder Problems, sich in einer teilweisen oder nicht festgelegten Reihenfolge auszuführen, ohne das Endergebnis zu beeinflussen. (vgl. Oracle 2022) Das primäre Ziel der Nebenläufigkeit ist es, die Leistung und Effizienz von Softwareanwendungen durch eine optimale Auslastung der zur Verfügung stehenden Systemressourcen zu steigern. Dieses Konzept, das sich von der Betriebssystemebene bis zur Anwendungsebene erstreckt, ist entscheidend für die effiziente Nutzung moderner Hardwaresystemen.

3.1.1 Prozesse und Threads

In einem Betriebssystem wird die Ausführung von Programmen durch Prozesse und Threads verwaltet. Ein Prozess ist eine Instanz eines laufenden Programms, das über einen eigenen Adressraum und Systemressourcen wie geöffnete Dateien oder Netzwerkverbindungen verfügt, wie aus Abbildung 1 hervorgeht. (vgl. Tanenbaum und Bos 2015, Kap. 2.1)

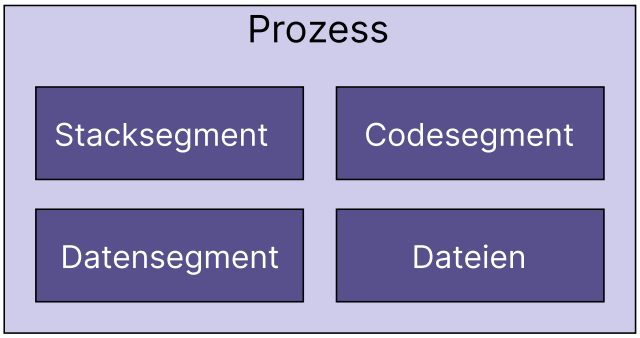

Abbildung 1 Aufbau eines Prozesses (vgl. Wolf und Krooß 2023, Abb. 26.3)

Prozesswechsel werden oft als aufwendige Operationen angesehen, da ein Wechsel zwischen den Prozessen eine Schnittstelle im Betriebssystem erfordert und einen Unterbrechungsaufruf an den Kernel auslöst (vgl. Silberschatz, Galvin, und Gagne 2018, Kap. 3).

Ein Thread ist gemäß Abbildung 2 eine leichtgewichtige Ausführungseinheit innerhalb eines Prozesses. Threads teilen sich den Adressraum sowie andere Ressourcen des übergeordneten Prozesses wie geöffnete Dateien und Netzwerkverbindungen. Im Unterschied zu Prozessen, die isoliert sind, teilen Threads den Speicher, was ihre Erstellung und Verwaltung effizienter macht. Threads werden auch als Leichtgewichtsprozesse bezeichnet, da sie einige Eigenschaften von Prozessen besitzen, jedoch mit weniger Overhead verbunden sind. Jeder Thread gehört genau einem Prozess an und ein Prozess kann aus vielen Threads bestehen, wenn das Betriebssystem Multithreading unterstützt. Ein Thread ist eine einzelne, sequenzielle Ausführung innerhalb eines Prozesses und die kleinste Abfolge von programmierten Anweisungen, die unabhängig von einem Scheduler, der typischerweise ein Teil des Betriebssystems ist, verwaltet werden kann. (vgl. Silberschatz, Galvin, und Gagne 2018, Kap. 4; Tanenbaum und Bos 2015, Kap. 2.2)

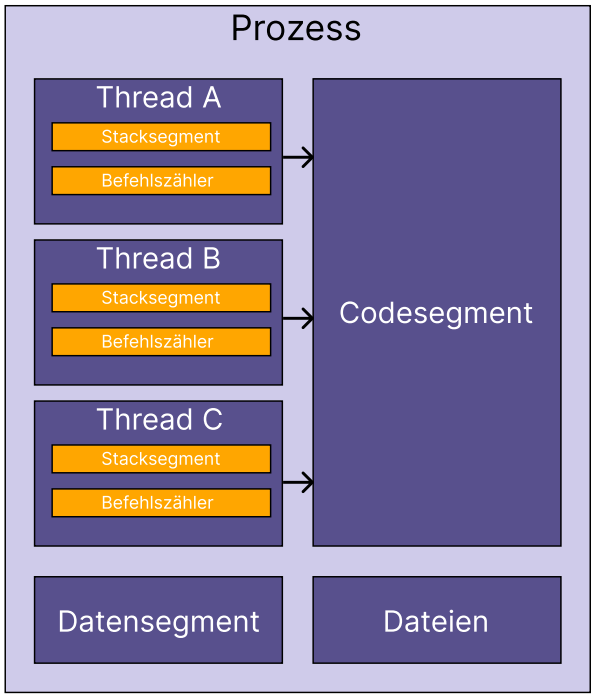

Abbildung 2 Threads in Prozessen (vgl. Wolf und Krooß 2023, Abb. 26.4)

Im Betriebssystem ist ein Thread die grundlegende Einheit, der Prozessorzeit zugewiesen wird. Threads ermöglichen die gleichzeitige Ausführung von Aufgaben innerhalb eines Prozesses, was die Effizienz und Reaktionsfähigkeit der Anwendung verbessern. Wenn beispielsweise eine Anwendung auf einem Computer gestartet wird, erstellt das Betriebssystem einen Prozess dafür, und dieser Prozess kann einen oder mehrere Threads enthalten, die verschiedene Aufgaben ausführen (vgl. Tanenbaum und Bos 2015, 2.2).

3.1.2 Nebenläufigkeit auf Systemebene

Auf Systemebene ermöglicht die Nebenläufigkeit die gleichzeitige Ausführung mehrerer Prozesse durch das Betriebssystem, ein Konzept, das auch als Multitasking bekannt ist (vgl. Silberschatz, Galvin, und Gagne 2018, Kap. 1.4.1). Im Rahmen des Multitaskings werden die Systemressourcen, wie CPU, Speicher und Ein-/Ausgabegeräte, zwischen den Prozessen aufgeteilt, um eine gleichzeitige Abarbeitung mehrerer Aufgaben zu ermöglichen.

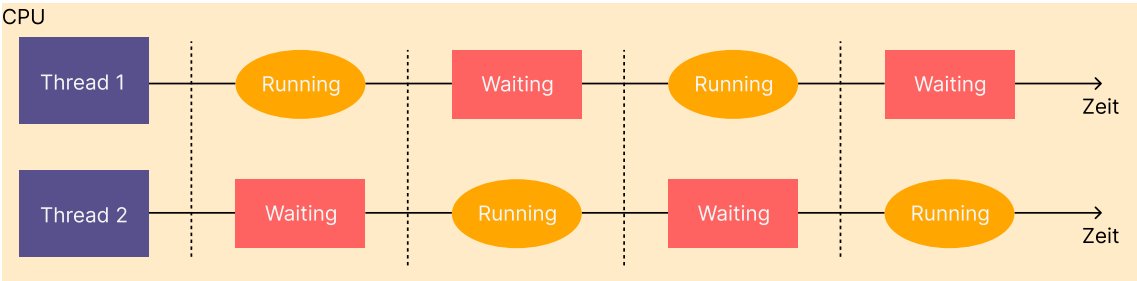

Abbildung 3 Scheduling von zwei Threads

Wie in Abbildung 3 dargestellt, ist das Scheduling (Planung) ein Mechanismus, der die Zuweisung von CPU-Zeit für jeden Prozess und Thread verwaltet, um sicherzustellen, dass jeder Prozess oder

Thread ausreichend und gleichmäßig Zeit erhält (vgl. Silberschatz, Galvin, und Gagne 2018, 163). In einem Einprozessorsystem wechselt die CPU schnell zwischen verschiedenen Threads, was den Eindruck erweckt, dass sie parallel ausgeführt werden. Der Dispatcher, ein Teil des Betriebssystems, ermöglicht diesen schnellen Wechsel zwischen Threads (vgl. Weinländer 1995, Kap. 3.2.3). Es wird jeweils nur ein Thread zur Ausführung gebracht, jedoch geschieht der Wechsel so schnell, dass der Eindruck entsteht, als würden die Threads gleichzeitig ausgeführt.

Es gibt verschiedene Algorithmen zur Zeitplanung, um die CPU-Zeit effizient zu verwalten. Zwei gängige Ansätze sind Round-Robin und First-Come-First-Serve. Beim Round-Robin-Verfahren werden jedem Prozess in zyklischer Reihenfolge feste Zeitscheiben ohne Berücksichtigung von Prioritäten zugewiesen. Dies ist ein einfacher und gerechter Ansatz. Im Gegensatz dazu werden bei First-Come-First-Serve Anfragen und Prozesse in der Reihenfolge ihrer Ankunft bearbeitet und durch eine FIFO-Warteschlange gekennzeichnet. (vgl. Baun 2017, Kap. 8.6)

Betrachtet man den historischen Kontext des Linux-Schedulers, fällt auf, dass sich die Scheduler im Laufe der Zeit stark weiterentwickelt haben. Die erste Version des Linux-Schedulers (v0.01) war sehr einfach und umfasste nur 20 Zeilen Code. In dieser Version wurden alle Aufgaben in einem Array ausgedrückt, das sowohl als Aufgabenliste als auch als Laufzeitwarteschlange diente. Die Zeitscheibe des Schedulers betrug 150 Millisekunden. In späteren Versionen des Linux-Schedulers wurden erhebliche Änderungen vorgenommen. Unter anderem wurde der O(1) Scheduler in den Versionen v2.6.0 bis v2.6.22 eingeführt und ab Version v2.6.23 wurde der Completely Fair Scheduler (CFS) implementiert. Diese Entwicklungen spiegeln die fortschreitende Optimierung und Anpassung des Linux-Schedulers an die Anforderungen moderner Rechensysteme wider. (vgl. Takeuchi 2019)

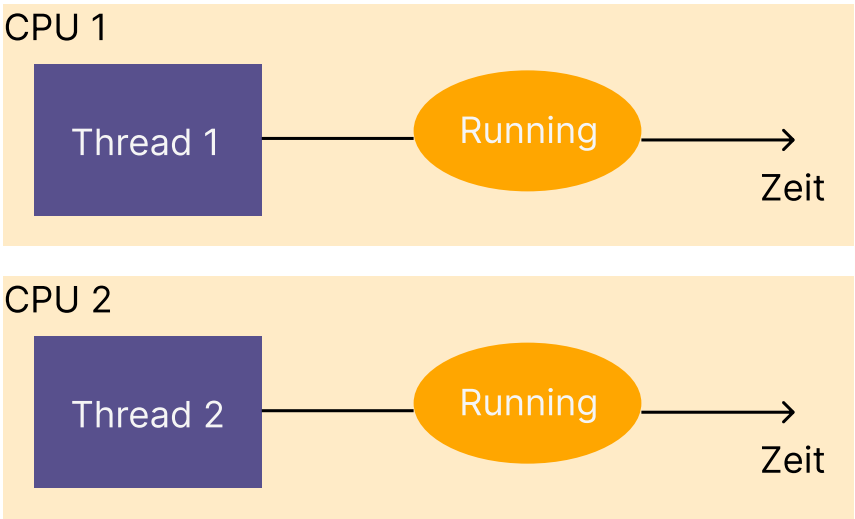

Abbildung 4 Parallelität von zwei Threads

Wie in Abbildung 4 veranschaulicht, tritt Parallelität auf, wenn mehrere Prozessoren oder Prozessorkerne vorhanden sind, welche die tatsächlich gleichzeitige Ausführung der Prozesse oder Threads ermöglichen. Im Gegensatz zum Scheduling, bei dem die Aufgaben scheinbar gleichzeitig auf einem einzelnen Prozessorkern ausgeführt werden, ermöglicht die Parallelität die gleichzeitige Ausführung der Aufgaben auf verschiedenen Kernen (vgl. Silberschatz, Galvin, und Gagne 2018,

163). Dies steigert die Gesamtleistung des Systems, da mehrere Aufgaben gleichzeitig ausgeführt werden können, ohne auf die Verfügbarkeit des Prozessors warten zu müssen.

Der wesentliche Unterschied zwischen Scheduling und Parallelität besteht darin, wie Threads auf den Prozessorkernen ausgeführt werden. Beim Scheduling werden die Threads sequenziell auf einem einzelnen Kern ausgeführt, doch in einer Weise, die den Anschein erweckt, dass sie gleichzeitig ablaufen. Hingegen ermöglicht die Parallelität die tatsächliche gleichzeitige Ausführung von Threads auf verschiedenen Kernen. Für das Scheduling ist lediglich ein einzelner Prozessorkern auf einem System ausreichend, um eine Funktion zu gewährleisten. Parallelität hingegen erfordert mehrere Kerne, um effektiv arbeiten zu können. Ein optimaler Ansatz für die Nebenläufigkeit besteht darin, Threads auf einem System mit mehreren Prozessorkernen parallel zu verteilen und auf den verwendeten Prozessorkernen das Scheduling betreiben.

3.1.3 Nebenläufigkeit auf Anwendungsebene

Auf Anwendungsebene ermöglicht die Nebenläufigkeit hypothetisch die gleichzeitige Ausführung von Aufgaben innerhalb einer Anwendung durch die Verwendung von Threads. Ein Beispiel für die Nebenläufigkeit auf Anwendungsebene ist eine Textverarbeitungssoftware wie Microsoft Word. In einem solchen Programm könnten beispielsweise drei separate Threads gleichzeitig ausgeführt werden.

Ein Thread wäre zuständig für die Entgegennahme und Verarbeitung von Eingaben des Benutzers. Der zweite Thread könnte im Hintergrund laufen, um die Textanzeige auf dem Bildschirm automatisch zu aktualisieren, wobei der dritte Thread eine automatische Rechtschreib- und Grammatikprüfung durchführen könnte.

Durch die gleichzeitige Ausführung dieser Threads kann das Programm eine reibungslose und ansprechende Benutzererfahrung bieten. Der Benutzer kann weiter tippen, während das Programm den Text aktualisiert und gleichzeitig die Rechtschreibung und Grammatik überprüft. Diese Form der Nebenläufigkeit auf Anwendungsebene ermöglicht es Entwicklern, effizientere und reaktionsfähigere Softwareanwendungen zu erstellen, die mehrere Aufgaben gleichzeitig ausführen können, ohne die Benutzerfreundlichkeit zu beeinträchtigen.

3.1.4 Zusammenfassung

In dem Abschnitt wurde das zentrale Konzept der Nebenläufigkeit in der Softwareentwicklung behandelt. Durch die nebenläufige Ausführung von Prozessen und Threads auf Betriebssystem- und Anwendungsebene ermöglicht die Nebenläufigkeit eine effiziente Nutzung der vorhandenen Hardware-Ressourcen. Insbesondere bei modernen Multicore-Systemen ist die effektive Verwaltung und Ausführung von Threads entscheidend, um die Performance und Reaktionsgeschwindigkeit von Softwareanwendungen zu optimieren. Es wurden verschiedene Mechanismen der Nebenläufigkeit wie Scheduling und Parallelität diskutiert, um ein tieferes Verständnis der zugrundeliegenden Herausforderungen zu ermöglichen.

Im nachfolgenden Kapitel 2.2 wird präziser auf die Umsetzung von Nebenläufigkeit in der Programmiersprache Java eingegangen. Java bietet dank der eingebauten Multithreading-Unterstützung eine stabile Plattform zur Entwicklung von nebenläufigen Anwendungen.

3.2 Nebenläufigkeit in Java

Die Nebenläufigkeit in Java erlaubt die gleichzeitige Ausführung von mehreren Programmteilen. Durch die Verwendung von Java können Anwendungen eine hohe Leistung und Durchsatz erreichen. Die Programmiersprache Java ist dabei besonders auf die Thread-Programmierung ausgerichtet, was sich unter anderem in der Verwendung der Klasse Thread und der Runnable-Schnittstelle aus dem Paket java.lang zeigt.

3.2.1 Java-Threads und Runnable Interface

Java bietet eine integrierte Unterstützung für Multithreading. Die Grundlage für die Thread-Programmierung in Java bilden die Thread-Klasse und die Runnable-Schnittstelle aus dem Paket java.lang (Oracle 2023a). Mit diesen Elementen können Threads in Java-Anwendungen erstellt und verwaltet werden. Wie in Listing 1 dargestellt, gibt es die Möglichkeit, in Java einen Thread zu erzeugen, indem man eine Klasse von der Thread-Klasse ableitet und die Methode run überschreibt. In diesem Fall enthält die Methode run den Code, der vom Thread ausgeführt werden soll.

```
public class MyThread extends Thread {
    @Override
    public void run() {
        // Code der durch den Thread ausgeführt werden soll
    }

    public static void main(String[] args) {
        MyThread thread = new MyThread();
        thread.start();
    }
}
```

Listing 1 Erstellung und Ausführung eines Java-Threads

Eine alternative und flexiblere Methode zur Erstellung eines Threads besteht darin, eine Klasse zu erstellen, die die Runnable-Schnittstelle implementiert. Ein Runnable-Objekt kann einem Thread zur Ausführung übergeben werden. Die Vorteile dieses Ansatzes sind, dass eine Klasse die Runnable-Schnittstelle sowie weitere Schnittstellen implementieren kann und gleichzeitig eine andere Klasse erweitern kann. Dies ist bei der Vererbung von Thread aufgrund der fehlenden Unterstützung von Mehrfachvererbung in Java nicht möglich. Zudem ermöglicht es eine klare Trennung zwischen der Ausführungslogik des Threads und der Anwendungslogik, was den Code verständlicher und wartbarer macht. (vgl. Ullenboom 2014, Kap. 14.2)

Listing 2 zeigt, wie Threads in Java ausgeführt werden. Hierfür wird die Methode start verwendet, welche die run-Methode des Threads oder des Runnable-Objekts aufruft, in dem der auszuführende Code definiert ist. Es ist wichtig, die start-Methode zu verwenden, da ein direkter Aufruf der run-Methode ohne die start-Methode dazu führt, dass der Code im aktuellen Thread und nicht in einem neuen Thread ausgeführt wird. (vgl. Gravelle 2023).

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // Code der durch den Thread ausgeführt werden soll
    }

    public static void main(String[] args) {
        Thread thread = new Thread(new MyRunnable());
        thread.start();
    }
}
```

Listing 2 Erstellung und Ausführung einer Runnable-Klasse

3.2.2 Schnittstelle zwischen Java und Betriebssystem

Java-Threads werden oft auch als native Threads bezeichnet. Diese dienen als Abstraktionsebene, um die Ausführung von Java-Code auf Betriebssystem-Threads zu ermöglichen. Im Kern ist ein nativer Thread eine Instanz der Klasse java.lang.Thread, die traditionell als dünne Abstraktion um einen Betriebssystem-Thread herum implementiert ist. Durch diese Abstraktion wird eine feste Verankerung zwischen dem Anwendungsebene und Betriebssystemebene geschaffen. (vgl. Ullenboom 2014, Kap. 14.1.2)

Ursprünglich nutzte Java zwischen 1997 und 2000 Green Threads, eine Implementierung von Benutzer-Threads, die unabhängig von den nativen Threads sind. Dabei fand eine N:1-Zuordnung zwischen Green Threads und einem einzigen nativen Thread statt. Viele Green Threads werden von einem einzigen Betriebssystem-Thread verwaltet. Die Java Virtual Machine (JVM) ist somit für das Scheduling und die Verwaltung dieser Threads verantwortlich. Hierbei wird auch von Benutzer-Modus-Scheduling gesprochen, da die Green Threads von der Anwendung und nicht vom Betriebssystem eingeplant werden (vgl. Weinländer 1995, Kap. 3.2). Green Threads waren sehr leicht zu implementieren. Ein großes Problem von ihnen bestand aber darin, dass sie im Gegensatz zu Betriebssystem-Threads keine Leistungsverbesserungen durch Parallelität erreichen konnten, da nur ein Betriebssystem-Thread verwendet wird.

Wie in der Abbildung 5 veranschaulicht, wechselte Java mit der Veröffentlichung von Java 1.2 im Jahr 1998 Java zur Verwendung von nativen Threads. Die Architektur nativer Threads sieht üblicherweise eine direkte 1:1-Zuordnung zu den Kernel-Threads des Betriebssystems vor. Dadurch kann das Betriebssystem die Ausführung und Verwaltung dieser Threads steuern. Jeder native Thread wird von einem Kernel-Thread des Betriebssystems repräsentiert und nach den Scheduling-Richtlinien des Betriebssystems geplant und ausgeführt. Mit dieser Änderung ist Java von der Verwendung eines Benutzer-Modus-Schedulers zur Verwendung eines Betriebssystem-Schedulers übergegangen. (vgl. Ullenboom 2014, Kap. 16)

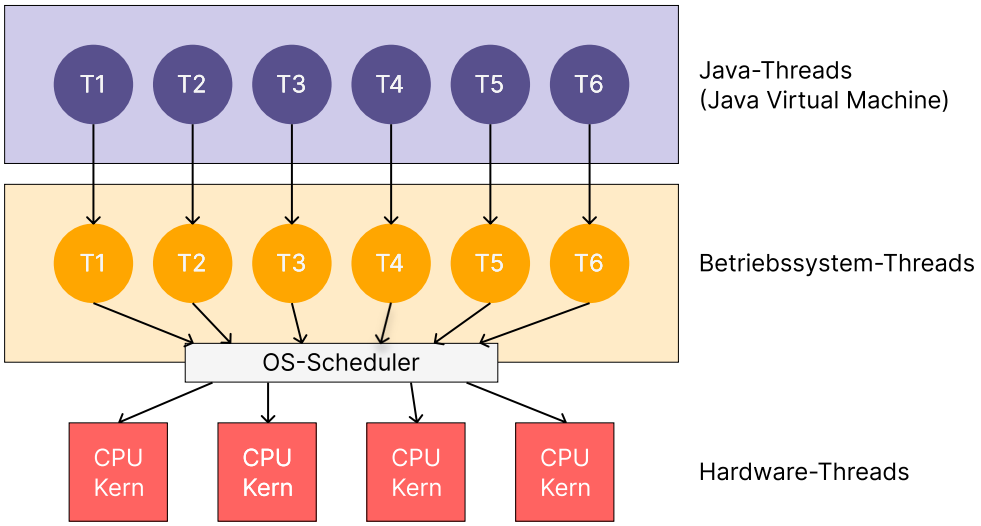

Abbildung 5 Zuweisung von Java-Threads zu OS-Threads

Ein grundlegender Aspekt der Funktionalität von nativen Threads ist die Ressourcenverwaltung. Jeder nativer Thread besitzt einen eigenen Stapelspeicher sowie weitere Ressourcen, die vom Betriebssystem verwaltet werden. Diese Ressourcen sind erforderlich, um den Ausführungszustand des Threads zu speichern und somit die Ausführung von Java-Code in diesem Thread zu ermöglichen. Die Erstellung und Verwaltung dieser Ressourcen sind jedoch mit einem gewissen Aufwand verbunden, wodurch die Erstellung neuer nativer Threads zu einer ressourcenintensiven Operation wird. (vgl. Gupta 2022)

Native Threads bilden die Basis für die Multithreading-Programmierung in Java. Durch ihre enge Integration in die Java-Laufzeitumgebung und das Betriebssystem bieten sie eine robustere und leistungsstärkere Plattform für die Ausführung von Java-Code in einer Multithreading-Umgebung als Green Threads.

3.2.3 Herausforderungen von nativen Threads

Die Implementierung und Verwaltung von nativen Threads in Java werfen jedoch in vielen Anwendungen eine Reihe von Herausforderungen und Problemen auf. Eines der zentralen Probleme ist die Skalierbarkeit, welche durch die direkte Abbildung von Java-Threads auf Betriebssystem-Threads beeinträchtigt wird, wie das Paper zum Thema Multithreading in Java: Performance and Scalability on Multicore Systems zeigt (Chen, Chang, und Hou 2010).

Damit einher geht eine Begrenzung der möglichen Anzahl von nativen Threads. Aufgrund der Ressourcenanforderungen, die mit der Erstellung und Verwaltung von Threads verbunden sind, ist es nicht möglich, eine große Anzahl von Threads zu erstellen. Dies ist darauf zurückzuführen, dass jeder dieser Threads mehr als 1 Megabyte Speicher benötigt (vgl. Sadakath 2022). Für jeden Thread müssen Ressourcen wie ein eigener Stack und ein eigener Heap-Speicher bereitgestellt werden. Dies stellt eine erhebliche Belastung für das System dar und kann die Leistung beeinträchtigen. (vgl. Oracle 2023b, Abschn. 14)

Beispiele für Nebenläufigkeitsprobleme, die beim Umgang mit nativen Threads auftreten können, sind das Producer-Consumer-Problem oder Race Conditions. Das Producer-Consumer-Problem,

auch als Problem des begrenzten Puffers bekannt, wurde seit 1965 von Edsger W. Dijkstra beschrieben (Dijkstra 1965). Dabei spielen zwei Prozesse zusammen: Der Produzent und der Konsument. Der Produzent erzeugt Daten, die vom Konsumenten verarbeitet werden. Die Herausforderung besteht darin, Datenproduktion und -verarbeitung zu synchronisieren, insbesondere wenn sie in unterschiedlichen Raten erfolgen. Im ursprünglichen Beispiel von Dijkstra wurden zwei Semaphoren verwendet, um die Synchronisation zu steuern: Eine für die Länge der Warteschlange und eine für die Anzahl der unbestätigten Operationen. Diese Synchronisation ist entscheidend, um zu verhindern, dass der Konsument auf Daten zugreift, die noch nicht erzeugt wurden, oder dass der Produzent Daten erzeugt, für die im Puffer kein Platz mehr ist, damit keine Pufferelemente überschrieben werden. (vgl. Tanenbaum und Bos 2015, 128) Race Conditions entstehen, wenn zwei oder mehr Prozesse gleichzeitig auf dieselbe Ressource zugreifen und mindestens einer der Prozesse diese Ressource verändert. Dies kann zu unvorhersehbaren Ergebnissen führen, wenn die Operationen in einer anderen als der vorgesehenen Reihenfolge ausgeführt werden. Ein alltägliches Beispiel ist ein Lichtschalter in einem Haus, bei dem mehrere Schalter dieselbe Deckenlampe steuern. Wenn zwei Personen das Licht gleichzeitig an verschiedenen Schaltern ein- oder ausschalten, kann dies zu einem unerwarteten Ergebnis führen. In der IT kann eine Race Condition auftreten, wenn z.B. zwei Prozesse fast gleichzeitig Lese- und Schreibbefehle auf denselben Speicherplatz ausführen, was zu Fehlern wie Datenkorruption, Systemabstürzen oder unerwartetem Programmverhalten führen kann. (vgl. Tanenbaum und Bos 2015, 119)

Diese Probleme können die Komplexität der Thread-Verwaltung erhöhen und das Risiko von schwer identifizierbaren und behebbaren Fehlern erhöhen. Die Nutzung der Klassen aus dem Paket java.util.concurrent können dazu beitragen, diese Probleme zu umgehen oder zu reduzieren, jedoch bleibt die grundlegende Komplexität bestehen (vgl. Oracle 2023a).

Ein weiteres Problem kann entstehen, wenn ein Thread auf eine Ressource wartet, die von einem anderen Thread gesperrt ist und dadurch blockiert wird. Ein häufiges Szenario ist der Zugriff auf eine Datei oder eine andere Ressource, die bereits von einem anderen Thread verwendet wird. In solchen Fällen wird der betreffende Thread blockiert und muss warten, bis die Ressource freigegeben wird. Dies führt zu schlechter Auslastung der Systemressourcen und kann die Leistung der Anwendung beeinträchtigen.

3.2.4 Thread-Per-Request Modell

Das Thread-Per-Request Modell ist ein weit verbreitetes Konzept in vielen serverseitigen Anwendungen, insbesondere in Java-basierten Webanwendungen und Frameworks. Im Kern besteht dieses Modell darin, jeder eintreffenden HTTP-Anfrage einen eigenen Thread zuzuweisen. Dies ermöglicht eine klare Trennung der Bearbeitung von unterschiedlichen Anfragen und eine einfache Programmierung, da Entwickler sich nicht um die Koordination nebenläufiger Prozesse kümmern müssen. (vgl. Schmidt und Vinoski 1995, Abschn. 1.2)

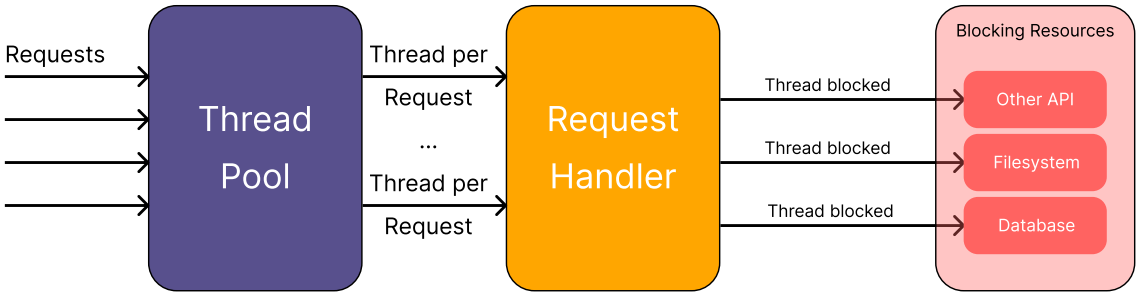

Abbildung 6 Beispiel für das Thread-Per-Request Modell

Wie die Abbildung 6 zeigt, wird in einem typischen Szenario des Thread-Per-Request Modells ein Pool von Threads erstellt, aus dem ein Thread für die Bearbeitung jeder neuen Anfrage entnommen wird. Nach Abschluss der Anfrage wird der Thread zurück in den Pool gelegt, um später erneut verwendet zu werden. Dieses Modell ist in vielen traditionellen Java-Frameworks wie Spring MVC implementiert, da es eine einfache und direkte Methode zur Behandlung von Client-Anfragen bietet (vgl. Chandrakant 2020a).

Das Modell hat jedoch auch einige Nachteile, da es bisher auf die Verwendung von nativen Threads angewiesen ist. Einer der größten Nachteile ist die Skalierbarkeit. Wie bereits erwähnt, verbraucht jeder Thread Systemressourcen wie CPU-Zeit und Speicher, wodurch die Erstellung eines neuen Threads für jede Anfrage schnell zu einem Engpass führen kann. Dies ist insbesondere in Hochlastszenarien mit vielen gleichzeitigen Anfragen problematisch. Dieses Modell hat die Tendenz, einen Großteil der Systemressourcen durch inaktive Threads zu binden, welche auf Ein- und Ausgabeoperationen warten. Diese Art der Überlastung kann die Systemleistung erheblich beeinträchtigen. (vgl. Carter 2022)

3.2.5 Zusammenfassung

Im diesem Kapitel wurden die Implementierung und Handhabung von Threads in der Java-Programmierung sowie die Schnittstelle zwischen Java und dem Betriebssystem durch native Threads erörtert. Es wurde herausgestellt, dass trotz der robusten Möglichkeiten zur Realisierung von Nebenläufigkeit mittels nativen Threads, auch erhebliche Herausforderungen und Einschränkungen existieren. Insbesondere im Kontext moderner serverseitiger Anwendungen und Frameworks wie beispielsweise Spring Web MVC stößt das traditionelle Thread-Per-Request-Modell schnell an seine Grenzen, obwohl es auf den ersten Blick als intuitiver Ansatz erscheint.

Eine mögliche Strategie zur Umgehung dieser Herausforderungen könnte die Bereitstellung zusätzlicher Hardware-Ressourcen zur Lösung der Skalierungsprobleme sein. Allerdings ist dieser Ansatz nicht nachhaltig und kann schnell kostspielig werden. Eine rein hardwarebasierte Skalierungslösung ist oft nicht in der Lage, die Skalierungsprobleme vollständig zu lösen und verschiebt lediglich die Grenzen, bis zu denen das System performant arbeiten kann.

Ein vielversprechender und alternativer Ansatz zur Bewältigung der Herausforderungen der Nebenläufigkeit in Java ist die reaktive Programmierung. Dieses asynchrone Programmierparadigma ermöglicht eine effiziente und skalierbare Handhabung von Nebenläufigkeit, insbesondere in Sze-

narien mit hoher Konkurrenz und vielen gleichzeitigen Anforderungen. Die reaktive Programmierung verfolgt das Ziel, die Systemressourcen effizienter zu nutzen und eine bessere Kontrolle über die Nebenläufigkeit zu erlangen, indem sie ein ereignisgetriebenes Modell und nicht-blockierende I/O-Operationen einsetzt.

Im nächsten Kapitel wird das Paradigma der reaktiven Programmierung im Detail vorgestellt und diskutiert, wie es die Nebenläufigkeit in Java-Webanwendungen verbessert und welche Herausforderungen es mit sich bringt.

3.3 Reaktive Programmierung

Reaktive Programmierung ist ein Programmierparadigma. Es befasst sich mit der Verarbeitung von Datenströmen und der Propagierung von Änderungen in diesen Datenströmen. Dieses Paradigma ermöglicht durch deklarativen Code die Konstruktion von asynchronen Verarbeitungspipelines. Der Kerngedanke liegt im Erstellen von Anwendungen, die eine reaktionsfähige, belastbare, elastische und nachrichtengetriebene Architektur aufweisen. (vgl. Royal 2023, Kap. 14) Reaktive Programmierung basiert auf der Idee der Ereignisverarbeitung in asynchronen Datenströme und wird heute in verschiedenen Bereichen wie GUI-Programmierung, Web-Programmierung, Mikroservices oder allgemein in reaktiven Systemen eingesetzt (vgl. Otta 2023). Diese Art der Programmierung ermöglicht es, dass Änderungen in einem System effizient und in einer gut definierten Weise behandelt werden können, wodurch sie insbesondere für Umgebungen mit hohen Anforderungen an Reaktionsfähigkeit und Skalierbarkeit geeignet ist.

Der Ursprung der reaktiven Programmierung geht auf die Veröffentlichung des Reactive Manifesto im Juli 2013 zurück (Bonér u. a. 2014). Dieses Manifest, verfasst von Jonas Bonér, Roland Kuhn, Martin Thompson und Dave Farley, legte die Grundprinzipien der reaktiven Programmierung fest. Dabei lag der Fokus auf Aspekten wie Reaktionsfähigkeit, Belastbarkeit, Elastizität und nachrichtenbasierten Architekturen und betonte die Bedeutung reaktiver Systeme in der modernen Softwareentwicklung. Im September 2014 wurde eine überarbeitete Version des Manifests veröffentlicht, die einige der ursprünglichen Konzepte weiter präzisierte und ergänzte. (vgl. Bonér u. a. 2014; vgl. Royal 2023)

In der reaktiven Programmierung steht der Datenfluss im Fokus. Wenn sich etwas verändert, aktualisiert sich der Datenfluss selbst. Ein asynchroner Datenstrom ist somit eine zeitlich geordnete Abfolge von Ereignissen (Zustandsänderungen), die drei verschiedene Ausgabemöglichkeiten besitzt: einen Wert (jeden Typs), einen Fehler oder ein Signal, welches den Abschluss des Datenstroms signalisiert. Diese Ereignisse werden asynchron erfasst, indem Funktionen definiert werden, die ausgeführt werden, wenn ein Wert, ein Fehler oder ein Signal, welches das Ende des Datenstroms signalisiert, ausgegeben wird. (vgl. Jansen und Jiang 2020; vgl. Royal 2023, Kap. 3)

Die reaktive Programmierung sollte nicht als Ersatz, sondern als Ergänzung zur threadbasierten Nebenläufigkeit betrachtet werden. Sie eignet sich insbesondere, wenn eine threadbasierte Nebenläufigkeit erforderlich ist, um die Ressourcennutzung zu verbessern. Durch die Umstellung des Programmflusses von einer synchronen Abfolge von Aufgaben auf einen asynchronen Strom von Ereignissen wird der Fokus auf Datenströme und deren Änderungsübertragung gelegt. Diese Vorgehensweise basiert auf asynchronem und nicht-blockierendem Code, um Datenströme effizient und skalierbar zu handhaben. Die Idee ist, auf eingehende Ereignisse zu reagieren, anstatt auf blockierende Zustände zu warten. (vgl. Royal 2023, Kap. 3)

3.3.1 Grundprinzipien des Reactive Manifesto

Wie bereits erwähnt, basiert die Philosophie der reaktiven Programmierung auf dem Reactive Manifesto. Die grundlegende Prinzipien des Manifests zielen darauf ab, Anwendungen zu entwickeln, die reaktionsschnell, belastbar und effizient skalierbar sind. Sie adressieren die Herausforderungen, die sich aus der asynchronen Verarbeitung von Ereignissen und Datenströmen ergeben, und fördern eine hohe Leistung und Zuverlässigkeit in verschiedenen Anwendungsbereichen. Im Folgenden werden die vier Grundprinzipien des Reactive Manifesto aufgeführt, welche im Reactive Manifesto definiert werden (Bonér u. a. 2014).

Reaktive Systeme müssen in der Lage sein, zeitnah auf Ereignisse zu reagieren. Eine schnelle und zuverlässige Reaktion verbessert die Benutzererfahrung und gewährleistet die zuverlässige Funktion des Systems (vgl. Bonér u. a. 2014).

Die Belastbarkeit eines Systems bezieht sich darauf, dass es auch im Falle von Fehlern und Ausfällen funktionsfähig bleibt. Reaktive Systeme sollten in der Lage sein, Fehler zu isolieren und zu behandeln, um kontinuierliche Funktionalität zu gewährleisten (vgl. Bonér u. a. 2014).

Elastische Systeme können sich an veränderte Lastbedingungen anpassen, indem sie Ressourcen dynamisch zuweisen und freigeben. Dies ermöglicht, dass reaktive Systeme effizient skaliert werden können und Ressourcen optimal genutzt werden (vgl. Bonér u. a. 2014).

Reaktive Systeme nutzen eine nachrichtenbasierte Architektur, um die Kommunikation zwischen den verschiedenen Systemkomponenten zu ermöglichen. Hierdurch wird eine Entkopplung gefördert, welche eine effiziente asynchrone Kommunikation ermöglicht (vgl. Bonér u. a. 2014).

3.3.2 Reaktive Stream Spezifikation

Die reaktive Stream Spezifikation ist ein Standard für asynchrone Stream-Verarbeitung mit nichtblockierender Rückkopplung. Diese Initiative zielt darauf ab, einen Standard für Laufzeitumgebungen wie die JVM und JavaScript sowie für Netzwerkprotokolle bereitzustellen (vgl. Christensen und Kuhn 2015). Dabei besteht die Spezifikation aus verschiedenen Teilen. Die API definiert die Typen, die implementiert werden müssen, um reaktive Streams zu implementieren und die Interoperabilität zwischen verschiedenen Implementierungen zu gewährleisten. Das Technology Compatibility Kit (TCK) ist eine standardisierte Testsuite für die Konformitätsprüfung von Implementierungen. Die Spezifikation wurde im Reactive Manifesto definiert und es gibt verschiedene Implementierungen, wie zum Beispiel RxJava oder Akka-Streams. Den reaktiven Erweiterungen steht es frei, zusätzliche Funktionen zu implementieren, die nicht durch die Spezifikation abgedeckt sind, solange sie die API-Anforderungen erfüllen und die Tests im TCK bestehen. Seit der Veröffentlichung der Version 1.0.1 am 9. August 2017 wurden verschiedene Verbesserungen in Bezug auf die Genauigkeit der Spezifikation, Verbesserungen des TCK und andere Klarstellungen vorgenommen. Die Spezifikation und die Schnittstellen blieben jedoch vollständig abwärtskompatibel zur Version 1.0.0, um die Übernahme durch zukünftige Erweiterungen zu erleichtern und die Kompatibilität mit anderen Standards zu verbessern. (vgl. Maurer, Kuhn, und Dokuka 2023)

Die Schlüsselkonzepte der Spezifikation lassen sich wie folgt zusammenfassen. Zuerst beschreibt die Spezifikation die reaktive Programmierung als ein Programmierparadigma, das sich auf den

Umgang mit asynchronen Datenströmen und die Verbreitung von Änderungen konzentriert. Es eignet sich besonders für Anwendungen, die hohe Skalierbarkeit und effiziente Thread-Nutzung erfordern. Aus der reaktiven Programmierung ergibt sich das kooperative Multithreading. Dadurch kann die Laufzeitumgebung die Thread-Planung besser verwalten, indem sie erkennt, wann Threads in Gebrauch oder im Leerlauf sind. (vgl. Long 2020, Kap. 5)

In der Spezifikation werden außerdem verschiedene Schnittstellen definiert, wie zum Beispiel die Publisher<T>-Schnittstelle, die in Listing 3 veranschaulicht wird.

```
package org.reactivestreams;

public interface Publisher<T> {

    public void subscribe(Subscriber<? super T> s);
}
```

Listing 3 Publisher Schnittstelle der Reactive Streams Spezifikation

Diese Schnittstelle wird verwendet, um Daten vom Typ T an Subscriber zu senden. Dafür wird eine subscribe-Methode definiert, um Subscriber mit dem Publisher zu verbinden. (vgl. Long 2020, Kap. 5)

Wie in der Subscriber<T>-Schnittstelle in Listing 4 zu sehen ist, empfängt sie Daten von einem Publisher vom Typ T. Sie verfügt über Methoden wie onSubscribe(), onNext(), onError() und onComplete(), um verschiedene Aspekte des Datenstroms zu handhaben. Wenn sich ein Subscriber über die onSubscribe-Methode registriert, erhält er eine Subscription. Dabei ist die Subscription eine der wichtigsten Klassen in der gesamten reaktiven Stream Spezifikation. Die onError-Methode verarbeitet alle im Stream aufgetretenen Fehler. Fehler- oder Exception-Instanzen sind nur eine weitere Art von Daten in der Spezifikation und werden auf die gleiche Weise wie normale Daten verarbeitet. (vgl. Long 2020, Kap. 5)

```
package org.reactivestreams;

public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable t);

    public void onComplete();
}
```

Listing 4 Subscriber Schnittstelle der Reactive Streams Spezifikation

Die Schnittstelle Subscription<T> in Listing 5 dient als Verbindung zwischen Publisher und Subscriber. Sie enthält Methoden wie request(long n) und cancel(), die es dem Subscriber ermögli-

chen, den Datenfluss zu kontrollieren. Jede Subscription, die der Subscriber in der Methode onSubscribe erhält, ist eindeutig. Neue Subscriber Instanzen erzeugen neue Subscription Instanzen. Eine Subscription ist eine Verbindung zwischen dem Publisher und dem Subscriber. Der Subscriber verwendet die Subscription, um weitere Daten mittels request(int) anzufordern. Letzteres ist entscheidend, da der Subscriber damit den Datenfluss steuern und die Geschwindigkeit der Verarbeitung bestimmen kann. Der Publisher erzeugt dadurch keine Daten, die der Subscriber nicht angefordert hat, so dass der Subscriber nicht überlastet wird. Die Subscription ermöglicht es dem Subscriber, mehr Daten anzufordern, wenn er bereit ist, diese zu verarbeiten. Dieser kontrollierte Datenverbrauch wird als Backpressure bezeichnet. (vgl. Long 2020, Kap. 5)

```
package org.reactivestreams;

public interface Subscription {

    public void request(long n);

    public void cancel();
}
```

Listing 5 Subscription Schnittstelle der Reactive Streams Spezifikation

Processor<T> ist die letzte Schnittstelle in der Spezifikation, wie in Listing 6 dargestellt. Es handelt sich dabei um eine Brücke zwischen dem Publisher und dem Subscriber, die sowohl die Schnittstelle Publisher als auch Subscriber implementiert. Der Processor ist somit ein Publisher und ein Subscriber zugleich. Der Processor schließt eine Subscription bei dem Publisher ab, verarbeitet die Daten weiter, indem er sie beispielsweise transformiert, und bietet sie als neuer Publisher einem Subscriber an. Eine mögliche Art der Weiterverarbeitung könnte das Filtern oder Mappen sein. Die Weiterverarbeitung kann jedoch auch deutlich komplexer aussehen. (vgl. Long 2020, Kap. 5)

```
package org.reactivestreams;

public interface Processor<T, R> extends Subscriber<T>, Publisher<R>
{}
```

Listing 6 Processor Schnittstelle der Reactive Streams Spezifikation

Abbildung 7 veranschaulicht das Zusammenspiel der Typen. Zuerst wird ein Subscriber beim Publisher registriert. Dies geschieht über die Methode subscribe. Anschließend ruft der Publisher die Methode Subscriber.onSubscribe() auf, mit der eine Subscription für die Kommunikation zwischen dem Subscriber und dem Publisher an den Subscriber übergeben wird. Daraufhin fordert der Subscriber über Subscription.requests() eine Menge von Datensätzen an. Der Subscriber spezifiziert dabei die maximale Anzahl an Daten, die er aktuell verarbeiten kann. Um die Daten zu übergeben, ruft der Publisher daraufhin die Methode Subscriber.onNext() auf, wodurch ein Datensatz übergeben wird. Dies darf er höchstens so oft machen, wie Datensätze angefordert wurden. Schließlich wird die Methode Subscription.cancel() vom Subscriber aufgerufen, um die Übergabe von Datensätzen zu beenden. Der Publisher kann nach dem Aufruf der cancel-Mehtode bestehende onNext() Aufrufe jedoch noch beenden.

Wichtig ist, dass der Subscriber jederzeit Subscription.request() aufrufen kann, ohne auf den Empfang aller Datensätze aus dem letzten Request warten zu müssen. Auf diese Weise kann der

Subscriber sicherstellen, dass seine Pipeline immer ausreichend gefüllt ist. Dabei ist die Anzahl der angeforderten Datensätze zu addieren. Fordert der Subscriber beispielsweise zunächst fünf Datensätze an und fordert nach drei Datensätzen die nächsten fünf Datensätze an, so werden ihm insgesamt zehn Datensätze geliefert. Der Publisher kann durch den Aufruf von Subscriber.onComplete() signalisieren, dass er keine weiteren Daten senden wird. Dies beendet die Beziehung zwischen dem Subscriber und dem Publisher. Zur Meldung eines Fehlers an den Subscriber kann der Publisher die Methode Subscriber.onError(Throwable t) aufrufen. Diese Methode dient nicht zur Validierung, sondern zeigt eine schwerwiegende Fehlersituation an, bei der keine sinnvolle Weiterverwendung des Streams mehr möglich ist. In diesem Fall endet auch die Beziehung zwischen Publisher und Subscriber endgültig. (vgl. Grammes und Schaal 2015)

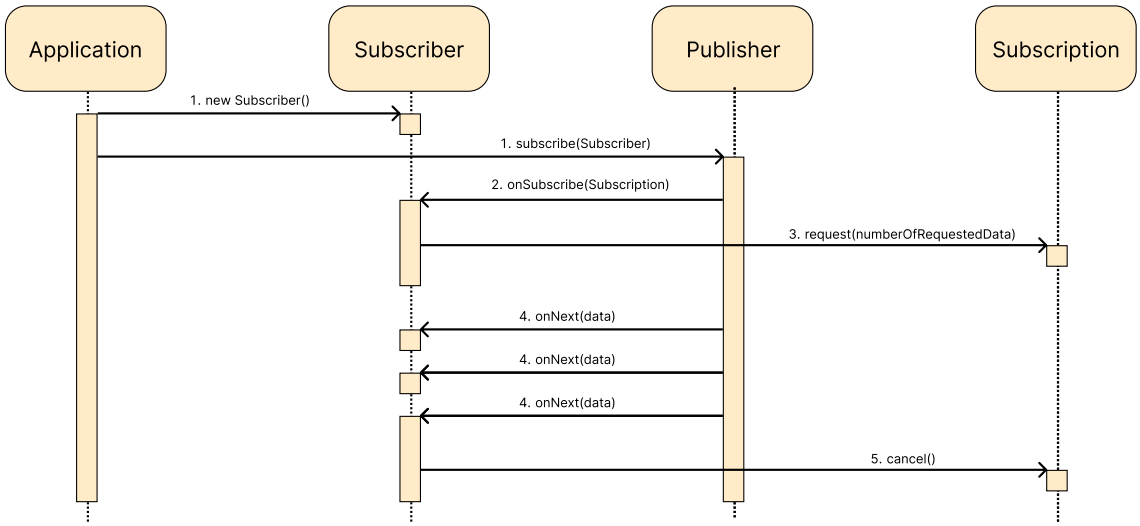

Abbildung 7 Interaktion von Subscriber, Publisher und Subscription (vgl. Grammes und Schaal 2015, Abb. 3)

Die Typen aus der Spezifikation wurden seit Version 9 in das JDK integriert. Reactive Streams bieten somit einen strukturierten Ansatz zum Erstellen von Anwendungen, die große Datenmengen, mehrere Benutzer und Netzwerkanrufe effizient handhaben können, mit einem Fokus auf nicht-blockierender, asynchroner Verarbeitung. Dieser Ansatz ist bisher in modernen Java-Anwendungen und Diensten unerlässlich, wo Leistung und Ressourceneffizienz entscheidend sind. (vgl. Long 2020, Kap. 5)

3.3.3 Anwendungsbeispiel für die reaktive Programmierung

Wie in dem vorherigen Abschnitt erklärt, bildet der Publisher das Kernstück der reaktiven Programmierung. Sie stellen eine Sammlung von Werten oder Ereignissen dar und reagieren auf Ereignisse, indem sie diese an Subscribern weiterleiten. Publisher können sowohl asynchrone als auch synchrone Datenströme erzeugen. Der Publisher ist somit die ursprüngliche Datenquelle. Es sendet eine variable Anzahl von Elementen, deren Menge flexibel und nicht auf eine feste Menge beschränkt ist. Außerdem kann der Publisher die Elemente erfolgreich übertragen und im Problemfall Fehlermeldungen ausgeben. Darüber hinaus ist es in der Lage, jederzeit eine beliebige

Anzahl von Subscribern zu bedienen. Im Kontext eines E-Commerce-Systems kann ein Publisher den kontinuierlichen Fluss von Bestellereignissen darstellen, wie in Abbildung 8 gezeigt. Diese enthalten relevante Informationen wie Kundenname, Produkt und Menge. Neue Bestellungen können in unregelmäßigen Abständen hinzukommen.

Abbildung 8 Bestellstrom Publisher

In diesem Szenario stellt der Publisher Bestellstrom genau diesen kontinuierlichen Strom von Bestellereignissen dar, die erfasst werden, wenn Kunden Bestellungen aufgeben. Die Subscriber sind wie bereits erwähnt die Konsumenten der von den Publisher bereitgestellten Werten. Sie definieren Methoden, um auf die durch ein Publisher bereitgestellten Werte, Fehler oder auf deren Fertigstellung zu reagieren. Durch die Subscription eines Publishers mit einem Subscriber wird die Verarbeitung der gesendeten Werte angestoßen.

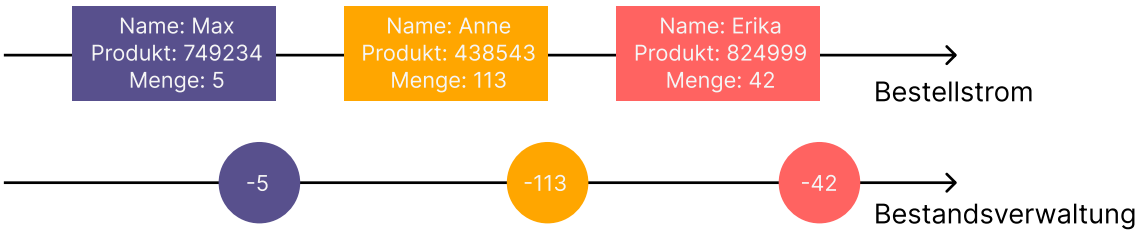

Abbildung 9 Bestandsverwaltung Subscriber

Im Zusammenhang mit dem E-Commerce-System kann der Bestandsverwaltungs-Subscriber als Beispiel für einen Subscriber in der Abbildung 9 betrachtet werden. Er reagiert auf Bestelleingänge, aktualisiert den Lagerbestand automatisch und gibt eine Fehlermeldung aus, falls die bestellte Produktmenge nicht mehr vorhanden ist. Bei Bedarf könnte sogar das Versandteam informiert werden.

Wie bereits erwähnt, zielt der Processor darauf ab, Einfluss auf die von dem Publisher veröffentlichten Daten zu nehmen, indem er Ereignisse transformiert oder kombiniert. Genauer gesagt ermöglichen Processors die Manipulation von Publishern durch Anwendung von Filtern, Transformationen oder anderen Operationen auf Datenströme. Dadurch generieren sie einen neuen Publisher und bieten somit eine Plattform für die Ausführung komplexer Logik auf Datenströmen. Im gegebenen Szenario kann ein Filter-Processor genutzt werden, um Bestellungsereignisse auszuschließen, die eine spezifische Mengengrenze für ein Produkt überschreiten. Praktisch könnte der Filter-Processor verwendet werden, um zu überprüfen, ob Bestellungsereignisse eine Mindestmenge erreichen, wie in der Abbildung 10 gezeigt. Wenn eine Bestellung mehr als 100 Einheiten eines Produkts umfasst, aktiviert sich der Filter-Processor und leitet diese speziellen Bestellungen an ein Publisher mit dem Namen Großbestellungen weiter.

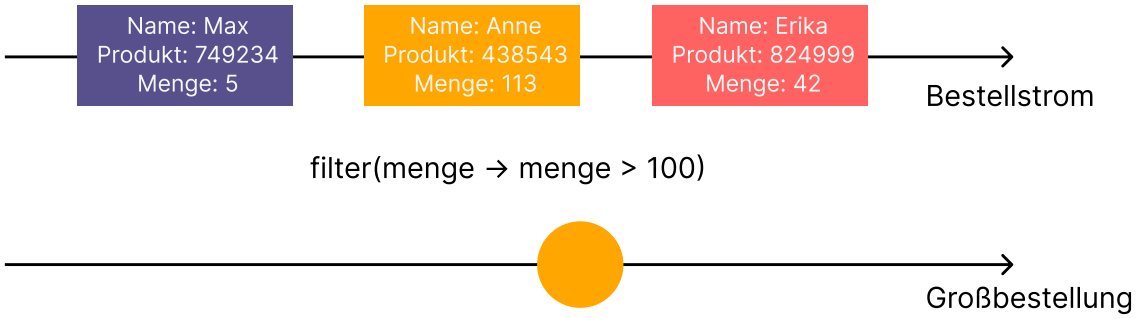

Abbildung 10 Großbestellungs-Filter Processor

Wie bereits erwähnt, ist Backpressure ein weiteres wichtiges Konzept, welches das Gleichgewicht zwischen der Produktion und dem Konsum von Daten aufrechterhält, insbesondere wenn die Rate der eingehenden Aufgaben seitens des Produzenten höher ist als die Fähigkeit des Konsumenten, diese zu verarbeiten. Für Backpressure gibt es verschiedene Konzepte. Eines davon wird durch Abbildung 11 veranschaulicht. Die Methode onBackpressureBuffer erstellt einen Puffer für alle noch nicht verarbeiteten Ereignisse des Publishers. Sobald der Subscriber des Publishers in der Lage ist, diese Daten zu verarbeiten, werden sie aus dem Puffer entnommen.

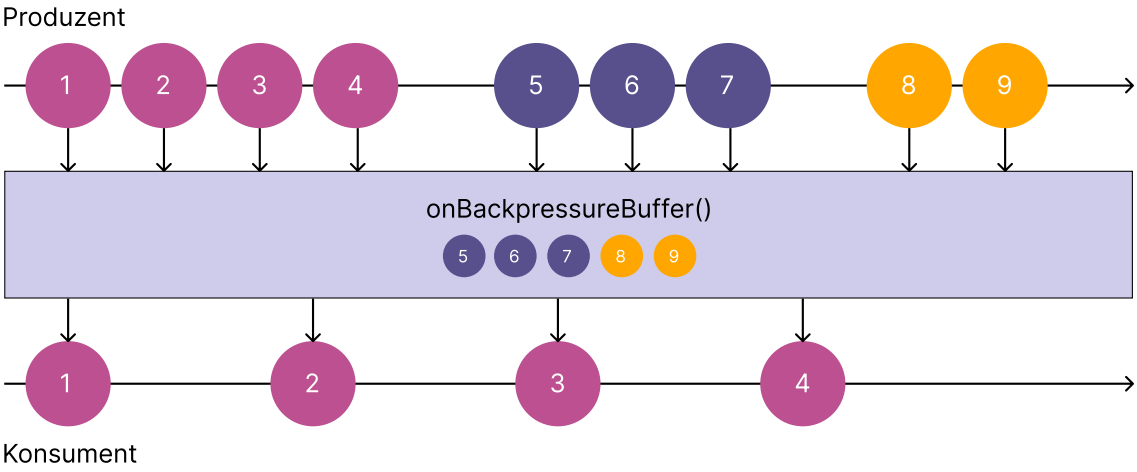

Abbildung 11 Beispiel für Backpressure (vgl. ReactiveX 2023)

Im gegebenen Szenario könnte Backpressure beispielsweise in Zeiten mit hohem Bestellaufkommen dazu beitragen, das Gleichgewicht zwischen der Bestellrate und der Fähigkeit des Systems, diese Bestellungen zu verarbeiten, aufrechtzuerhalten. Insbesondere im Fall des Bestandsverwaltungs-Subscriber könnte dies sinnvoll sein, da dieser in Zeiten hoher Auslastung viele Schreiboperationen an eine Datenbank senden müsste. Dies kann durch das Bereitstellen zusätzlicher Ressourcen oder durch Puffern der Bestellungen bis zum Zeitpunkt der Verarbeitung erfolgen. Durch diese Maßnahmen kann das System stabilisiert werden.

3.3.4 Vorteile und Herausforderungen

Reaktive Programmierung ist ein beliebtes Werkzeug für Entwickler geworden, die auf die sich ständig ändernden Anforderungen moderner Softwareentwicklungsumgebungen reagieren müssen. Das Programmierparadigma bietet eine Lösung für die Entwicklung von reaktionsfähigen,

robusten und skalierbaren Anwendungen, die effektiv auf Ereignisse reagieren und eine verbesserte Benutzererfahrung bieten können. Wie bei jeder Technologie gibt es jedoch auch bei der reaktiven Programmierung eigene Vorteile und Herausforderungen.

Asynchrone und nicht-blockierende Codeausführung ermöglichen reaktiven Systemen eine effiziente Nutzung von Ressourcen wie CPU und Speicher, was zu einer verbesserten Leistung führt. Dies ist ein entscheidender Vorteil, insbesondere in Umgebungen mit begrenzten Ressourcen oder in Anwendungen, die eine hohe Leistung erfordern und großen Durchsatz verlangen. Im Vergleich zu Multithreading erfordert sie einen geringeren Speicherverbrauch, was zu einer verbesserten Leistung führt. Auch die Skalierbarkeit spielt eine zentrale Rolle. Die reaktive Programmierung ermöglicht eine effiziente Skalierung, um unter wechselnden Lastbedingungen stets eine gleichbleibende Leistung zu gewährleisten.

Bei der Handhabung von asynchronen Datenströmen und der Bewältigung von Nebenläufigkeit bietet die reaktive Programmierung eindeutig Vorteile. Allerdings sind bei diesem Ansatz einige Herausforderungen und Probleme zu berücksichtigen. So erfordert die Umstellung von einem traditionellen, sequenziellen Programmiermodell auf reaktive Ansätze oft ein Umdenken und eine Umstrukturierung des Codes. Das kann in großen und komplexen Anwendungen zu erhöhter Komplexität führen. Entwickler mit wenig Erfahrung in reaktiver Programmierung können Schwierigkeiten beim Verständnis von Observables, Observer und Operatoren haben. Obwohl es eine Spezifikation für reaktive Streams gibt, besteht immer noch keine vollständige Standardisierung. Dies kann zudem zu Kompatibilitätsproblemen zwischen verschiedenen reaktiven Bibliotheken führen und die Code-Portabilität beeinträchtigen. Darüber hinaus kann das Debuggen von reaktiven Code komplexer sein als bei traditionellen Ansätzen. Das Verfolgen von Datenströmen und die Identifizierung von Fehlern in komplexen, reaktiven Ketten kann zeitaufwändig sein. Zudem können Fehler, die in einem Teil des Datenstroms auftreten, den gesamten Datenfluss beeinträchtigen. Außerdem kann die Einarbeitung in reaktive Programmierung zeitaufwändig sein, vor allem für Entwickler, die zuvor keine Erfahrung damit gesammelt haben. Die Verwendung von reaktiven Mustern erfordert oft ein Verständnis für funktionale Programmierung und asynchrone Konzepte.

Trotz dieser Herausforderungen ist reaktive Programmierung eine wertvolle Technik, um die Anforderungen moderner Anwendungen an Skalierbarkeit, Reaktionsfähigkeit und Effizienz zu erfüllen.

3.3.5 Zusammenfassung

Reaktive Programmierung, ein Paradigma, das sich auf die Verarbeitung und Propagierung von Datenströmen konzentriert, ermöglicht die Erstellung asynchroner Verarbeitungspipelines durch deklarativen Code. Diese Art der Programmierung ist ideal für hochreaktive und skalierbare Umgebungen und findet Anwendung in Bereichen wie GUI- und Webprogrammierung, Microservices und reaktive Systeme. Der Ursprung der reaktiven Programmierung kann auf das Reactive Manifesto von 2014 zurückgeführt werden, dass die Grundprinzipien Reaktivität, Robustheit, Elastizität und nachrichtengesteuerte Architekturen definiert.

Zu den Kernkonzepten der reaktiven Programmierung gehören Observables, Observer und Operatoren, die eine effiziente Verwaltung von Datenströmen und die automatische Propagierung von Änderungen ermöglichen. Backpressure ist ein weiteres wichtiges Konzept, um das Gleichgewicht zwischen Datenproduktion und -konsum aufrechtzuerhalten, insbesondere in Überlastungssituationen.

Reaktive Programmierung in Java wird durch reaktive Erweiterungen ermöglicht. Die reaktive Stream Spezifikation ist ein Standard für die Verarbeitung asynchroner Streams mit nicht-blockierender Rückkopplung und hat seit ihrer Veröffentlichung im August 2017 mehrere Verbesserungen erfahren, bleibt jedoch abwärtskompatibel, um die Adoption zu erleichtern und die Kompatibilität mit anderen Standards zu verbessern.

Zu den Vorteilen der reaktiven Programmierung gehören eine effiziente Ressourcennutzung, eine verbesserte Leistung und Skalierbarkeit sowie Vorteile bei der Handhabung asynchroner Datenströme und der Nebenläufigkeit. Herausforderungen ergeben sich aus dem Übergang von traditionellen, sequenziellen Programmiermodellen zu reaktiven Ansätzen, die zu erhöhter Komplexität, möglichen Kompatibilitätsproblemen zwischen verschiedenen reaktiven Bibliotheken und Schwierigkeiten beim Debugging führen können. Trotz dieser Herausforderungen stellt die reaktive Programmierung eine wertvolle Technik dar, um den Anforderungen moderner Anwendungen gerecht zu werden.

Als Überleitung zum nächsten Kapitel, dem Projekt Loom, soll an dieser Stelle auf die Möglichkeiten zur Verbesserung der nebenläufigen Programmierung eingegangen werden. Projekt Loom zielt darauf ab, die Komplexität der Entwicklung, Wartung und Überwachung von nebenläufigen Hochdurchsatzanwendungen zu reduzieren. Durch die Einführung virtueller Threads in Java bietet Projekt Loom ein leichtgewichtiges Nebenläufigkeitsmodell, dass eine effiziente Verwaltung von Threads auf einer neuen Ebene ermöglicht, ideal für die Thread-per-Request Programmierung.

3.4 Projekt-Loom

Das Projekt Loom ist eine Ansatz der OpenJDK Community zur erheblichen Reduzierung des Aufwands für das Schreiben, der Wartung und Überwachung von nebenläufigen Hochdurchsatzanwendungen. Im Kern hat Projekt Loom das Ziel, ein Hochdurchsatz-, leichtgewichtiges Nebenläufigkeitsmodell in Java zu ermöglichen, welches durch das Einführen von virtuellen Threads erreicht wird. (vgl. Pressler 2020)

Virtuelle Threads wurden als Vorschau in der Java-Version 19 im Jahr 2022 als Lösung für die Beschränkungen des traditionellen Nebenläufigkeitsmodells eingeführt. Im Gegensatz zu nativen Threads sind virtuelle Threads leichtgewichtiger und werden ähnlich wie die Green-Threads von der JVM verwaltet, und nicht vom Betriebssystem. Im Rahmen dieses Ansatzes bietet die Implementierung von virtuellen Threads eine leistungsfähige Möglichkeit, Threads auf einer neuen Ebene zu verwalten. Dies macht sie ideal für die Thread-per-Request-Programmierung, bei der eine große Anzahl von virtuellen Threads ohne Durchsatzreduzierung erstellt werden können. Die Einführung von Projekt Loom und virtuellen Threads gilt als eine der bedeutendsten Entwicklungen seit langer Zeit in der Geschichte von Java. Die virtuellen Threads wurden im September 2023 als Teil von Java 21 LTS veröffentlicht. Mit Projekt Loom wird eine erhebliche Erleichterung beim Schreiben skalierbarer Software mit Millionen von nebenläufigen Anfragen erwartet. (vgl. Oleschuk 2023)

Eine bemerkenswerte Eigenschaft von Projekt Loom ist, dass die Ressourcen der echten Java-Threads von der JVM autonom verwaltet werden. Wenn ein virtueller Thread eine blockierende Operation ausführt, wird die JVM nicht mehr davon beeinträchtigt und die echten Hardware-Ressourcen können weiterhin genutzt werden.

3.4.1 Virtuelle Threads

Jede Implementierung eines Threads, ob leichtgewichtig oder schwergewichtig, hängt von zwei Konstrukten ab: der Aufgabe und dem Scheduler. Die zentralen Elemente der Architektur virtueller Threads sind daher die Aufgabe und der Scheduler, wobei eine Aufgabe auch als Continuation (Fortsetzung) bezeichnet wird. Der Scheduler ist verantwortlich für die Zuweisung der Aufgaben an die CPU und die erneute Zuweisung der CPU einer pausierten Aufgabe. Bisher setzte Java auf Betriebssystem-Implementierungen sowohl für die Continuation als auch für den Scheduler. (vgl. Baeldung 2019)

Um eine Continuation im Betriebssystem auszuführen, ist es bisher notwendig, den gesamten Aufrufstapel zu speichern und bei der Wiederaufnahme wieder abzurufen. Dies erfordert aufgrund der Betriebssystem-Implementierung von Java-Threads, die neben dem Java-Aufrufstapel auch den nativen Aufrufstapel enthält, einen großen Platzbedarf. Ein noch größeres Problem besteht in der Verwendung des Betriebssystem-Scheduler. Da dieser im Kernel-Modus läuft, gibt es keine Unterscheidung zwischen Threads und jede CPU-Anfrage wird auf dieselbe Art behandelt. Diese Art des Schedulings ist insbesondere für Java-Anwendungen nicht optimal. (vgl. Baeldung 2019)

Im Projekt Loom wird die Implementierung eines virtuellen Threads durch die Bereitstellung der Klassen VirtualThread, Continuation und neue Ansätze für den Scheduler erreicht.

In den Prototypen des OpenJDK wird eine neue Klasse namens VirtualThread hinzugefügt. Sie funktioniert ähnlich wie die Thread-Klasse. (vgl. Baeldung 2019) Die Implementierung der Klasse VirtualThread soll es Entwicklern ermöglichen, dieselbe einfache Abstraktion wie bei traditionellen Threads zu verwenden, jedoch mit besserer Leistung und geringerem Platzbedarf (vgl. Pressler 2020).

Dabei gibt es einige signifikante Unterschiede zwischen nativen und virtuellen Threads. Für ein besseres Verständnis für die Implementierung wird mit dem Listing 7 die Implementierung eines virtuelle Threads veranschaulicht. Virtuelle Threads können jede Aufgabe in eine interne Benutzer-Modus-Continuation umwandeln, was es ermöglicht, die Aufgabe in der Java-Laufzeitumgebung zu unterbrechen und fortzusetzen, anstatt im Betriebssystem-Kernel zu arbeiten. Außerdem wird für virtuelle Threads ein Benutzer-Modus-Scheduler verwendet. Im Rahmen des Projekt Looms soll sich dabei nicht auf eine umfangreiche Erforschung eines Scheduler-Designs konzentriert werden, da das Team der Ansicht ist, dass der bereits in Java vorhandene ForkJoinPool-Scheduler als sehr guter VirtualThread-Scheduler dienen kann (vgl. Pressler 2020).

```
final class VirtualThread extends BaseVirtualThread {
    // ForkJoinPool als Standard Scheduler
    private static final ForkJoinPool DEFAULT_SCHEDULER =
              createDefaultScheduler();

    // Benutzer-Modus-Scheduler
    private final Executor scheduler;

    // Benutzer-Modus-Continuation
    private final Continuation cont;

    // Statuscode des virtuellen Threads
    private volatile int state;

    // Statuscodes eines virtuellen Threads
    private static final int NEW = 0;
    private static final int STARTED = 1;
    private static final int RUNNABLE = 2;
    private static final int RUNNING = 3;
    private static final int PARKING = 4;
    private static final int PARKED = 5;
    private static final int PINNED = 6;
    private static final int YIELDING = 7;
    private static final int TERMINATED = 99;
    private static final int SUSPENDED = 256;
    private static final int RUNNABLE_SUSPENDED = 258;
    private static final int PARKED_SUSPENDED = 261;

    // Plattform-Thread als CarrierThread
    private volatile Thread carrierThread;

    // weitere Attribute
    ...
    // Methoden
    ...
}
```

Listing 7 Vereinfachte Implementierung der VirtualThread-Klasse

Um virtuelle Threads auf der CPU auszuführen, müssen diese einem nativen Thread zugewiesen werden, welcher auch als Carrier-Thread bezeichnet wird. Das Einplanen von virtuellen Threads, die dem Carrier Thread zugewiesen werden sollen, wird nun vom Benutzer-Modus-Scheduler entschieden. Auf diese Weise können unzählige virtuelle Threads auf einem nativen Thread ausgeführt werden, wodurch virtuelle Threads nicht den Problemen von nativen Threads unterliegen und im Wesentlichen auf Hunderttausende oder Millionen skaliert werden können, während native Threads nur auf einige Tausend skaliert, werden können. (vgl. Behler 2022)

Im Abschnitt 3.2 wurde bereits erwähnt, dass die ersten Threads in Java Green Threads waren. Diese führten eine N:1-Zuordnung von Green Threads zu einem dedizierten nativen Thread durch. Green Threads waren sehr leichtgewichtig, aber aufgrund der Verwendung von nur einem einzigen nativen Thread hatten sie erhebliche Skalierungsprobleme, da das Potenzial von Multicore-Systemen nicht ausgeschöpft werden konnte. Aus diesem Grund haben sich nach den Green-Threads die nativen Java-Threads durchgesetzt, da sie eine direkte Zuordnung von Java-Threads zu Betriebssystem-Threads ermöglichen. Mit der Möglichkeit, mehrere tausend native Threads durch native Threads zu erstellen, bot Java eine solide Plattform für Multithreading-Funktionalitäten. Allerdings traten im Laufe der Entwicklung moderner Anwendungen Skalierungsprobleme bei nativen Threads auf, die mit einem gewissen Overhead einhergingen.

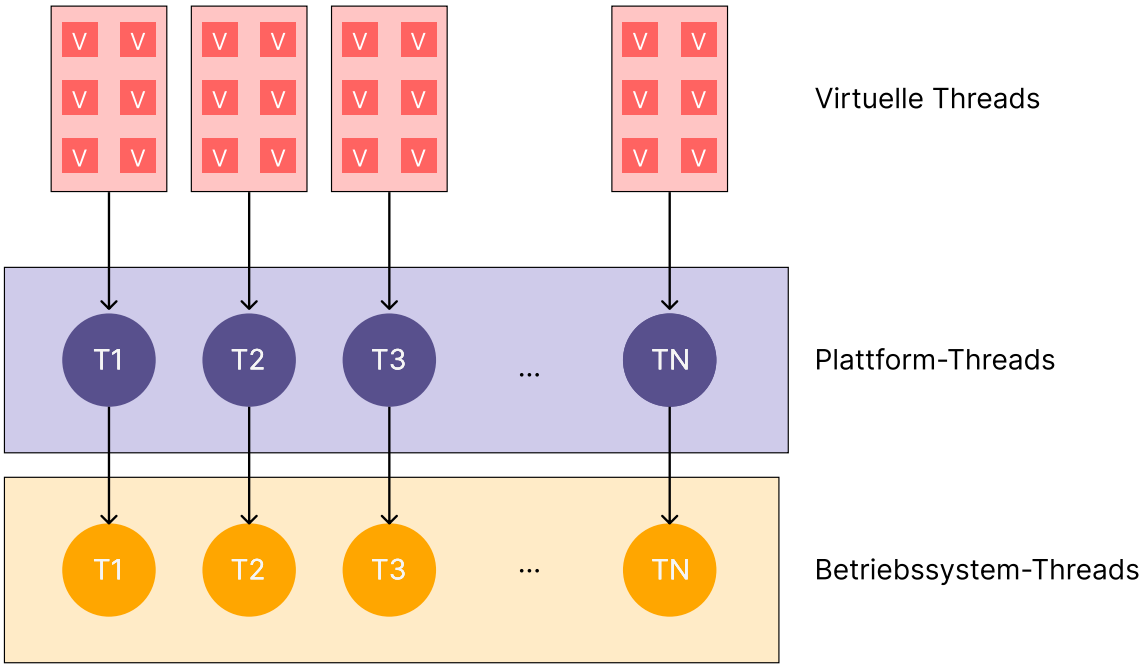

Abbildung 12 Zuweisung von virtuelle Threads zu nativen Threads

Durch die Einführung von virtuellen Threads werden die Vorzüge beider Ansätze vereint. Virtuelle Threads ermöglichen die M:N-Zuordnung von virtuellen Threads auf native Java-Threads, wie durch die Abbildung 12 veranschaulicht. Ein nativer Thread fungiert als Carrier-Thread für eine unbegrenzte Anzahl virtueller Threads. Ein bedeutender Vorteil dabei ist die Speicherung des Stacks von virtuellen Threads. Virtuelle Threads haben einen dynamisch veränderbaren Stack, der im Heap-Speicher abgelegt ist und sich an das jeweilige Problem anpasst. Im Vergleich zu nativen Threads ist dieser Stack kleiner. Virtuelle Threads haben typischerweise einen flacheren Call-Stack und werden für blockierende Aufgaben eingesetzt, beispielsweise beim Warten auf I/O-

Operationen (vgl. Oracle 2023b). Da native Threads selbst bis zu einem gewissen Maß hochskalierbar sind, können über sie eine große Anzahl von virtuellen Threads bereitgestellt werden. Dadurch kann eine sehr ähnlich hohe Skalierbarkeit einer Anwendung im Vergleich zur reaktiven Programmierung gewährleistet werden (vgl. Piazzolla 2022).

Durch die Zuweisung von virtuellen Threads zu einem Carrier-Thread durch den Benutzer-Modus-Scheduler wird der virtuelle Thread dann auf dem Carrier-Thread nach den Vorgaben des Betriebssystem-Schedulers ausgeführt. Ein großer Vorteil besteht darin, dass der Benutzer-Modus-Scheduler jederzeit durch Verwendung von Benutzer-Modus-Continuations einen virtuellen Thread stoppen kann, wobei der Thread-Stack leichtgewichtig und performant auf dem Heap abgelegt werden kann. Dieser virtuelle Thread wird dann vom Carrier-Thread entfernt und ein anderer virtueller Thread wird ihm zur Bearbeitung durch Laden des Thread-Stacks vom Heap zugewiesen. Wie bereits erwähnt ist ein Scheduler eine Systemkomponente, die die Verwaltung der Thread-Ausführung auf der CPU übernimmt. Threads in traditionellen Nebenläufigkeitsmodellen, wie sie in Java verwendet werden, sind sehr komplex und werden eins zu eins einem Betriebssystem-Thread zugeordnet. Dabei liegt die Aufgabe der Thread-Planung beim Betriebssystem. Mit dem Projekt Loom wird ein neuer Ansatz für das Scheduling vorgeschlagen. Die Implementierung ermöglicht den Einsatz von austauschbaren Benutzer-Modus-Scheduler zusammen mit virtuellen Threads, wobei ForkJoinPool als Standard-Scheduler im asynchronen Modus verwendet wird (vgl. Baeldung 2019; vgl. Pressler 2021).

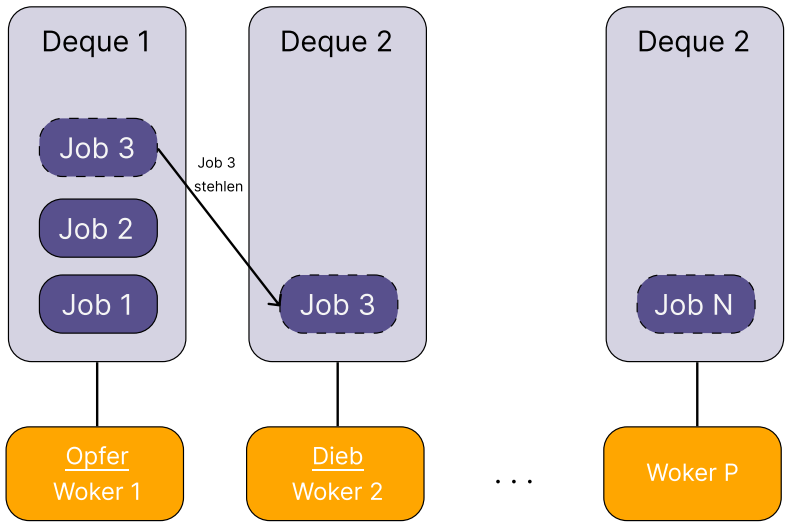

Abbildung 13 Beispiel für den Work-Stealing-Algorithmus (vgl. Charousset 2020)

Der ForkJoinPool arbeitet nach dem Work-Stealing-Algorithmus. Jeder Thread verwaltet eine Task-Deque (doppelseitige Warteschlange) und führt die Aufgabe von seinem Kopf aus. Dabei blockieren inaktive Threads nicht, wenn ihre Deque leer ist und sie auf eine neue Aufgabe warten. Stattdessen entnehmen sie eine Aufgabe vom Ende der Deque eines anderen Threads und arbeiten diese ab. Im Rahmen von Projekt Loom werden virtuelle Threads mithilfe einer FIFO-Warteschlange geplant, die von einem dedizierten ForkJoinPool verarbeitet wird. Dabei ist der Standard-Scheduler in der Klasse java.lang.VirtualThread definiert. (vgl. Baeldung 2020)

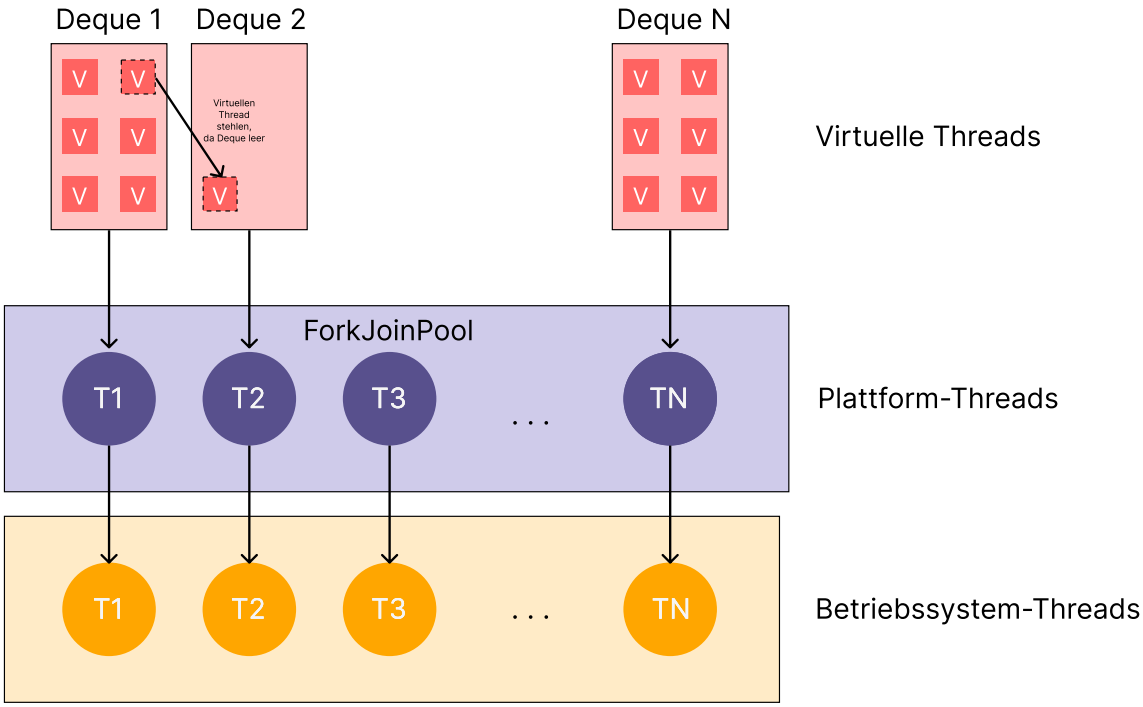

Abbildung 14 Beispiel für das ForkJoinPool-Scheduling in Projekt Loom (vgl. Ciocîrlan 2023)

Wie die Abbildung 14 veranschaulicht bedeutet das, dass die virtuellen Threads in der Deque des ForkJoinPools abgelegt und verwaltet werden. Wenn ein virtueller Thread dem Carrier-Thread durch den Scheduler zugewiesen wird, führt der Carrier-Thread die Continuation des virtuellen Threads auf der CPU aus. Dabei kann die Continuation jederzeit angehalten werden und der Carrier-Thread übernimmt eine andere Continuation.

Der Begriff Continuation (oder auch Ko-Routine) bezieht sich dabei auf eine Sequenz von Anweisungen, die an einem bestimmten Punkt unterbrochen werden und zu einem späteren Zeitpunkt vom Aufrufer fortgesetzt werden können. Jede Continuation besitzt einen Einstiegspunkt sowie einen Unterbrechungspunkt (yield point), an dem die Ausführung pausiert wird. Bei Fortsetzung der Continuation durch den Aufrufer kehrt die Steuerung zum letzten Unterbrechungspunkt zurück. Dieses Unterbrechen und Wiederaufnehmen findet durch das Projekt Loom in der Laufzeitumgebung der Sprache statt und nicht mehr im Betriebssystem, wodurch aufwendige Kontextwechsel zwischen nativen Threads vermieden werden. (vgl. Baeldung 2019) Die Funktionsweise einer Continuation erinnert ein wenig an die eines Debuggers. Sie ermöglicht das Anhalten der Ausführung an einem bestimmten Punkt im Code und das Speichern des Kontexts zu diesem Zeitpunkt für die spätere Fortsetzung, ähnlich wie ein Debugger. Der gespeicherte Kontext kann dann zu einem späteren Zeitpunkt wieder aufgenommen werden, um die Ausführung genau an der Stelle fortzusetzen, an der sie unterbrochen wurde.

Die Continuation ist ein wichtiger Bestandteil dieses Projektes, da die Implementierung von virtuellen Threads auf der Klasse basiert. Eine Continuation könnte im Zusammenhang mit virtuellen Threads auch als Zustandsautomat mit vielen Zuständen betrachtet werden (vgl. Ciocîrlan 2023). Das folgende Diagramm in der Abbildung 15 fasst die Beziehungen zwischen diesen Zuständen zusammen.

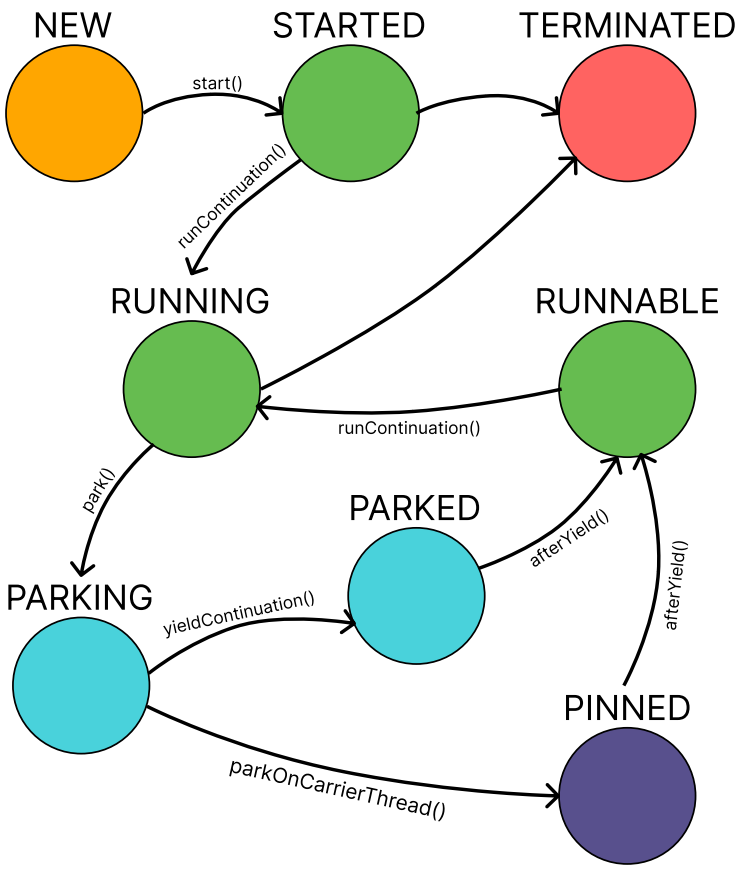

Abbildung 15 Lebenszyklus eines virtuellen Threads (vgl. Ciocîrlan 2023)

Nachdem ein neues VirtualThread-Objekt instanziiert wurde, befindet es sich im Zustand NEW. Durch Aufruf der Start-Methode wechselt der virtuelle Thread in den Zustand STARTED und wird einem Carrier-Thread zur Ausführung zugewiesen. Die in Grün markierten Zustände (STARTED, RUNNING, RUNNABLE) veranschaulichen die Zustände, in denen der Thread an den Carrier-Thread gebunden ist. Während in den hellblauen Zuständen PARKING und PARKED die Ausführung des virtuellen Threads gestoppt wird und dieser vom Carrier-Thread entbunden wird. (vgl. Ciocîrlan 2023)

Ein Thread gelangt in den PARKING Zustand, wenn er während seines Ausführungsprozesses im RUNNING Zustand auf einen blockierenden Punkt wie beispielsweise eine I/O-Operation trifft. Das Erreichen eines blockierenden Punktes wird durch die park-Methode signalisiert. Sobald der Thread im PARKING Zustand angekommen ist, wird die yieldContinuation-Methode ausgeführt, um den virtuellen Thread vom Carrier-Thread zu lösen. Wenn dieser Vorgang erfolgreich ist, wird der Thread in den Zustand PARKED versetzt. Bei Ankunft an dieser Stelle wird die afterYield-Methode ausgeführt, welche den Thread wieder zurück in die Warteschlange des Schedulers wie beispielsweise die Deque des ForkJoinPools verschiebt. (vgl. Ciocîrlan 2023)

Allerdings gibt es Fälle, in denen eine blockierende Operation den virtuellen Thread nicht vom Carrier-Thread trennt und somit den zugrunde liegenden Carrier-Thread blockiert. Wenn die yieldContinuation-Methode nicht in der Lage ist, den Thread zu lösen, wird dieser durch die parkOnCarrierThread-Methode an seinem Carrier-Thread gebunden und in den Zustand PINNED gesetzt. Dies ist kein Fehler, sondern es handelt sich um ein Verhalten, das die Skalierbarkeit der Anwendung einschränkt. Wenn ein virtueller Thread einem Carrier-Thread angehängt wird, kann die

JVM bei entsprechender Konfiguration des Carrier-Pools stets einen neuen nativen Thread zum Carrier-Pool hinzufügen. (vgl. Ciocîrlan 2023)

Es gibt jedoch nur zwei Szenarien, in denen ein virtueller Thread mit dem Carrier-Thread verbunden und damit in den Zustand PINNED versetzt wird. Wenn er entweder Code innerhalb eines synchronisierten Blocks oder einer Methode ausführt oder wenn er eine native Methode oder eine fremde Funktion aufruft. Ein Beispiel wäre die Nutzung einer nativen Bibliothek über das Java Native Interface (JNI) (vgl. Baeldung 2018).

Zusammengefasst ermöglicht die Implementierung von Continuations im Projekt Loom eine effizientere Verwaltung des Aufrufstacks. Wenn eine Continuation angehalten wird, wird der Aufrufstack im Heap gespeichert. Bei der Wiederaufnahme der Continuation durch einen Carrier-Thread wird der Aufrufstack leichtgewichtig aus dem Heap wiederhergestellt. Das Ziel von Projekt Loom ist es, die Leichtgewichtigkeit der Continuations und virtuellen Threads zu gewährleisten, weshalb eine sorgfältige Verwaltung des Aufrufstacks von zentraler Bedeutung ist. (vgl. Baeldung 2019)

3.4.2 Überarbeitung von Java-Bibliotheken und der JVM

Die Modernisierung von Java-Bibliotheken und JVM im Rahmen des Projekt Looms ist eine grundlegende Initiative zur Bewältigung der Herausforderungen, die mit der traditionellen Thread-Handhabung in Java verbunden sind. Die klassischen Java-Bibliotheken sind hauptsächlich auf die Interaktion mit nativen, blockierenden Threads ausgerichtet, was in Szenarien mit hoher Nebenläufigkeit zu erheblichen Leistungseinbußen führt. Mit der zunehmenden Komplexität und den Anforderungen moderner Softwareanwendungen wird deutlich, dass eine effizientere Form der Nebenläufigkeit erforderlich ist, um eine bessere Leistung und Skalierbarkeit zu gewährleisten. Projekt Loom löst dieses Problem durch die Einführung von virtuellen Threads, welche eine leichte und nicht-blockierende Nebenläufigkeit ermöglichen. Im Gegensatz zur reaktiven Programmierung, welche zwar auch eine Lösung für Probleme der Nebenläufigkeit bietet, ermöglicht Projekt Loom eine einfachere und nativere Nebenläufigkeit, die leichter zu verstehen und zu verwenden ist. Um den reibungslosen Übergang zu dieser neuen Form der Nebenläufigkeit zu erleichtern und gleichzeitig die Kompatibilität und Benutzerfreundlichkeit für die Entwickler zu gewährleisten, mussten die Java-Bibliotheken und die JVM umfassend überarbeitet werden.

Die Prototypen von Loom führten signifikante Veränderungen in der JVM und der Java-Bibliothek ein, um die Unterstützung von virtuellen Threads und strukturierter Nebenläufigkeit zu gewährleisten. Das Ziel dieser Überarbeitung ist es, die Handhabung von Nebenläufigkeit zu erleichtern, um eine verbesserte Leistung und Skalierbarkeit zu erreichen.

Darüber hinaus werden spezielle Überarbeitungen in den Kernbibliotheken von java.net durchgeführt. Das Ziel ist, Funktionen und APIs zu implementieren, die eine benutzerfreundliche, durchsatzstarke, leichtgewichtige Nebenläufigkeit und neue Programmiermodelle wie strukturierte Nebenläufigkeit unterstützen.

Im Zuge des Projekts Loom werden Integrationen mit bestehenden Bibliotheken durchgeführt, um die Unterstützung virtueller Threads und die Nebenläufigkeit in Java-Anwendungen zu verbessern. Dies ist ein wichtiger Schritt, um sicherzustellen, dass Entwickler die Standard-Java-

Bibliotheken und die JVM weiterhin nutzen können, während sie gleichzeitig von den neuen Funktionen profitieren, die durch das Projekt Loom eingeführt wurden.

Zusammenfassend sind die Änderungen, die durch das Projekt Loom an den bestehenden Java-Bibliotheken und der JVM vorgenommen werden, von großer Bedeutung. Meistens liegt es nicht am vom Entwickler geschriebenen Code, wenn blockierende Operationen ausgeführt werden. Stattdessen sind es oft die von Entwicklern verwendeten Java-Bibliotheken, die blockierende I/O-Operationen durch die Verwendung nativer Threads auslösen. Durch die Überarbeitung dieser Bibliotheken wird es ermöglicht, dass der von den Entwicklern geschriebene Code in Zukunft nicht neu geschrieben werden muss, um die Vorteile der virtuellen Threads zu nutzen. Stattdessen zielt das Projekt Loom darauf ab, die bestehenden Java-Bibliotheken für die zukünftige Verwendung zu optimieren, indem diese Loom-Ready gemacht werden, während die Schnittstellen der Java-Bibliotheken unverändert bleiben.

3.4.3 Vergleich mit anderen Nebenläufigkeitsmodellen

Projekt Loom bedeutet einen signifikanten Schritt in Richtung einer einfacheren und effektiveren Nebenläufigkeit in Java. Im Vergleich zu anderen Nebenläufigkeitsmodellen und -technologien sowohl innerhalb als auch außerhalb der Java-Welt weist Projekt Loom einige bemerkenswerte Unterschiede und Vorteile auf.

Im Vergleich zur reaktiven Programmierung, die eine ereignisbasierte Verarbeitung und komplexe Programmierstrukturen erfordert, bietet das Projekt Loom eine einfache sowie effiziente Lösung für Nebenläufigkeit. Entwickler können durch die Verwendung virtueller Threads eine hohe Skalierbarkeit und Ressourceneinsparungen erreichen, ohne dabei den anspruchsvollen Code und die kognitive Belastung der reaktiven Programmierung in Kauf nehmen zu müssen.

Virtuelle Threads funktionieren ähnlich wie Coroutines-basierte Nebenläufigkeit in Kotlin und anderen Programmiersprachen. Im Vergleich zu Coroutines, die in Kotlin und einigen anderen Programmiersprachen für die Nebenläufigkeit verwendet werden, bieten virtuelle Threads in Projekt Loom eine vergleichbare Einfachheit und Effizienz. Virtuelle Threads können direkt mit der bestehenden Java-Infrastruktur und den Bibliotheken interagieren, was sie für bestehende Java-Entwickler attraktiver machen könnte (vgl. Chandrakant 2020b). Betrachtet man Coroutinen außerhalb der Java- und Kotlin-Welt, so findet man beispielsweise Golang-Goroutines[1], die eine weitere Form der leichtgewichtigen Nebenläufigkeit darstellen und in der Programmiersprache Go eine einfache und effiziente Lösung für leichtgewichtige Nebenläufigkeit bieten. Erlang-Prozesse könnten ebenfalls in Betracht gezogen werden. Erlang ist bekannt für sein starkes Nebenläufigkeitsmodell, das auf leichtgewichtigen Prozessen basiert. Zusätzlich gibt es ähnliche Konstrukte wie async/await in JavaScript oder Fibers in Ruby. All diese Konzepte ähneln dem Konzept von virtuellen Threads. (vgl. Nurkiewicz 2022)

[1] Golang Gorotuine: Leichtgewichtiger Thread der von der Go Laufzeitumgebung verwaltet wird (https://go.dev/tour/concurrency/1)

Allgemein lässt sich sagen, dass virtuelle Threads ein bedeutsamer Fortschritt in der Diskussion zur Modernisierung von Java darstellen. Im Gegensatz zu anderen Programmiersprachen verfügt Java erst seit kurzem über ein leichtgewichtiges Nebenläufigkeitsmodell durch virtuelle Threads.

3.4.4 Zusammenfassung

Das Projekt Loom stellt einen signifikanten Fortschritt in der Java-Programmierung dar, da es durch die Einführung von virtuellen Threads die Nebenläufigkeit revolutioniert. Diese Innovation bietet eine leichtgewichtige, effiziente und leicht verständliche Lösung für die Herausforderungen der Nebenläufigkeit, die traditionell mit der Verwendung von nativen, blockierenden Threads verbunden sind. Die Hauptidee des Projekt Looms ist die Implementierung virtueller Threads, die im Gegensatz zu herkömmlichen Threads von der JVM verwaltet werden und eine effizientere Thread-Verarbeitung ermöglichen. Dieses Modell ist besonders nützlich in anspruchsvollen Nebenläufigkeitsanwendungen und stellt eine attraktive Alternative zur reaktiven Programmierung dar, indem es eine einfachere und nativere Nebenläufigkeit bietet.

Im Rahmen des Projekts Loom wurden die Java-Bibliotheken und die JVM gründlich überarbeitet, um die Unterstützung für virtuelle Threads und strukturierte Nebenläufigkeit sicherzustellen. Diese Überarbeitungen sind von entscheidender Bedeutung für die Vereinfachung der Handhabung von Nebenläufigkeit sowie für die Erzielung besserer Leistung und Skalierbarkeit. Die Überarbeitungen beinhalten spezifische Anpassungen in den Kernbibliotheken von java.net sowie Integrationen mit bereits vorhandenen Bibliotheken, um die Unterstützung für virtuelle Threads und die Nebenläufigkeit in Java-Anwendungen zu optimieren.

Projekt Loom wurde im Vergleich zu anderen Nebenläufigkeitsmodellen innerhalb und außerhalb der Java-Welt untersucht, um seine Position und Vorteile besser zu verstehen. Die Ergebnisse zeigen, dass Projekt Loom eine ähnlich leichte und effiziente Nebenläufigkeit wie Coroutine-basierte Modelle in Kotlin, Golang-Goroutinen und Erlang-Prozessen bietet oder ähnliche Nebenläufigkeitsmodelle wie beispielsweise in JavaScript, Python und Ruby.

Mit dem Abschluss dieses Kapitels wird ein umfassendes Verständnis für das Projekt Loom und seine Auswirkungen auf die Nebenläufigkeit in Java erlangt. Als nahtloser Übergang zum nächsten Kapitel wird nun das Spring Framework in den Fokus gerückt, welches eine wichtige Plattform für die Java-Entwicklung darstellt. Das Spring Framework bietet umfassende Unterstützung für die Entwicklung von Java-Anwendungen und hat sich als Lösung für viele Herausforderungen bei der Entwicklung von Unternehmensanwendungen bewährt. Ähnlich wie Projekt Loom strebt auch Spring danach, die Komplexität der Entwicklung zu verringern und Entwicklern zu ermöglichen, sich auf die Anwendungslogik zu konzentrieren. Im nächsten Kapitel werden wir die Architektur des Spring Frameworks sowie seine Module und Ansätze zur Anfragenverarbeitung untersuchen, um zu verstehen, wie es Entwicklern dabei hilft, moderne, effiziente und skalierbare Anwendungen zu erstellen.

3.5 Spring Framework

Das Spring Framework ist eine wichtige Plattform für die Java-Entwicklung, die umfassende Infrastrukturunterstützung für die Erstellung von Java-Anwendungen bietet. Es übernimmt einen Großteil der Hintergrundprozesse, was Entwicklern ermöglicht, sich hauptsächlich auf die Entwicklung der Anwendungslogik zu konzentrieren. Spring ermöglicht die Entwicklung von Anwendungen aus Plain Old Java Objects und erleichtert es Entwicklern, moderne Webanwendungen zu erstellen, die Mikroservice-Architekturen, Cloud-Systeme, reaktive Verarbeitung und serverlose Workloads nutzen. (vgl. Johnson u. a. 2023; vgl. Walls 2012, Kap. 1) Das Framework hat sich in der Entwicklung von Anwendungen als effektive Antwort auf zahlreiche Herausforderungen bewährt. Es verwendet verschiedene Technologien wie die aspektorientierte Programmierung (AOP), Plain Old Java Objects und Dependency Injection (DI), um die Komplexität der Entwicklung zu verringern. Spring begann zunächst als Dependency-Injection-Container und hat sich zu einem umfangreichen Framework entwickelt. Es bietet eine Vielzahl von Funktionen und Modulen wie Datenzugriff, AOP, Messaging, Transaktionsmanagement und Webanwendungen. (vgl. Walls 2012, Kap. 1) Die Bedeutung des Spring Frameworks in der modernen Softwareentwicklung ist immens, da es eine solide Basis für die Entwicklung effizienter, skalierbarer und wartbarer Anwendungen bietet. Dank seiner modularen Architektur können die Entwickler nur die Teile des Frameworks verwenden, die sie benötigen, was eine flexible Entwicklung ermöglicht. Ferner unterstützt Spring die Entwicklung von loosely coupled (locker gekoppelten) Anwendungen durch seine Unterstützung für Dependency Injection. Dies macht das Framework zu einem wertvollen Werkzeug für Projekte jeder Größenordnung, von kleinen bis hin zu großen Unternehmensanwendungen. Der Anwendungsbereich des Spring Frameworks ist äußerst vielfältig. Es ermöglicht nicht nur die Entwicklung von Webanwendungen, sondern auch von RESTful Web Services, Mikroservices, Cloud-nativen Anwendungen und vielem mehr. Aufgrund der breiten Palette an unterstützten Funktionen und Technologien stellt Spring eine solide Grundlage für die Entwicklung in verschiedenen Anwendungsbereichen und Domänen bereit.

In den nachfolgenden Abschnitten wird die Architektur des Spring-Frameworks beschrieben, einschließlich seiner Module und Ansätze zur Anfragenverarbeitung, insbesondere Spring MVC und Spring WebFlux.

3.5.1 Architektur des Spring Frameworks

Die Architektur des Spring Frameworks ist modular und basiert auf zwei Kernprinzipien: Dependency Injection und Aspect-Oriented Programming. Dependency Injection ist ein Designprinzip, das die Kopplung zwischen verschiedenen Komponenten einer Anwendung reduziert, indem Abhängigkeiten von einer zentralen Stelle aus bereitgestellt werden (vgl. Walls 2012, Abschn. 1.1.2). Aspect-Oriented-Programming ist ein Programmierparadigma, dass die Trennung von Belangen (Separation of Concerns) innerhalb von Softwareentwürfen und -implementierungen fördert. Dabei werden Aspekte, das heißt gemeinsame Funktionalitäten oder Belange, die über mehrere Module oder Klassen verteilt sind, in separate Einheiten gekapselt. Aspect-Oriented Programming ermöglicht somit die Trennung von Aspekten von der Hauptgeschäftslogik, wodurch der Code besser organisiert und leichter wartbar wird. (vgl. Walls 2012, Kap. 4.1)

Zusätzlich umfasst das Framework eine Vielzahl an Modulen, die auf einem Basiscontainer aufbauen. Diese modulare Struktur erlaubt Entwicklern, nur die erforderlichen Funktionen auszuwählen und unnötige Module zu entfernen. Einige der zentralen Module sind:

1. Kerncontainer: Der Kerncontainer bildet das Fundament des Frameworks und beinhaltet unverzichtbare Komponenten wie Beans, Core, Context und Expression Language. In ihm werden die zentralen Funktionen wie Bean-Lebenszyklus-Management, Dependency Injection und AOP-Integration bereitgestellt.
2. Datenzugriff/Integration: Diese Schicht beinhaltet Module wie JDBC, JMS und Transactions, welche den Zugriff auf Datenbanken und Messaging-Dienste erleichtern.
3. Web-Schicht: Die Web-Schicht umfasst die Module Web, Web-MVC, Web-Websocket und Web-Reactive, die Funktionen zur Entwicklung von Webanwendungen und zur Verarbeitung von HTTP-Anfragen bereitstellen.
4. AOP, Aspekte und Instrumentierung: Diese Module unterstützen die aspektorientierte Programmierung und bieten Funktionen zur Instrumentierung, um die Architektur von Anwendungen zu verbessern.
5. Sicherheit: Die Sicherheitsmodule liefern umfängliche Sicherheitsdienstleistungen für Java Anwendungen

Die flexible Entwicklungsmöglichkeit bei Spring wird durch seine modulare Architektur ermöglicht. Entwickler können die benötigten Funktionen auswählen und nicht benötigte Module eliminieren. Durch die Unterteilung in verschiedene Module können Entwickler ihre Aufmerksamkeit auf die spezifischen Anforderungen ihrer Projekte konzentrieren, da das Framework die zugrunde liegende Infrastrukturverwaltung übernimmt.

3.5.2 Spring Web MVC

Spring MVC ist ein Framework für die Entwicklung von Web-Anwendungen auf Java-Basis. Es wird auch als Spring Web MVC bezeichnet und implementiert das Design-Muster Model-View-Controller (MVC). Dadurch wird eine Trennung zwischen Datenmodell, Präsentationsinformationen und Steuerungsinformationen ermöglicht. Spring MVC verwendet zudem alle grundlegenden Funktionen des Core Spring Frameworks, wie beispielsweise die Inversion of Control und Dependency Injection. (vgl. Paraschiv 2018)

Wie die Abbildung 16 Spring MVC Dispatcher Servlet veranschaulicht, ist das Framework ist um ein DispatcherServlet herum strukturiert. Diese Komponente empfängt alle eingehenden HTTP-Anfragen und leitet sie an die zugehörigen Controller weiter. Dazu werden zunächst über das Handler-Mapping die entsprechenden Controller für eine HTTP-Anfrage ermittelt. Die Einträge des Handler-Mappings wurden zu Beginn des Spring MVC Frameworks aus einer XML-Datei abgerufen und an den entsprechenden Controller weitergeleitet. Der Controller gibt eine ModelAndView-Instanz zurück, welche vom DispatcherServlet ausgewertet wird. Die aufzurufende View-Komponente wird durch den Eintrag des View-Resolvers in der XML-Datei bestimmt. In der heutigen Zeit gibt es modernere Alternativen zur XML-basierten Konfiguration, wie beispielsweise die Annotations-basierte oder Java-basierte Konfiguration (vgl. Walls 2012, Abschn. 7.1.1).

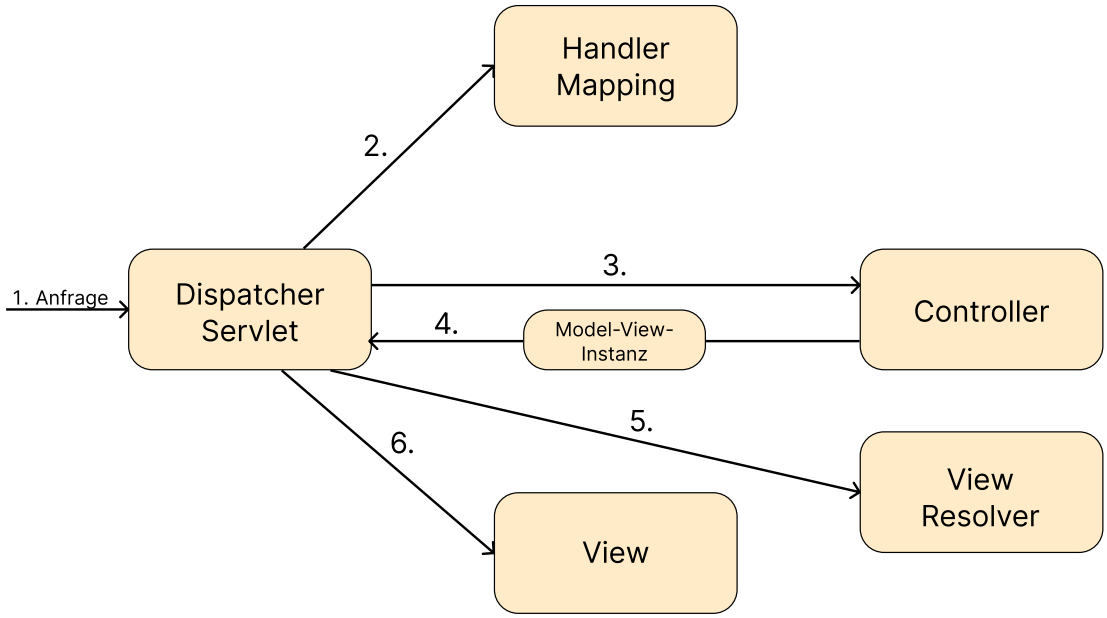

Abbildung 16 Spring MVC Dispatcher Servlet (vgl. Walls 2012, Abb. 7.1)

Mit der Einführung der Servlet API erfolgt die traditionelle Verarbeitung von Anfragen in Spring MVC erfolgt synchron. Dabei wird für jede Anfrage ein separater Java-Thread erstellt. Spring MVC nutzt somit das Thread-Per-Request-Modell, welches in einem vorherigen Abschnitt erläutert wurde. (vgl. Deinum, Rubio, und Long 2023, Kap. 4) Die Anfragen werden vom DispatcherServlet empfangen und an die entsprechenden Controller weitergeleitet. Diese Controller bearbeiten die Anfragen, interagieren mit dem Datenmodell und geben die Antwortdaten an die View-Komponente weiter. Die View-Komponente stellt diese Daten dann dar und sendet sie an den Client zurück. (vgl. Johnson u. a. 2023)

3.5.3 Spring WebFlux

Spring WebFlux ist ein Web-Framework für den reaktiven Stack von Spring, das eine vollständig nicht-blockierende funktionale Grundlage bietet und reaktive Datenströme unterstützt. Es wurde speziell dafür entwickelt, auf Servern wie Netty, Undertow und Servlet-Containern zu laufen. Spring WebFlux bietet ähnliche Vorteile im Ausführungsmodell wie andere nicht-blockierende Web-Stacks. Der Hauptantrieb für die Entwicklung von Spring WebFlux war der Bedarf nach einem nicht-blockierenden Web-Stack, um die Konkurrenz von modernen Webanwendungen mit wenigen Threads zu bewältigen und die Skalierbarkeit mit weniger Hardware-Ressourcen zu verbessern. Spring WebFlux basiert auf dem Projekt Reactor und implementiert die Spezifikation der reaktiven Datenströme. Es bietet Unterstützung für zwei Programmiermodelle: Annotation-basierte reaktive Komponenten und funktionales Routing und Handling. (vgl. Deinum, Rubio, und Long 2023, Kap. 4)

Wie bereits zuvor erwähnt, handelt es sich bei der reaktiven Programmierung um ein Programmierparadigma, das hauptsächlich auf nicht-blockierende Datenströme setzt, um die Kommunikation zu gewährleisten. Durch dieses Modell wird der Code reaktiv und reagiert auf Ereignisse

oder Änderungen in den Daten, anstatt blockierende Aufrufe zu verwenden. Spring WebFlux integriert dieses Modell durch die Implementierung der reaktiven Stream Spezifikation (Christensen und Kuhn 2015). Es bietet Unterstützung für reaktive Typen wie Mono und Flux, die die Grundbausteine für die Erstellung reaktiver Anwendungen in Spring WebFlux darstellen. Bei Abfragen, die höchstens ein Ergebnis liefern, gibt Mono einen Stream zurück, der null oder ein Element enthält. Flux hingegen gibt einen Stream zurück, der null oder mehr Elemente enthält, was bei Abfragen, die mehrere Ergebnisse liefern, hilfreich ist. (vgl. Deinum, Rubio, und Long 2023, tbl. 4.3)

3.5.4 Projekt Loom und virtuelle Threads

In dem Abschnitt zum Projekt Loom wurde bereits erläutert, dass virtuelle Threads eine Hauptfunktion darstellen, die durch dieses Projekt eingeführt wurde. Sie ermöglichen eine benutzerfreundliche, durchsatzstarke und leichtgewichtige Nebenläufigkeit in Java und sind besonders nützlich für die effiziente Ausführung von Anwendungen, die viele Anfragen nebenläufig verarbeiten.

Entwicklern wird mit der Veröffentlichung der Version 6 des Spring Frameworks im September 2023 die Möglichkeit bereitgestellt, die Funktionalität virtueller Threads zu testen, indem sie eine Spring-Boot-Anwendung von Grund auf mit virtuellen Threads erstellen können (vgl. Piazzolla 2023). Dies kann die Verarbeitung von Anfragen in Spring MVC grundlegend beeinflussen. Virtuelle Threads können als Worker-Threads genutzt werden, um die Verarbeitung von Anfragen effektiver zu gestalten. Mithilfe von virtuellen Threads können Entwickler eine große Anzahl gleichzeitiger Anfragen effizienter verarbeiten. Virtuelle Threads erfordern weniger Ressourcen als herkömmliche Threads und bieten somit eine leichtgewichtige und hochdurchsatzfähige Nebenläufigkeit. Dies ist besonders nützlich, um viele Anfragen gleichzeitig zu bearbeiten. Insgesamt bietet die Integration virtueller Threads aus dem Projekt Loom in Spring MVC eine vielversprechende Möglichkeit, die Effizienz und Skalierbarkeit von Spring-basierten Anwendungen zu verbessern, insbesondere unter hoher Last.

3.5.5 Zusammenfassung

Im Abschnitt über das Spring Framework wurde eine gründliche Analyse seiner Architektur und Kernkomponenten vorgestellt, die es zu einer wichtigen Grundlage in der Java-Entwicklung macht. Die modulare Architektur und die Prinzipien der Dependency Injection und Aspect-Oriented Programming sind von grundlegender Bedeutung, um die Komplexität zu reduzieren und eine flexible Entwicklung zu fördern.

Ebenfalls beschrieben wurden Spring MVC und Spring WebFlux, zwei zentrale Ansätze zur Verarbeitung von Anfragen. Während Spring MVC das traditionelle, synchrone Modell der Anfragenverarbeitung auf Java-Basis verfolgt, bietet Spring WebFlux eine reaktive, nicht-blockierende Grundlage für Anwendungen, die auf hohe Konkurrenz und Skalierbarkeit abzielen.

Die Vorstellung von Projekt Loom und virtuellen Threads eröffnet neue Möglichkeiten zur Verbesserung der Anfragenverarbeitung in Spring-basierten Anwendungen. Die Integration virtueller

Threads in Spring MVC verspricht eine Verbesserung der Effizienz und Skalierbarkeit, insbesondere unter hoher Last.

Mit einem guten Verständnis des Spring Frameworks, seiner Architektur und den verschiedenen Ansätzen zur Verarbeitung von Anfragen sind wird in dem nächsten großen Kapitel die Methodik des Experiments dargelegt. Ziel ist es, die Effizienz und Skalierbarkeit unterschiedlicher Ansätze zur Anfragenverarbeitung in Spring-basierten Anwendungen zu evaluieren. Der Fokus liegt dabei auf einem Vergleich zwischen Spring MVC und herkömmlichen Java-Threads, Spring MVC und den virtuellen Threads des Loom-Projekts sowie Spring WebFlux, welches reaktive Programmierung einsetzt. Die gewonnenen Erkenntnisse sollen einen klaren Überblick über Vor- und Nachteile dieser Ansätze geben und dabei unterstützen, fundierte Entscheidungen in der Praxis zu treffen. Die Methodik erläutert ausführlich die experimentelle Designstruktur, die Auswahl der Messinstrumente und die Bewertungskriterien, um ein umfassendes Verständnis der geplanten Untersuchung zu gewährleisten.

4 Methodik

Die Wahl der Methodik in wissenschaftlichen Arbeiten ist von entscheidender Bedeutung, um valide und zuverlässige Ergebnisse zu gewährleisten. In dieser Masterarbeit werden Effizienz und Skalierbarkeit verschiedener Ansätze zur Handhabung und Nebenläufigkeit von Anfragen in Spring-basierten Anwendungen untersucht. Es ist daher wichtig, eine klare und reproduzierbare experimentelle Vorgehensweise zu etablieren. Das vorliegende Kapitel zielt darauf ab, diese Methodik ausführlich darzustellen.

Zunächst wird das Design des Experiments beschrieben, welches die Zielsetzungen, Hypothesen und erwarteten Ergebnisse umfasst. Anschließend folgt eine detaillierte Vorstellung der Systemarchitektur der entwickelten Anwendungen. Die im Rahmen des Experiments auszuführenden spezifischen Aufgaben werden definiert und es wird die konkrete Testumgebung mit ihrer Konfiguration beschrieben. Um eine sinnvolle Analyse und Interpretation der gewonnenen Daten durchführen zu können, wird abschließend die Methodik der Datenerhebung und -auswertung beschrieben.

Das Verständnis dieser Methodik ist nicht nur für die Reproduzierbarkeit des Experiments von essenzieller Bedeutung, sondern auch, um die im weiteren Verlauf dieser Arbeit präsentierten Ergebnisse im korrekten Kontext interpretieren und bewerten zu können.

4.1 Design des Experiments

Das Hauptziel dieses Experiments ist die Evaluation der Effizienz und Skalierbarkeit von drei verschiedenen Ansätzen zur Anfragenverarbeitung in Spring-basierten Anwendungen. Konkret werden Spring MVC mit traditionellen Java-Threads, Spring MVC mit den von Loom-Projekt eingeführten virtuellen Threads und Spring Webflux, das reaktive Programmierung verwendet, miteinander verglichen. Dieses Experiment zielt darauf ab, fundierte Erkenntnisse über die Vor- und Nachteile zu schaffen und klare Anwendungsbeispiele in der Praxis zu identifizieren. Folglich können mehrere Hypothesen abgeleitet werden.

Es wird vermutet, dass die Anwendung, die Spring MVC in Verbindung mit traditionellen Java-Threads verwendet, bei hohen Lasten zu einem erhöhten Ressourcenverbrauch neigt und möglicherweise Skalierungsprobleme aufweist. Im Vergleich dazu wird erwartet, dass die Anwendung unter Verwendung der virtuellen Threads des Loom-Projekts eine größere Anzahl von Anfragen mit einem insgesamt niedrigeren Ressourcenverbrauch verarbeiten kann. Spring Webflux, das auf reaktiver Programmierung basiert, wird vermutlich in der Lage sein, eine große Anzahl von simultanen Anfragen effizient zu bearbeiten. Allerdings könnten höhere Ressourcenverbräuche in bestimmten Szenarien auftreten, insbesondere bei CPU-intensiven Aufgaben.

Um diese Hypothesen zu bestätigen, wird davon ausgegangen, dass es während des Experiments signifikante Unterschiede im Ressourcenverbrauch und in den Antwortzeiten zwischen den drei Ansätzen geben wird. Es wird erwartet, dass alle Ansätze eine Grundlast ohne größere Probleme

bewältigen können, jedoch sollen sich ihre wahren Stärken und Schwächen insbesondere bei größerer Belastung zeigen. Die gewonnenen Daten werden wertvolle Erkenntnisse darüber liefern, welcher Ansatz für verschiedene Anwendungen am besten geeignet ist, sei es für datenbankintensive Operationen, CPU-intensive Aktivitäten oder eine Kombination aus beidem.

4.2 Systemarchitektur

In diesem Abschnitt wird die Systemarchitektur der drei unterschiedlichen Implementierungen präsentiert. Jede Implementierung verfolgt einen anderen Ansatz, um Anfragen zu verarbeiten und zu beantworten. Dabei unterscheiden sie sich in Bezug darauf, wie sie Threads oder reaktive Muster nutzen. Trotz dieser Unterschiede haben alle Implementierungen eine gemeinsame Grundlage in ihrer Architektur.

4.2.1 Allgemeine Architektur

Alle drei Systeme basieren auf einer ähnlichen Struktur, um sicherzustellen, dass die Unterschiede in den Ergebnissen hauptsächlich durch die Art der Anfrageverarbeitung und nicht durch grundlegend verschiedene Systemdesigns beeinflusst werden.

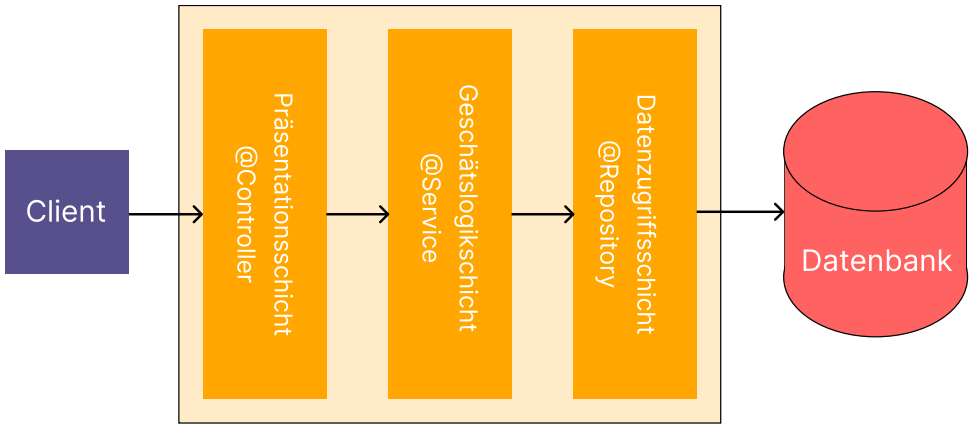

Abbildung 17 Allgemeine Architektur der Spring Anwendungen (Augsten und Koller 2021)

Wie aus der Abbildung 17 ersichtlich, sind die Hauptkomponenten die Präsentationsschicht, die Geschäftslogikschicht und die Datenzugriffsschicht. Die Präsentationsschicht, auch Controller genannt, empfängt Anfragen und gibt sie an die entsprechenden Dienste weiter, während in der Geschäftslogikschicht die eigentliche Logik ausgeführt wird. Die Datenzugriffsschicht interagiert mit der Datenbank, um Daten abzufragen oder zu speichern.

Ein Datenfluss beginnt beim Controller, durchläuft die Schicht der Geschäftslogik und endet in der Datenzugriffsschicht, bevor eine Antwort an den Client zurückgesendet wird.

4.2.2 Spring Web MVC mit nativen Threads

In dieser Implementierung wird Spring MVC in seiner traditionellsten Form genutzt. Wie die Abbildung 18 zeigt, wird jede eingehende Anfrage einem nativen Java Thread aus dem Thread Pool zugewiesen. Im Tomcat Webserver ist die Größe des Thread Pools standardmäßig auf 200 festgelegt (vgl. Apache Software Foundation 2023). Der zugewiesene Thread ist verantwortlich für die Verarbeitung der Anfrage durch die verschiedenen Schichten des Systems, beginnend beim Empfang durch das DispatcherServlet, das als zentrale Steuerungseinheit fungiert, bis hin zur Weiterleitung an den entsprechenden Controller. Der Controller führt dann die notwendige Geschäftslogik aus, interagiert mit dem Datenmodell, und bereitet eine Antwort vor, die an den Client zurückgesendet wird. Nachdem die Antwort gesendet wurde, wird der Thread beendet und die Ressourcen werden freigegeben. Diese Architektur ist besonders effektiv, wenn die Anfragen unabhängig voneinander sind und keine Blockierungen oder Verzögerungen verursachen, da jede Anfrage isoliert in einem separaten Thread behandelt wird.

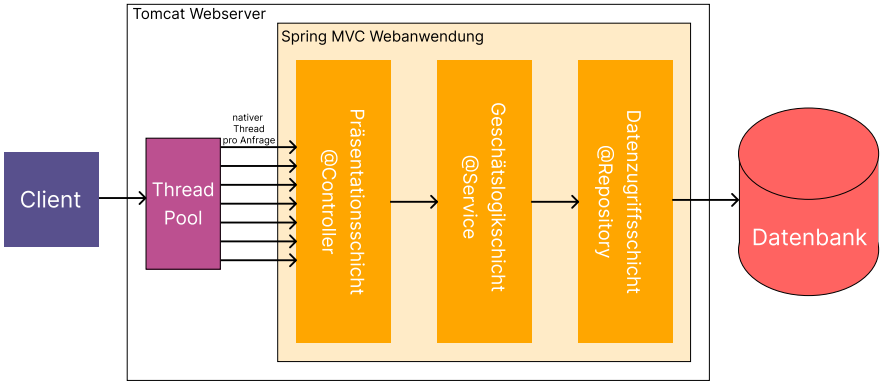

Abbildung 18 Architektur der Spring MVC Anwendung mit nativen Threads

4.2.3 Spring MVC mit virtuellen Threads

Während diese Implementierung auch auf Spring MVC aufbaut, wird sie durch die Verwendung von virtuellen Threads aus dem Projekt Loom erweitert. Das bedeutet, dass für jede eingehende Anfrage ein eigener virtueller Thread verwendet wird, wie in Abbildung 19 dargestellt.

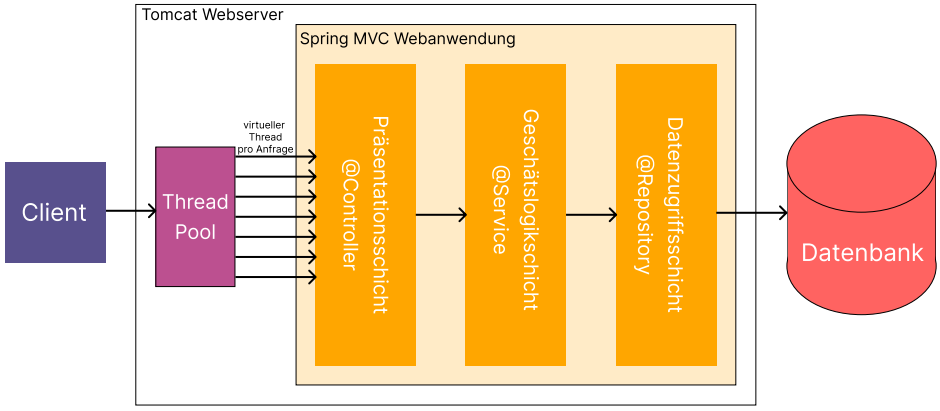

Abbildung 19 Architektur der Spring MVC Anwendung mit virtuellen Threads

Statt traditionellen nativen Threads werden virtuelle Threads eingesetzt, die ressourcenschonender sind und eine höhere Anzahl von nebenläufigen Anfragen ermöglichen. Die Implementierung gewährleistet somit eine effizientere Verarbeitung ohne einen erhöhten Verbrauch an Systemressourcen. Die Verwendung von virtuellen Threads wird durch den Austausch des normalen Tomcat ThreadExecutors durch einen neuen VirtualThreadPerTaskExecutor erreicht. Dies wird in Listing 8 veranschaulicht.

```
@EnableAsync
@Configuration
@ConditionalOnProperty(
        value = "spring.thread-executor",
        havingValue = "virtual"
)
public class ThreadConfiguration {

    @Bean
    public AsyncTaskExecutor applicationTaskExecutor() {
        return new TaskExecutorAdapter(Executors.newVirtualThreadPerTaskExecutor());
    }

    @Bean
    public TomcatProtocolHandlerCustomizer<?> protocolHandlerVirtualThreadExecutorCustomizer()
{
        return protocolHandler ->
protocolHandler.setExecutor(Executors.newVirtualThreadPerTaskExecutor());
    }
```

Listing 8 Austausch des ThreadExecutors für den Tomcat Webserver

4.2.4 Spring Webflux

Spring Webflux, das auf dem reaktiven Programmierparadigma basiert und standardmäßig Netty als Webserver verwendet, bietet eine Architektur, die auf die Verarbeitung von asynchronen und nicht-blockierenden Operationen ausgerichtet ist. Anstelle von dedizierten Threads für jede Anfrage, verwendet Webflux das Reactor-Framework, um asynchrone Datenflüsse zu verarbeiten, wodurch es besonders gut für Szenarien geeignet ist, in denen hohe Konkurrenz und Skalierbarkeit gefordert sind. Dabei wird, wie in der Abbildung 20 zu sehen ist, der von dem Netty-Webserver integrierte Event-Loop für einkommende Ereignisse verwendet. Dieser Event-Loop basiert auf einem einzelnen Thread und ist für die asynchrone Verarbeitung der Anfrage gedacht.

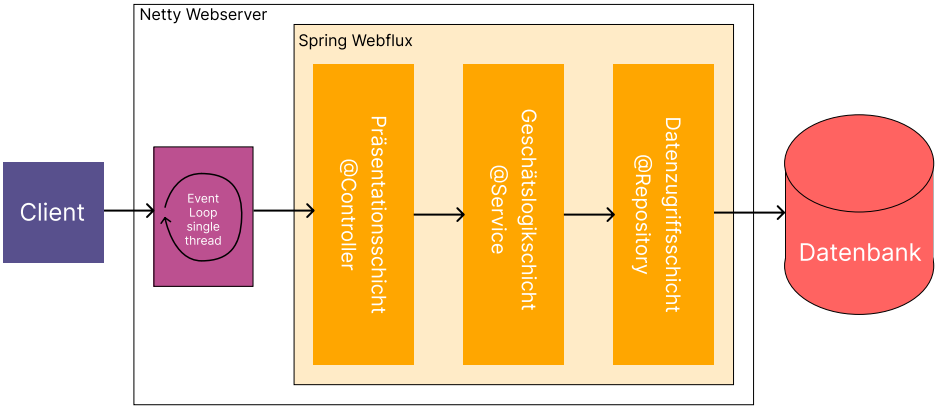

Abbildung 20 Architektur der Spring Webflux Anwendung

4.3 Aufgaben-Definition

In diesem Abschnitt werden die spezifischen Aufgaben definiert, die im Rahmen des Experiments durchgeführt werden. Das Ziel besteht darin, verschiedene Szenarien zu simulieren, die typische Anforderungen an moderne Webanwendungen abbilden. Es werden drei unterschiedliche Aufgabentypen vorgestellt: Eine datenbankzentrierte Aufgabe, eine CPU-intensive Aufgabe sowie eine kombinierte Aufgabe, die Elemente der beiden zuvor genannten Aufgabentypen integriert.

4.3.1 Datenbank-Aufgabe

Bei dieser Aufgabe geht es um Operationen, die die Interaktion mit einer Datenbank beinhalten. Konkret geht es darum, Daten von einer PostgreSQL-Datenbank abzurufen und auch in diese zu schreiben. Die API verarbeitet Anfragen zu einem spezifischen Nutzerprofil. Ein GET-Request ruft das Profil eines Nutzers anhand seiner Nutzer-ID ab, während bei einem POST-Request neue Nutzerdetails in die Datenbank eingefügt werden.

Datenbankanfragen sind typischerweise I/O-Operationen, die herkömmliche native Java-Threads blockieren können. Dies hat direkte Auswirkungen auf die Effizienz und Skalierbarkeit des Systems, insbesondere bei hoher Last. Der Fokus dieses Tasks liegt auf der Evaluierung der Effizienz und Performance unterschiedlicher Implementierungen bei der Arbeit mit einer Datenbank unter Berücksichtigung dieser blockierenden I/O-Operationen.

4.3.2 CPU-lastige Aufgabe

Bei diesem Task steht die Beanspruchung der CPU im Fokus, ohne dass eine I/O-Blockierung vorliegt. Die API beinhaltet einen Endpunkt, der für einen bestimmten Index Fibonacci-Zahlen berechnet. Da die Berechnung insbesondere für größere Indizes zunehmend komplex wird, gilt dies als rechenintensiver Vorgang. Ein weiteres Beispiel wäre das Durchsuchen großer Datenstrukturen, wie beispielsweise das Auffinden des kürzesten Pfades in einem Graphen. Für diese Untersuchung wird jedoch die Fibonacci-Berechnung angewendet.

4.3.3 Kombinierte Aufgabe

In der Praxis werden Webanwendungen oft mit einer Mischung aus datenbankzentrierten und CPU-lastigen Aufgaben konfrontiert. Es sollten daher Elemente beider Aufgaben integriert werden, um eine optimale Gesamtleistung und Effizienz zu gewährleisten. Ein mögliches Szenario könnte das Abrufen von Daten aus der Datenbank sein, gefolgt von einer aufwendigen CPU-Berechnung unter Verwendung dieser Daten. Dies ermöglicht eine realistische Bewertung der jeweiligen Implementierungen unter gemischten Arbeitsbelastungen.

Die API wird einen Endpunkt zur Verfügung stellen, der zunächst eine Datenbankanfrage auslöst, um eine Liste von Zahlen zu erhalten. Anschließend wird für jede dieser Zahlen die Fibonacci-Berechnung durchgeführt und das Ergebnis zurückgegeben. Dadurch wird ein Szenario simuliert,

in dem sowohl Datenbankinteraktionen als auch CPU-intensive Berechnungen benötigt werden, um eine Einschätzung der Leistungsfähigkeit in solchen kombinierten Szenarien zu ermöglichen.

4.4 Testumgebung und Konfiguration

Die Zuverlässigkeit und Gültigkeit von Experimenten hängen maßgeblich von der Konsistenz und Kontrolle der Testumgebung ab. Um verlässliche Ergebnisse zu gewährleisten, wurden spezifische Hardware- und Software-Konfigurationen, Docker-Einstellungen und Netzwerkkonfigurationen festgelegt und beibehalten.

4.4.1 Hardware- und Software-Setup

Das Experiment wird auf einem MacBook Pro 2023 mit einem M2-Max-Prozessor, welcher 12 CPU-Kerne bereitstellt, und 32 GB RAM durchgeführt. Das Setup gewährleistet ausreichend Ressourcen auch bei hoher Arbeitsbelastung. Als Betriebssystem wird MacOS 14.0 Sonoma verwendet und die Anwendungen werden unter OpenJDK 21 ausgeführt, um die Verwendung virtueller Threads zu ermöglichen. Die PostgreSQL-Datenbankversion 13 wird lokal auf dem MacBook gehostet, um Netzwerklatenzen auszuschließen.

4.4.2 Isolierung der Anwendungen

Für eine präzise Kontrolle und Isolierung der Anwendungsressourcen wird die Container-Software Docker verwendet. Laut einer Umfrage der Website Stack Overflow wurde diese Software von Tausenden von Entwicklern als wichtigste Plattform im Jahr 2022 nach Linux und Windows gewählt. Bei der Frage nach der *Most Wanted Platform* belegte Docker sogar den ersten Platz. (vgl. Öggl und Kofler 2021, 11; StackOverflow 2022)

Aufgrund seiner großen Popularität wird Docker für die Containerisierung im Rahmen des Experiments verwendet. Jeder Container wird so konfiguriert, dass er auf bestimmte Hardware-Ressourcen zugreifen kann. Es werden verschiedene Ressourcenkonfigurationen getestet, einschließlich Einschränkungen bei CPU-Kernen und RAM.

1. Basis-Konfiguration: Der Docker-Container wird mit einem CPU-Kern und 4 GB RAM ausgestattet.
2. Mittlere Konfiguration: Der Container erhält hierbei zwei CPU-Kerne und 8 GB RAM, was einem typischen Mittelklasse-Server oder einer Cloud-VM entspricht.
3. Hochleistungs-Konfiguration: Diese Konfiguration erlaubt dem Docker-Container die Nutzung von vier CPU-Kernen und 16 GB RAM und entspricht somit einer leistungsstarken Serverumgebung.

Durch die Ressourcenbeschränkungsfunktionen von Docker wird garantiert, dass kein Container mehr Ressourcen in Anspruch nimmt, als ihm zugewiesen wurden.

4.4.3 Netzwerkkonfiguration

Die Netzwerkkonfiguration wird so ausgelegt, dass alle Container auf demselben Docker-Netzwerk laufen, was die interne Kommunikation und die Verbindung zur Datenbank erleichtert. Unerwartete Verzögerungen in Experimenten werden so vermieden. Die Details zur Verbindung mit der PostgreSQL-Datenbank werden in den Umgebungsvariablen jedes Containers festgelegt, um die Konfiguration und den Zugriff zu vereinfachen.

4.5 Testdesign und Durchführung

Eine methodische und geordnete Testdurchführung ist entscheidend, um verlässliche und reproduzierbare Ergebnisse zu erzielen. In diesem Abschnitt wird beschrieben, wie die Tests mithilfe von Locust und Spring Boot Actuator gestaltet und durchgeführt werden.

4.5.1 Belastungstest der Anwendungen mit Locust

Locust ist ein oft genutztes Open-Source-Lasttest-Tool, das zur Bewertung der Leistung und Skalierbarkeit von Anwendungen verwendet wird. Zur Durchführung dieses Experiments wird ein Locust-Python-Skript auf einem anderen System ausgeführt, um eine Beeinträchtigung der Ressourcen des Testsystems durch die Ausführung des Skripts zu vermeiden.

Am Anfang des Tests wird eine Grundlast von 15 Benutzern simuliert, die gleichzeitig auf die Anwendungen zugreifen. Um eine Laststeigerung zu simulieren, wird die Anzahl der simulierten Benutzer jede Sekunde um 15 erhöht, bis 7500 Benutzer erreicht sind. Das System wird 10 Minuten lang getestet. Dabei werden sowohl die Grundlast als auch die Laststeigerung getestet.

4.5.2 Health-Monitoring der Anwendungen mit Spring Actuator

Der Spring Boot Actuator wird genutzt, um die Leistung der Anwendungen intern zu überwachen. Es werden Endpunkte für Metriken wie CPU- und Speichernutzung und durchschnittliche Antwortzeit konfiguriert. Diese Größen werden regelmäßig (alle 60 Sekunden) während des gesamten Tests abgerufen und für die spätere Analyse gespeichert.

4.5.3 Durchführungsplan

Das Experiment folgt einem systematischen Ansatz. Zunächst werden Tests in der Spring-MVC-Konfiguration mit normalen Java-Threads durchgeführt, dann in der Spring-MVC-Konfiguration mit virtuellen Threads und schließlich in Spring-Webflux, um den reaktiven Programmierungsansatz zu testen. So wird sichergestellt, dass die Ergebnisse nicht durch anfängliche Vorbehalte

beeinflusst werden. Jede Systemkonfiguration wird dann nacheinander für alle drei Docker-Konfigurationsstufen (Basis, Mittel und Hochleistung) getestet. Um sicherzustellen, dass die Ergebnisse zuverlässig sind, wird jeder Test dreimal durchgeführt und die Ergebnisse werden dann gemittelt, um Schwankungen auszugleichen. Auf diese Weise wird sichergestellt, dass die Ergebnisse sowohl umfassend als auch konsistent sind.

4.6 Datensammlung und Analyse

Von der Systematik und Genauigkeit der Datenerhebung und -auswertung hängen Qualität und Aussagekraft der Versuchsergebnisse in hohem Maße ab. In diesem Abschnitt werden die Metriken, die Analyse-Tools sowie die Interpretation der Daten beschrieben.

4.6.1 Metriken

In diesem Experiment werden gezielte Leistungsindikatoren herangezogen, die von wesentlicher Bedeutung sind, um die Performance und Effektivität der zu analysierenden Systeme zu bewerten.

1. Median der Antwortzeiten: Diese Kennzahl repräsentiert die mittlere Zeitdauer, die für die Bearbeitung und Beantwortung einer Anfrage durch das System erforderlich ist. Sie liefert Erkenntnisse darüber, wie zügig das System unter verschiedenen Lastszenarien reagiert. Konkret befinden sich 50% der Anfragen oberhalb und 50% unterhalb dieses Wertes.
2. Durchsatz: Dieser Indikator bezeichnet die Anzahl der Anfragen, die das System innerhalb eines festgelegten Zeitraums erfolgreich verarbeitet. Ein hoher Durchsatz signalisiert, dass das System in der Lage ist, eine große Anzahl von Benutzern simultan zu bedienen, ohne Einbußen in der Leistung zu erfahren.
3. CPU-Nutzung: Diese Metrik gibt Aufschluss über die Effizienz der Systemressourcennutzung im Hinblick auf die CPU. Der prozentuale Anteil der CPU-Auslastung, den die Anwendung im Betriebssystem beansprucht, wird hierbei betrachtet. Eine hohe CPU-Nutzung kann auf suboptimale Skalierung des Systems unter gewissen Bedingungen oder auf vorhandene Engpässe hindeuten, die einer Lösung bedürfen.
4. Speichernutzung: Die Analyse der Speichernutzung konzentriert sich auf den Eden-Space im Heap-Speicher der Java-Anwendung. Der Eden-Space ist der Bereich im Heap, der für die Allokation neuer Objekte verwendet wird. Bei voller Auslastung dieses Speichers entfernt der Garbage-Collector nicht mehr genutzte Objekte, während weiterhin genutzte Objekte in den Survivor Space verschoben werden. Die Untersuchung dieses Speicherbereichs ermöglicht eine umfassende Betrachtung der Speichernutzungsverhalten der Anwendung unter verschiedenen Lastbedingungen.

4.6.2 Tools und Techniken zur Datenauswertung

Die Analyse und Interpretation der gesammelten Daten erfolgt primär mittels der Programmiersprache Python und der Verwendung von Bibliotheken wie Pandas, NumPy und Matplotlib. Diese Instrumente bieten umfassende Unterstützung bei der Manipulation, Analyse und Visualisierung von Daten und eignen sich optimal für die Verarbeitung und Auswertung von komplexen Datensätzen.

Die CSV-Dateien, welche von den Performance-Testwerkzeugen Locust und Actuator/Prometheus erzeugt werden, werden in Pandas Dataframes eingelesen. Mit Pandas, einer Bibliothek, die besonders effektiv im Umgang mit tabellarischen Daten ist, können diese Daten strukturiert und flexibel gehandhabt werden. Komplexe Datenoperationen wie Filterung, Gruppierung und Aggregation lassen sich intuitiv durchführen.

Für weitere Analysen dieser Daten kommen NumPy und Matplotlib zum Einsatz. NumPy wird für die Durchführung mathematischer Operationen auf den Datensätzen verwendet, wie z.B. die Berechnung von Mittelwerten, Medianen und Standardabweichungen.

Die Ergebnisse werden durch Matplotlib visualisiert, um aussagekräftige Grafiken und Diagramme zu erstellen, die eine visuelle Interpretation und einen Vergleich der Leistungsdaten ermöglichen. Diese Visualisierungen unterstützen dabei, Trends und Muster in den Daten objektiv zu erkennen, die bei einer bloßen Betrachtung der Rohdaten möglicherweise nicht auffallen.

Die Verwendung dieser Tools ermöglicht insgesamt eine präzise und umfassende Analyse der erhobenen Daten. Die Kombination aus Pandas, NumPy und Matplotlib erweist sich als äußerst effektiv zur Verarbeitung, Analyse und Darstellung der umfangreichen Datensätze, die während der Performance-Tests generiert werden.

4.7 Limitationen und Herausforderungen

In diesem Abschnitt werden die Einschränkungen und Herausforderungen des durchgeführten Experiments dargelegt, um ein umfassendes Bild des Forschungsprozesses und seiner Ergebnisse zu vermitteln.

Eine potenzielle Einschränkung der Verallgemeinerbarkeit der Ergebnisse könnte sich ergeben haben, da die Experimente unter Laborbedingungen durchgeführt wurden. Obwohl die verwendete Hardware- und Software-Konfiguration repräsentativ für typische Anwendungsszenarien ist, sollte darauf hingewiesen werden, dass die Ergebnisse möglicherweise nicht alle realen Anwendungsumgebungen widerspiegeln. Diese eingeschränkte Übertragbarkeit der Ergebnisse auf andere Kontexte und somit die begrenzte Universalität der Forschungsergebnisse sollte berücksichtigt werden.

Der nächste Punkt betrifft die Repräsentativität der Daten. Die gewählten Testfälle und Daten sind so gestaltet, dass sie einige realistische Szenarien abdecken. Allerdings können sie nicht sämtliche potenzielle Anwendungsfälle in der Praxis widerspiegeln, was die Allgemeingültigkeit der Schlussfolgerungen einschränken könnte.

Des Weiteren haben die in der Studie genutzten Analysewerkzeuge und Testumgebungen ihre eigenen Limitierungen. Messungenauigkeiten oder Einschränkungen im Funktionsumfang der Werkzeuge können die Genauigkeit und Zuverlässigkeit der Ergebnisse beeinträchtigen.

Zudem ist es nicht garantiert, dass die Ergebnisse ohne Schwierigkeiten auf größere oder komplexere Systeme übertragen werden können. In größeren oder dynamischeren Umgebungen können Unterschiede in der Leistung und der Ressourcennutzung noch offensichtlicher werden.

Ein weiterer wichtiger Punkt ist die Berücksichtigung von Einschränkungen durch externe Faktoren. Externe Faktoren wie das Verhalten des Betriebssystems oder Einflüsse des Netzwerks können zu unkontrollierten Verzerrungen der Ergebnisse führen.

Zukünftige Untersuchungen sollten versuchen, die genannten Einschränkungen anzugehen. Dies könnte beispielsweise durch Experimente in realistischeren, vielseitigeren Umgebungen sowie durch die Anwendung von erweiterten Analysewerkzeugen oder zusätzlichen Metriken geschehen, welche unterschiedliche Aspekte der Systemleistung erfassen. Durch diese Empfehlungen soll die Robustheit und Relevanz zukünftiger Forschung in diesem Bereich gesteigert werden.

4.8 Zusammenfassung

Im Kapitel wird zunächst das Design des Experiments diskutiert, einschließlich der Ziele, Hypothesen und erwarteten Ergebnisse. Der Schwerpunkt liegt auf dem Vergleich der Effizienz und Skalierbarkeit von drei verschiedenen Ansätzen zur Anfragenverarbeitung in Spring-basierten Anwendungen: Spring MVC mit nativen Java-Threads, Spring MVC mit virtuellen Threads des Loom-Projekts und Spring WebFlux, das reaktive Programmierung nutzt.

Die Architektur der drei verschiedenen Implementierungen wird ausführlich vorgestellt und dabei werden die unterschiedlichen Ansätze zur Anfrageverarbeitung sowie ihre Auswirkungen auf die Systemleistung und -skalierbarkeit hervorgehoben.

Anschließend werden spezifische Aufgaben für das Experiment definiert, die durchgeführt werden. Diese Aufgaben beinhalten datenbankzentrierte, CPU-intensive und kombinierte Szenarien, um eine realistische Bewertung der einzelnen Implementierungen unter variablen Arbeitsbelastungen zu ermöglichen.

Die Testumgebung und -konfiguration werden exakt beschrieben, einschließlich der Hardware- und Software-Setups, der Containerisolierung der Anwendungen mittels Docker und der Netzwerkkonfiguration. Dadurch wird die Zuverlässigkeit und Gültigkeit der Experimente gewährleistet.

Das Testdesign und die Durchführung werden ausführlich beschrieben, einschließlich des Einsatzes von Locust für Belastungstests und Spring Boot Actuator zur Überwachung der Anwendungen.

Die Datensammlung und -analyse sind zentrale Bestandteile der Methodik und umfassen die Erfassung und Auswertung spezifischer Leistungsindikatoren wie Median der Antwortzeiten, Durch-

satz, CPU- und Speichernutzung. Die Auswertung und Interpretation der erhobenen Daten werden hauptsächlich unter Verwendung von Python mittels der Bibliotheken Pandas, NumPy und Matplotlib durchgeführt.

Abschließend werden im Rahmen des Experiments festgestellte Limitationen und Herausforderungen erörtert. Unter anderem werden die Generalisierbarkeit der Ergebnisse, die Repräsentativität der erhobenen Daten sowie die tool-spezifischen Limitationen, Skalierbarkeit, Variabilität und Stabilität der Ergebnisse, methodologische Herausforderungen und externe Faktoren betrachtet. Basierend darauf werden Empfehlungen für zukünftige Forschungen gegeben.

Die klare Darlegung dieser Methodik ist von großer Bedeutung für die Reproduzierbarkeit des Experiments und die objektive Auslegung und Beurteilung der erlangten Ergebnisse, die im Verlauf der Arbeit präsentiert werden.

5 Ergebnisse

In diesem Kapitel werden die Ergebnisse des durchgeführten Experiments präsentiert, die die Effizienz und Skalierbarkeit von verschiedenen Implementierungsansätzen in Spring-basierten Anwendungen untersucht haben. Diese Ansätze beinhalten Spring MVC mit nativen Java-Threads, Spring MVC mit virtuellen Threads des Loom-Projekts und Spring WebFlux, welches die reaktive Programmierung nutzt.

Das Experiment, welches im vorherigen Kapitel beschrieben wurde, fand unter kontrollierten Bedingungen statt, um eine umfassende Bewertung der Leistungsfähigkeit jedes Ansatzes zu ermöglichen. Die gesammelten Daten umfassen wichtige Leistungsindikatoren wie Antwortzeiten, Durchsatz sowie CPU- und Speichernutzung. In verschiedenen Szenarien wurden Metriken erhoben, welche datenbankzentrierte, CPU-intensive und kombinierte Arbeitslasten beinhalteten, um die Reaktion der verschiedenen Implementierungen auf unterschiedliche Anforderungen zu bewerten.

Das Ziel dieses Kapitels ist es, eine klare und objektive Darstellung der Ergebnisse zu liefern, ohne dabei in eine tiefgehende Analyse oder Interpretation der Daten einzusteigen.

Die Ergebnisse werden für jeden Aufgabentypen und jeder Konfiguration einzeln präsentiert, wobei besonderer Wert auf eine klare und verständliche Visualisierungen gelegt wird, um die Datensammlung gut vergleichbar und verständlich zu gestalten.

5.1 Ergebnisse der Datenbank-Aufgabe

In diesem Abschnitt werden die Ergebnisse einer umfassenden Untersuchung der Leistungsfähigkeit verschiedener Implementierungen von Web-Technologien vorgestellt, die im Kontext einer spezifischen Datenbankaufgabe getestet wurden. Ziel der Studie war es, ein vertieftes Verständnis der Leistungsmerkmale und der Effizienz der betrachteten Technologien unter verschiedenen Konfigurationen und Lastbedingungen zu entwickeln. Dabei wird sich auf die Veranschaulichung des Durchsatzes, der medianen Antwortzeit sowie der CPU- und Speichernutzung konzentriert.

Die Ergebnisse werden in drei Hauptteile gegliedert: niedrige, mittlere und hohe Konfiguration. Diese Unterteilung ermöglicht es, die Skalierbarkeit und Anpassbarkeit der verschiedenen Frameworks unter progressiv ansteigenden Lastszenarien zu bewerten. Die Untersuchung liefert wertvolle Einblicke in die Stärken und Grenzen jeder Technologie, die insbesondere für Softwareentwickler, Systemarchitekten und technische Entscheidungsträger von Interesse sind.

In den folgenden Abschnitten werden die gesammelten Daten detailliert vorgestellt und kurz diskutiert, um ein umfassendes Bild der Leistungsfähigkeit der einzelnen Implementierungen zu zeichnen.

5.1.1 Ergebnisse der niedrigen Konfiguration

Die Abbildung 21 illustriert die Messergebnisse der niedrigen Konfiguration in Bezug auf den Durchsatz, die mediane Antwortzeit sowie die Nutzung von CPU und Speicher. Hinsichtlich des Durchsatzes erreichten alle drei Anwendungen bei einer Nutzerzahl von bis zu 570 in der niedrigen Konfiguration einen ähnlichen Durchsatz von etwa 200 Anfragen pro Sekunde, wie aus dem linken oberen Diagramm ersichtlich wird. Bei einer höheren Nutzerzahl zeigten sich jedoch signifikante Unterschiede zwischen den verschiedenen Technologien. Während Spring MVC mit nativen Threads eine konstante Leistung aufwies, konnte eine Steigerung des Durchsatzes bei Spring MVC mit virtuellen Threads und Spring WebFlux festgestellt werden. Spring WebFlux erreichte bei einer Nutzerzahl von 1530 einen Durchsatz von etwa 650 Anfragen pro Sekunde. Spring MVC mit virtuellen Threads erreichte bei 3975 Nutzern einen Spitzenwert von 1537 Anfragen pro Sekunde. Danach schwankte die Leistung von der Implementierung mit virtuellen Threads bis zu 5235 Nutzern zwischen 1100 und 1537 Anfragen pro Sekunde. Ab Nutzerzahl 5235 sank die Durchsatzrate der Anwendung, während Spring WebFlux und Spring MVC mit nativen Threads relativ konstant blieben. Interessanterweise war der Durchsatz von Spring MVC mit virtuellen Threads bei einer Nutzerzahl von 6975 unter dem von Spring WebFlux, was auf eine signifikante Leistungsverschiebung hinweist, wenn die Nutzerzahlen hoch sind.

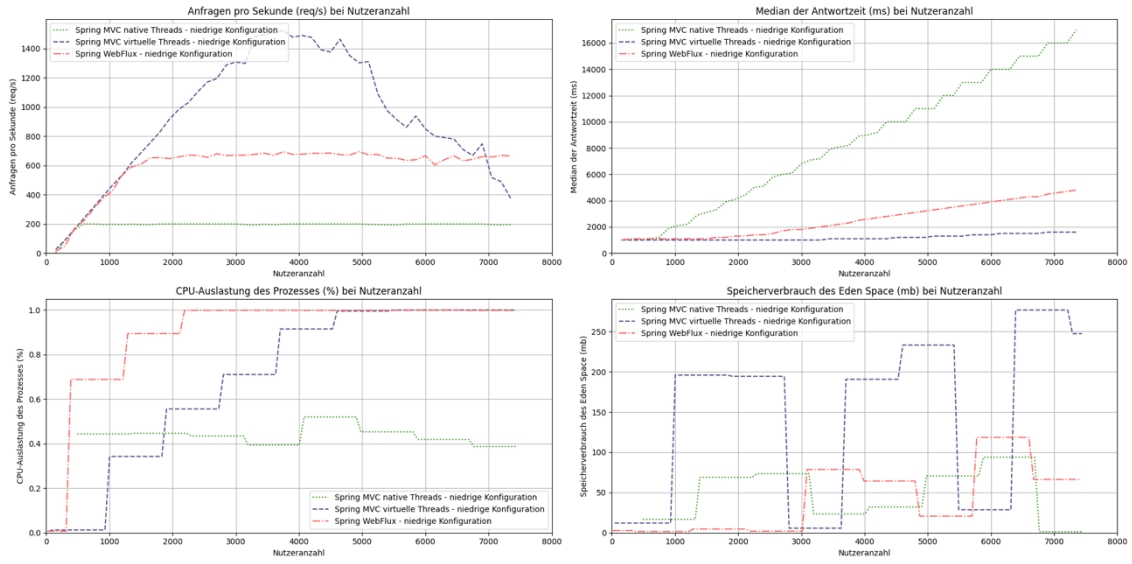

Abbildung 21 Ergebnisse der Datenbankaufgabe mit der niedrigen Konfiguration

In Bezug auf die mediane Antwortzeit blieben Spring WebFlux und die Implementierung mit virtuellen Threads bis zu 1530 Nutzern konstant bei einer Antwortzeit von etwa 1000ms. Danach stieg die mediane Antwortzeit von Spring WebFlux kontinuierlich an und erreichte bei 7500 Nutzern mit 6000ms ihren Höhepunkt. Die Implementierung mit virtuellen Threads zeigte im Gegensatz dazu bis etwa 4000 Nutzer eine bemerkenswerte Robustheit in der Antwortzeit mit leichten Schwankungen zwischen 1000ms und 1100ms. Erst danach stieg die Antwortzeit allmählich an und erreichte bei 7500 Nutzern eine mediane Antwortzeit von 1700ms. Diese Feststellung zeigt, dass die mediane Antwortzeit von WebFlux trotz der höheren Durchsatzfähigkeit der virtuellen Threads im Vergleich zu WebFlux signifikant höher ist. Dies deutet auf eine unterschiedliche Handhabung der Last zwischen beiden Technologien hin.

Die CPU-Auslastung veranschaulicht eine interessante Dynamik zwischen den verschiedenen Technologien. Bei WebFlux und den virtuellen Threads erreichte die CPU-Auslastung sogar 100%. Allerdings erreichte WebFlux diese maximale Belastung bei etwa 2190 Nutzern, während Spring MVC mit virtuellen Threads erst bei 4605 Nutzern an diese Grenze stieß. Im Gegensatz dazu wies die native Thread-Implementierung in Spring MVC eine moderatere CPU-Auslastung zwischen 35% und 55% auf. Das deutet darauf hin, dass trotz der höheren CPU-Auslastung von WebFlux und virtuellen Threads insbesondere WebFlux effektiver ist, um bei höheren Nutzerzahlen die Last zu bewältigen und einen konsistenten Durchsatz aufrechtzuerhalten. Dies zeigt sich in einer stabileren medianen Antwortzeit beider Ansätze. Bemerkenswert ist, dass sobald die CPU-Auslastung bei der Implementierung mit virtuellen Threads 100% erreicht, der Durchsatz stetig abnimmt und die mediane Antwortzeit ansteigt.

Beim Speicherverbrauch wiesen die virtuellen und nativen Threads anfangs einen initial größeren Eden Space von etwa 15 MB bis 20 MB auf, was auf einen höheren anfänglichen Speicherbedarf hindeutet.

Im Vergleich dazu benötigt Spring WebFlux nur 2,6 MB. Die Speichernutzung von Spring MVC mit virtuellen Threads schwankte in Intervallen zwischen 0 MB und 270 MB. Dies lässt vermuten, dass kontinuierliches Erstellen und Löschen von virtuellen Threads stattfinden, welches wiederum auf eine dynamische Anpassung an die Last und effizientes Speichermanagement durch den Garbage-Collector schließen lässt. Bei der nativen Thread-Implementierung hingegen zeigt sich eine stabile und gleichmäßige Nutzung des Speichers, was auf eine weniger dynamische, aber konstantere Handhabung des Speichers hindeutet.

5.1.2 Ergebnisse der mittleren Konfiguration

Abbildung 22 präsentiert die Ergebnisse der Messungen in Bezug auf Durchsatz, mittlere Antwortzeit sowie CPU- und Speicherauslastung für die mittlere Konfiguration.

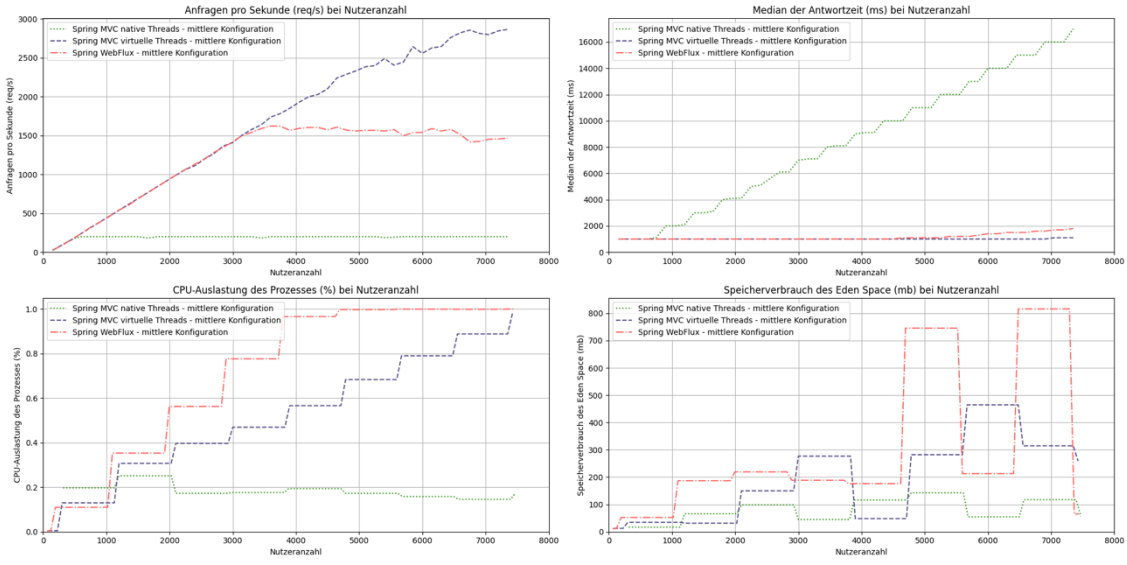

Abbildung 22 Ergebnisse der Datenbankaufgabe mit der mittleren Konfiguration

In der mittleren Konfiguration, ähnlich wie in der niedrigen Konfiguration, erreichten alle drei Anwendungen bis zu einer Nutzerzahl von 570 einen vergleichbaren Durchsatz von etwa 200 Anfragen pro Sekunde. Dies zeigt die anfängliche Konsistenz der Leistung über verschiedene Konfigurationen hinweg. Ab 570 Nutzern traten jedoch erneut signifikante Unterschiede zwischen den Technologien auf. Sowohl Spring MVC mit virtuellen Threads als auch Spring WebFlux erbringen eine deutlich höhere Leistung im Vergleich zur nativen Threads-Version hinsichtlich des Durchsatzes. Interessanterweise konnte Spring WebFlux in dieser Konfiguration eine hohe Leistung über einen längeren Zeitraum aufrechterhalten als in der niedrigeren Konfiguration. Hier wurde ein konstanter Durchsatz von rund 1500 Anfragen pro Sekunde erst ab 3135 Benutzern erreicht. Spring MVC mit virtuellen Threads erreichte bei 7500 Nutzern einen Höchstwert von 2843 Anfragen pro Sekunde, was die bessere Skalierbarkeit und Leistung dieser Technologie in der mittleren Konfiguration unterstreicht.

In Bezug auf die mediane Antwortzeit haben Spring WebFlux und die Implementierung mit virtuellen Threads bis zu einer Nutzerzahl von etwa 5280 konstante Antwortzeiten von 1000ms bis 1100ms gezeigt, was auf eine effiziente Bewältigung der steigenden Last hindeutet. Diese Ergebnisse stellen eine Verbesserung im Vergleich zur niedrigen Konfiguration dar, wo die Antwortzeiten von WebFlux bei höheren Nutzerzahlen stark anstiegen. Die mittlere Antwortzeit von WebFlux begann erst ab 5280 Nutzern zu steigen und erreichte kurz vor 7500 Nutzern einen Höchstwert von 1800ms. Im Gegensatz dazu zeigte Spring MVC mit virtuellen Threads durchweg eine beeindruckende Stabilität in der medianen Antwortzeit.

In Bezug auf die CPU-Auslastung ergab sich wieder ein Muster wie bei der niedrigen Konfiguration. Sowohl Spring WebFlux als auch Spring MVC mit virtuellen Threads wiesen eine zunehmende CPU-Last auf, die nahezu 100% erreichte. WebFlux erreichte diesen Punkt jedoch bereits bei einer Nutzerzahl von 4695, während Spring MVC mit virtuellen Threads die maximale CPU-Auslastung erst kurz vor 7500 Nutzern erreichte. Dies suggeriert, dass die Umsetzung mittels virtueller Threads in der mittleren Konfiguration eine effizientere Nutzung der CPU aufweist und erst bei hohen Nutzerzahlen an ihre Grenzen stößt.

Beim Speicherverbrauch zeigte sich bei Spring MVC mit virtuellen Threads in der mittleren Konfiguration ein gesteigerter Eden-Space von bis zu 470 MB im Vergleich zur niedrigen Konfiguration, was auf eine intensivere Nutzung der Ressourcen hinweist. Spring WebFlux hingegen wies beim Eden-Space einen signifikant höheren Verbrauch als die virtuellen Threads auf, mit einem Spitzenwert von 815 MB. Diese Beobachtung deutet darauf hin, dass unter bestimmten Bedingungen WebFlux mehr Speicher benötigt. Dies könnte auf eine intensivere Verarbeitung oder Speicherzuweisung hinweisen.

Insgesamt lassen diese Ergebnisse die Unterschiede in der Leistung und Effizienz der verschiedenen Implementierungen in der mittleren Konfiguration deutlich erkennen und bieten wichtige Erkenntnisse zum Verhalten der Technologien unter unterschiedlichen Bedingungen.

5.1.3 Ergebnisse der hohen Konfiguration

Die Abbildung 23 stellt die Ergebnisse der hohen Konfiguration dar, die den Durchsatz, die mediane Antwortzeit sowie die CPU- und Speichernutzung umfassen.

In der höchsten Konfiguration erzielten alle drei Anwendungen bis zu einer Nutzerzahl von 570 einen vergleichbaren Durchsatz von etwa 200 Anfragen pro Sekunde, ähnlich wie in den niedrigen und mittleren Konfigurationen. Diese Ergebnisse bestätigen die Konsistenz der Leistung unter geringerer Belastung und über verschiedene Konfigurationen hinweg. Allerdings zeigten sich oberhalb dieser Nutzerzahl wieder deutliche Leistungsunterschiede. Sowohl Spring MVC mit virtuellen Threads als auch Spring WebFlux zeigten eine bessere Leistung im Vergleich zur nativen Threads-Implementierung. Beachtenswert ist, dass sowohl Spring WebFlux als auch virtuelle Threads in dieser Konfiguration bis ca. 5805 Benutzer eine nahezu identische und stabile Performance erreichten, was eine Verbesserung gegenüber der mittleren Konfiguration darstellt, bei der WebFlux an seine Leistungsgrenzen stieß. Bei 7500 Nutzern erreichte Spring MVC mit virtuellen Threads einen Höhepunkt von 3433 Anfragen pro Sekunde, was die hervorragende Skalierbarkeit und Effizienz dieser Technologie unter anspruchsvollen Bedingungen unterstreicht. Die mediane Antwortzeit blieb für Spring WebFlux und virtuelle Threads bis zum Ende der Tests konstant niedrig, im Bereich von 1000ms bis 1100ms. Dies deutet auf eine deutliche Verbesserung im Vergleich zu den niedrigeren und mittleren Konfigurationen hin, bei denen WebFlux bei höheren Benutzerzahlen längere Antwortzeiten aufwies. Die Ergebnisse betonen die hohe Effizienz und Skalierbarkeit von Spring WebFlux und virtuellen Threads, insbesondere unter extremen Belastungsbedingungen.

In der höchsten Konfiguration ist eine ähnliche Tendenz wie in den vorherigen Konfigurationen erkennbar. Mit zunehmender Nutzerzahl stieg die CPU-Nutzung von WebFlux und virtuellen Threads. Allerdings erreichte WebFlux bei 7500 Nutzern eine maximale CPU-Auslastung von 90%, während virtuelle Threads eine niedrigere Auslastung von 51% aufwiesen. Im Vergleich zu den niedrigeren Konfigurationen wird die Last in der höheren Konfiguration effizienter verarbeitet, und die virtuellen Threads weisen eine bessere CPU-Nutzung auf.

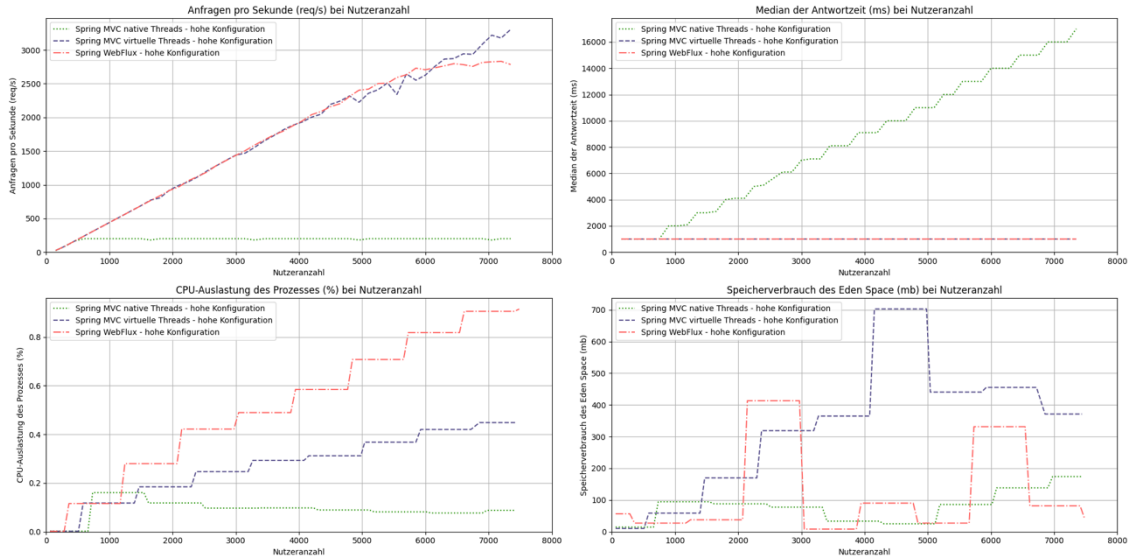

Abbildung 23 Ergebnisse der Datenbankaufgabe mit der hohen Konfiguration

In Bezug auf den Speicherverbrauch zeigten virtuelle Threads in der hohen Konfiguration einen höheren Eden-Space-Verbrauch als WebFlux, mit einem Maximalwert von 703 MB. Im Vergleich dazu erreichte WebFlux einen maximalen Eden-Space von 413 MB. Diese Beobachtung lässt vermuten, dass virtuelle Threads in der hohen Konfiguration eine flexiblere Speicherzuweisung oder

-nutzung aufweisen, während WebFlux eine effizientere Speicherverwaltung unter hohen Lastbedingungen zeigt.

Zusammenfassend zeigen diese Ergebnisse, dass die Leistung und Effizienz der verschiedenen Implementierungen in der hohen Konfiguration stark variieren und wichtige Erkenntnisse über das Verhalten der Technologien unter verschiedenen Bedingungen liefern.

5.2 Ergebnisse der CPU-Aufgabe

In diesem Abschnitt werden die Ergebnisse der CPU-Aufgabe vorgestellt, die Teil der umfassenden Untersuchung der Leistungsfähigkeit verschiedenen Implementierungsansätze sind. Der Fokus liegt dabei auf Spring MVC mit nativen Java-Threads, Spring MVC mit virtuellen Threads des Loom-Projekts und Spring WebFlux, die im Kontext CPU-intensiver Arbeitslasten analysiert werden.

Die Ergebnisse sind in niedrige, mittlere und hohe Konfigurationen unterteilt, um eine differenzierte Betrachtung der Leistungseigenschaften unter variierenden CPU-Lasten zu ermöglichen.

In den folgenden Abschnitten werden die spezifischen Ergebnisse für jede Konfiguration detailliert dargestellt. Diese strukturierte Darstellung liefert ein klares Bild der CPU-bezogenen Leistungsfähigkeit der untersuchten Technologien und ermöglicht einen direkten Vergleich ihrer Effizienz und Skalierbarkeit unter verschiedenen Bedingungen.

Durch diese genaue Analyse wird ein tieferes Verständnis der Stärken und Grenzen jeder Implementierung hinsichtlich CPU-intensiver Anforderungen erreicht, was für die Auswahl und Optimierung geeigneter Technologien in entsprechenden Anwendungsfällen entscheidend ist.

5.2.1 Ergebnisse der niedrigen Konfiguration

Wie die Abbildung 24 zusehen erreichte bei der niedrigen Konfiguration Spring MVC mit nativen Threads einen Durchsatz von 50 bis 75 Anfragen pro Sekunde bis zu einer Nutzerzahl von etwa 4000. Ab 4100 Nutzern nahm der Durchsatz jedoch interessanterweise sogar auf bis zu 115 Anfragen pro Sekunde zu. Allerdings ging diese Steigerung mit einem Anstieg der fehlgeschlagenen Anfragen pro Sekunde einher. Anfangs war dies nur ein geringer Prozentsatz, aber ab 6960 Nutzern erreichte die Fehlerquote bei den nativen Threads 100%. Dies bedeutet, dass alle Anfragen fehlschlugen. Im Gegensatz dazu zeigten sowohl virtuelle Threads als auch Spring WebFlux eine deutlich bessere Leistung. Beide Technologien erreichten frühzeitig einen Durchsatz von bis zu 175 Anfragen pro Sekunde und hielten ihn konstant, ohne dass Anfragen fehlschlugen bis zum Ende des Tests.

Diese Ergebnisse spiegeln sich auch in der medianen Antwortzeit wider. Während die Antwortzeiten der nativen Threads schnell anstiegen und am Ende des Tests bei etwa 59000ms lagen, zeigten virtuelle Threads und Spring WebFlux einen gleichmäßigeren Anstieg. Am Ende des Tests erreichten beide eine Antwortzeit von circa 23000ms, was auf ihre effizientere Handhabung der Anfragen unter Last hindeutet.

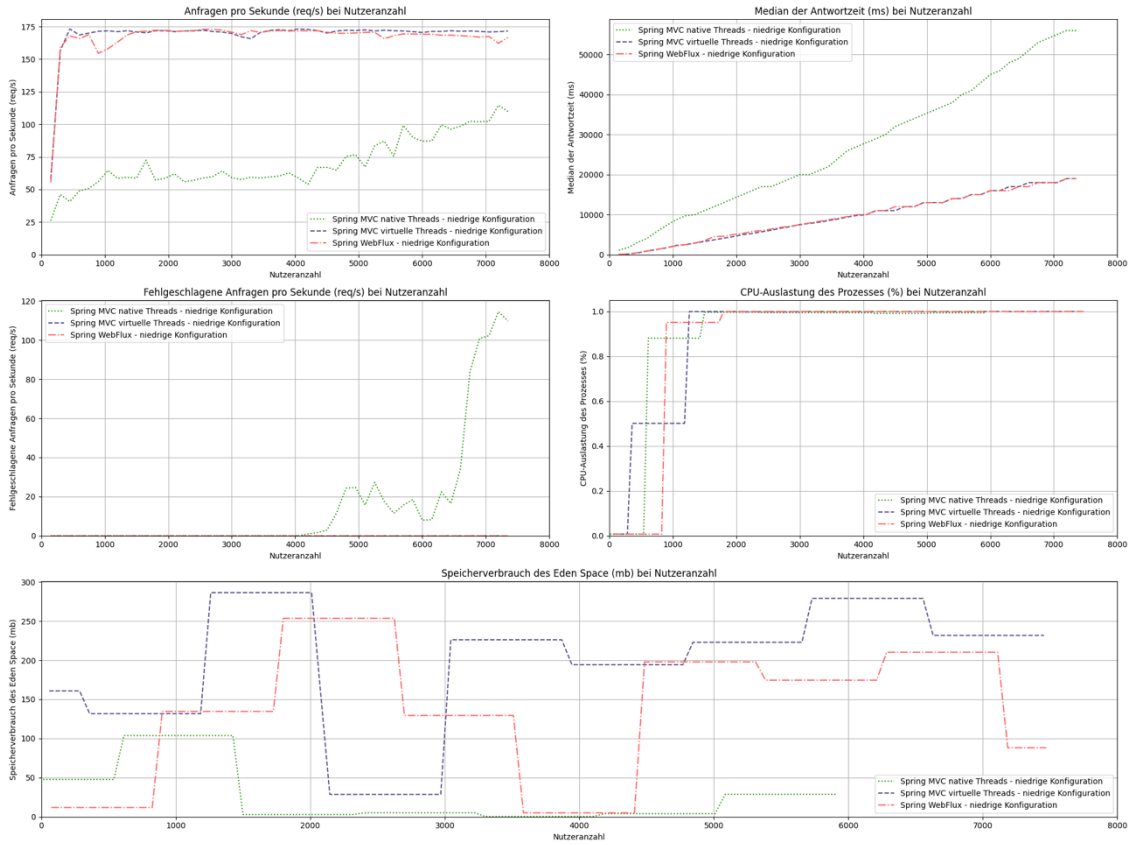

Abbildung 24 Ergebnisse der CPU-Aufgabe mit der niedrigen Konfiguration

In Bezug auf die CPU-Nutzung zeigten alle drei Implementierungen ein ähnliches Muster. Ab einer Nutzerzahl von etwa 1800 erreichte die CPU eine Auslastung von 100%. Dies deutet darauf hin, dass die CPU-Ressourcen bei allen drei Ansätzen vollständig ausgeschöpft wurden.

In Bezug auf den Speicherverbrauch beanspruchten die nativen Threads am wenigsten Ressourcen mit einem maximalen Verbrauch von etwa 100MB. Virtuelle Threads verbrauchten jedoch etwas mehr Speicher als Spring WebFlux, mit einem Spitzenwert von 286 MB. Spring WebFlux zeigte eine effiziente Speicherausnutzung mit einem maximalen Speicherverbrauch von 253 MB.

5.2.2 Ergebnisse der mittleren Konfiguration

Abbildung 23 zeigt die Ergebnisse der CPU-Aufgabe unter der mittleren Konfiguration. In dieser Konfiguration konnte eine Verbesserung der Leistung aller drei Anwendungen beobachtet werden, wobei interessante Unterschiede zwischen den Technologien festgestellt wurden.

Im Vergleich zur niedrigen Konfiguration war die Leistung aller Anwendungen besser. Spring MVC mit nativen Threads erreichte einen Durchsatz von 230 bis 280 Anfragen pro Sekunde. Virtuelle Threads und Spring WebFlux waren fast gleichauf. Allerdings schnitt Spring MVC mit virtuellen Threads etwas besser ab, da die Anwendung einen Durchsatz von bis zu 350 Anfragen pro Sekunde erreichte, während Spring WebFlux maximal 330 Anfragen pro Sekunde erzielte. Diese Steigerung im Vergleich zur niedrigen Konfiguration unterstreicht die verbesserte Effizienz der Technologien.

Die mediane Antwortzeit spiegelt diese Beobachtungen zum Durchsatz wider. Spring MVC mit nativen Threads hatte die schlechteste Leistung mit einer maximalen medianen Antwortzeit von 14000ms. Virtuelle Threads und Spring WebFlux verzeichneten nahezu identische Steigerungen in der medianen Antwortzeit bei zunehmender Anzahl von Nutzern. Bei 7500 Nutzern schnitt Spring WebFlux minimal schlechter ab als virtuelle Threads mit einer medianen Antwortzeit von 10000ms. Im Vergleich dazu lag die mediane Antwortzeit der virtuellen Threads bei 9600ms.

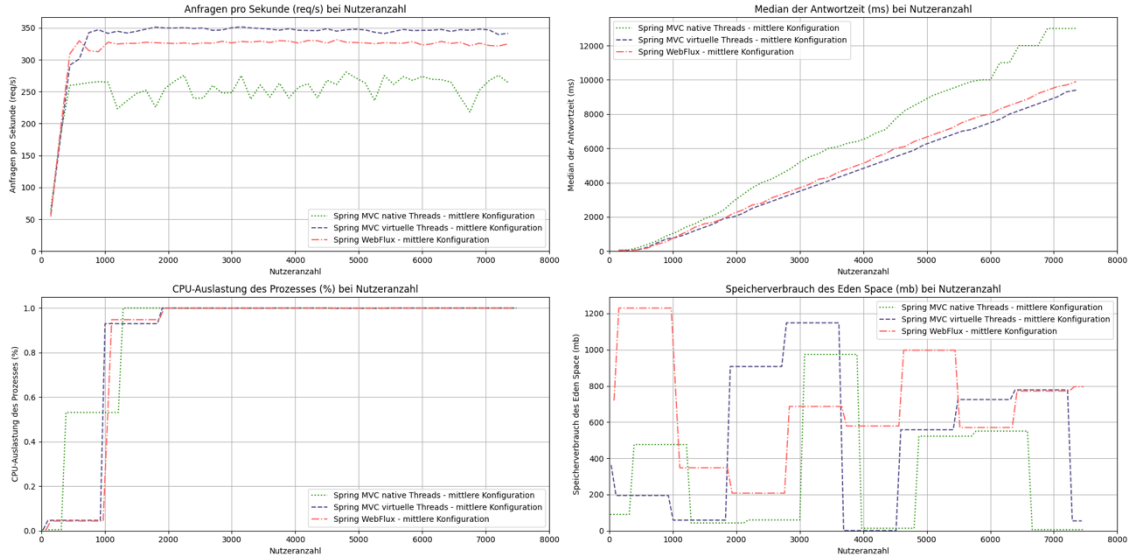

Abbildung 25 Ergebnisse der CPU-Aufgabe mit der mittleren Konfiguration

Alle drei Anwendungen nutzten in ähnlicher Weise wie in der niedrigen Konfiguration die CPU-Ressourcen vollständig aus. Bei einer Nutzerzahl von etwa 1900 erreichte die CPU-Auslastung 100% bei schlussendlich allen drei Anwendungen. Dies zeigt, dass die CPU-Kapazität unabhängig von der Implementierung bei mittlerer Last vollständig ausgelastet wird.

Bei der mittleren Konfiguration verbrauchten die nativen Threads am wenigsten Speicher, mit einem Höchstwert von 974 MB. Bei der erstgenannten Konfiguration war der Speicherverbrauch der nativen Threads ebenfalls am geringsten. Spring WebFlux startete mit einem Verbrauch von 1229 MB, fiel dann jedoch ab 1100 Nutzern unter 400 MB und stieg anschließend langsam wieder auf bis zu 1000 MB an. Die virtuellen Threads begannen knapp unter 400 MB, stiegen jedoch ab 1800 Nutzern auf über 900 MB an und erreichten ab 2800 Nutzern ihren Höchstwert von 1148 MB. Diese Beobachtungen deuten auf ähnliche Tendenzen wie bei der Datenbankaufgabe hin und zeigen konsistente Muster im Speicherverbrauch auf.

Die Ergebnisse der mittleren Konfiguration liefern weitere Einblicke in die Leistungsfähigkeit und Skalierbarkeit der verschiedenen Technologien. Besonders bemerkenswert sind die verbesserte Leistungsfähigkeit und Reaktionszeit im Vergleich zur niedrigen Konfiguration.

5.2.3 Ergebnisse der hohen Konfiguration

Die Abbildung 24 zeigt die Ergebnisse der CPU-Aufgabe unter der hohen Konfiguration des Docker-Containers. Es sind deutliche Leistungsunterschiede zwischen den verschiedenen Implementierungen zu erkennen, insbesondere nach Überschreiten einer bestimmten Nutzerzahl.

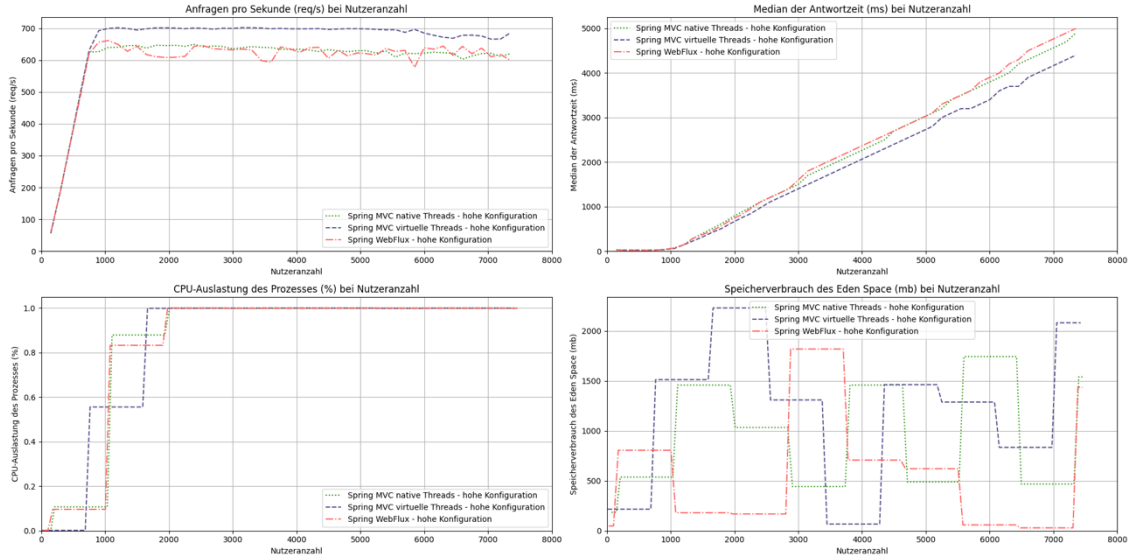

Abbildung 26 Ergebnisse der CPU-Aufgabe mit der hohen Konfiguration

Zu Beginn wiesen alle drei Anwendungen einen vergleichbaren Durchsatz bis zu einer Nutzerzahl von 780 auf. Ab diesem Punkt setzte sich Spring MVC mit virtuellen Threads leicht von Spring WebFlux und den nativen Threads ab. Während der Durchsatz von WebFlux und nativen Threads zwischen 565 und 645 Anfragen pro Sekunde schwankte, wobei WebFlux stärkere Schwankungen aufwies, erreichten die virtuellen Threads einen Spitzenwert von bis zu 700 Anfragen pro Sekunde und waren hierbei stabiler.

Diese Beobachtungen spiegelten sich auch in der medianen Antwortzeit wider. Alle drei Technologien wiesen einen ähnlich linearen Anstieg in der Antwortzeit auf. Am Ende des Tests wurde bei 7500 Nutzern eine mediane Antwortzeit von 4500ms für virtuelle Threads gemessen, was etwas niedriger ist als bei nativen Threads mit 5000ms und Spring WebFlux mit 5100ms.

Es gab keine Überraschungen in Bezug auf die CPU-Nutzung. Alle drei Implementierungen erreichten ab etwa 2000 Nutzern eine CPU-Auslastung von 100%, was eine vollständige Nutzung der CPU-Ressourcen signalisiert.

Bezüglich des Speicherverbrauchs zeigte sich in dieser Konfiguration, dass die virtuellen Threads am meisten Ressourcen nutzten, mit einem Höchstwert von 2229 MB und einem Tiefstwert von 67 MB. WebFlux verzeichnete einen maximalen Verbrauch von 1818 MB und einen minimalen von 29 MB. Interessanterweise verbrauchten die nativen Threads im Vergleich zu den anderen Konfigurationen deutlich mehr Speicher, mit einem Maximum von 1740 MB und einem Minimum von 442 MB.

Diese Ergebnisse der hohen Konfiguration liefern wichtige Erkenntnisse über die Leistungsfähigkeit und Skalierbarkeit der verschiedenen Technologien unter extremen CPU-Lastbedingungen.

Besonders hervorzuheben ist, dass die virtuellen Threads in dieser Konfiguration die beste Performance zeigten, sowohl im Hinblick auf den Durchsatz als auch auf die mediane Antwortzeit.

5.3 Ergebnisse der kombinierten Aufgabe

Dieser Abschnitt präsentiert die Ergebnisse einer kombinierten Aufgabe, die eine wichtige Komponente bei der umfassenden Bewertung der Leistungsfähigkeit von den Implementierungsansätzen auf Basis von Spring darstellt. Die Aufgabe kombiniert Elemente von datenbank- und CPU-lastigen Anforderungen, um ein realistischeres und anspruchsvolleres Szenario zu schaffen, das typisch für viele moderne Anwendungen ist.

Die Auswertung dieses Abschnitts fokussiert sich auf die Analyse und den Vergleich der Leistung von Spring MVC, nativen Java-Threads, Spring MVC mit virtuellen Threads des Loom-Projekts und Spring WebFlux unter kombinierter Belastung. Es wird ein ganzheitlicher Überblick über die Fähigkeit der verschiedenen Technologien geboten, effizient mit gemischten Anforderungen umzugehen, da datenbank- und CPU-intensive Operationen in einem einzigen Testumfeld integriert wurden.

Wie in den vorherigen Abschnitten erfolgt eine Unterteilung in niedrige, mittlere und hohe Konfigurationen, um die Leistungsfähigkeit der Technologien unter Bedingungen unterschiedlicher Skalierung detailliert zu untersuchen. Diese Struktur ermöglicht es, die Leistung jeder Implementierung nicht nur empirisch zu bewerten, sondern auch ihre Anpassungsfähigkeit und Zuverlässigkeit unter wechselnden und potenziell anspruchsvollen Betriebsbedingungen zu verstehen.

In den folgenden Abschnitten werden die Ergebnisse für jede Konfiguration detailliert dargestellt. Dabei wird besonderes Augenmerk auf wesentliche Metriken wie Durchsatz, mediane Antwortzeit, CPU- und Speicherauslastung gelegt. Durch die Verbindung dieser Metriken und der Erkenntnisse aus spezifischen Datenbank- und CPU-Aufgaben können umfassendere Schlussfolgerungen über die Eignung der verschiedenen Technologien für komplexe Anwendungsfälle gezogen werden.

Diese Ergebnisse sollen ein umfassenderes Bild der Leistungsfähigkeit und Effizienz der unterschiedlichen Implementierungen unter realen Bedingungen liefern.

5.3.1 Ergebnisse der niedrigen Konfiguration

Wie aus Abbildung 27 ersichtlich ist, erzielte die Anwendung mit virtuellen Threads und WebFlux bei geringer Konfiguration einen höheren Durchsatz als die Anwendung mit nativen Threads. Zwischen 1000 und 3000 Nutzern gab es ein maximales Durchschnitt von 37 Anfragen pro Sekunde. Im Gegensatz dazu erreichten die nativen Threads maximal 16,4 Anfragen pro Sekunde bei 1890 Nutzern. Ab einer höheren Nutzerzahl waren die Anfragen zunehmend nicht mehr erfolgreich.

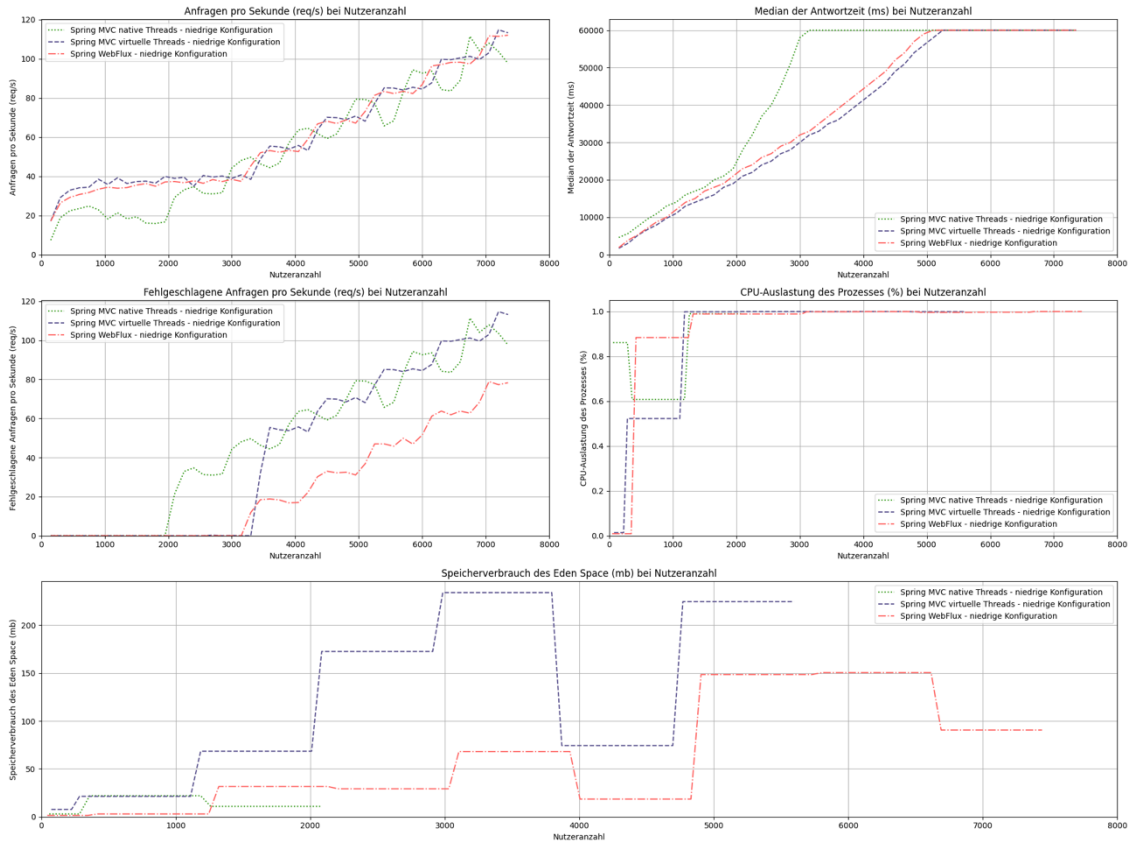

Abbildung 27 Ergebnisse der kombinierten Aufgabe mit der niedrigen Konfiguration

Interessanterweise waren bei den nativen Threads ab einer Nutzerzahl von 2000 keine CPU- und Speichermessungen mehr möglich, was auf eine vollständige Reaktionsunfähigkeit der Anwendung hindeutet. Im Gegensatz dazu blieben virtuelle Threads und WebFlux bis zu einer Nutzerzahl von 2865 fehlerfrei, jedoch zeigte sich bei den virtuellen Threads ab einer Nutzerzahl von 3795 eine Fehlerrate von 100%. WebFlux zeigte eine höhere Stabilität und erreichte erst bei 7500 Nutzern eine Fehlerrate von 71%.

In Bezug auf die durchschnittliche Antwortzeit zeigten native Threads eine deutlich höhere Geschwindigkeit als virtuelle Threads und WebFlux. Als etwa 5000 Nutzer erreicht wurden, erreichten native Threads die Timeout-Grenze von 60000 ms, während virtuelle Threads und WebFlux diesen Punkt erst bei einer höheren Anzahl von Nutzern erreichten. Die CPU-Auslastung erreichte in allen Implementierungen bei 1150 Nutzern 100%. Allerdings war die native Thread-Anwendung bald danach nicht mehr reaktionsfähig. Im Vergleich dazu blieb WebFlux bis zum Ende des Tests reaktionsfähig.

Der Speicherverbrauch war bei den nativen Threads am niedrigsten, mit einem Maximum von 21 MB. Nachdem 2000 Nutzer erreicht wurden, traten Probleme auf. Virtuelle Threads zeigten mit maximal 234 MB den höchsten Verbrauch, gefolgt von WebFlux mit maximal 152 MB.

5.3.2 Ergebnisse der mittleren Konfiguration

In der mittleren Konfiguration waren virtuelle Threads und WebFlux erneut erfolgreicher als native Threads, wie in Abbildung 28 zu sehen ist. Die nativen Threads erreichten eine Anfragenrate von 45 bis 80 pro Sekunde, während virtuelle Threads bis zu 84 Anfragen pro Sekunde und WebFlux konstant etwa 78 Anfragen pro Sekunde erreichten. Bei den nativen Threads stieg die Fehlerrate rapide an, ab 4440 Nutzern und ab 5175 Nutzern konnten alle Anfragen nicht mehr erfolgreich beantwortet werden. Virtuelle Threads haben erst ab 5160 Nutzern Fehler aufgewiesen und bei 5.310 Nutzern eine Fehlerrate von 100% erreicht.

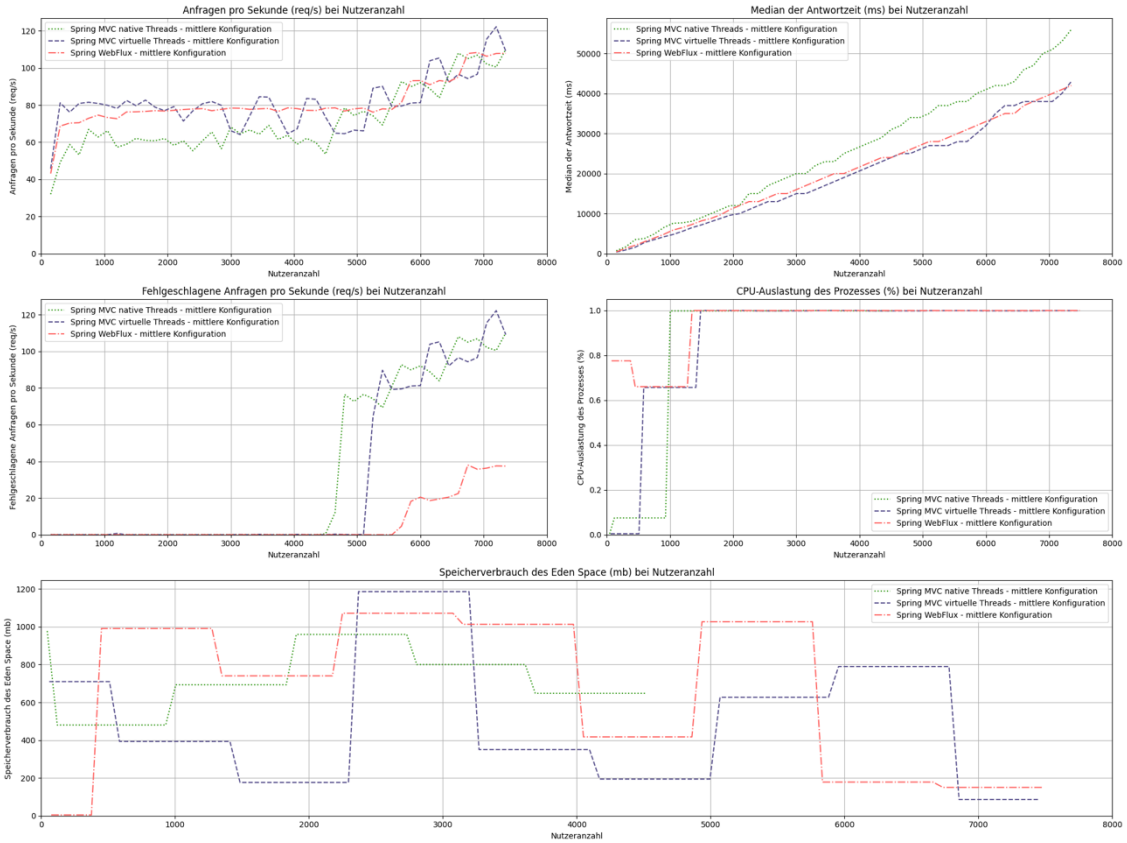

Abbildung 28 Ergebnisse der kombinierten Aufgabe mit der mittleren Konfiguration

Im Gegensatz dazu zeigte WebFlux eine längere Stabilität und erreichte erst bei 5.685 Nutzern eine steigende Fehlerrate, die bei 7500 Nutzern 34,5% betrug.

Die mediane Antwortzeit entwickelte sich bei virtuellen Threads und WebFlux fast identisch, mit einem Maximum von 43000ms für WebFlux und 59000ms für virtuelle Threads bei 7500 Nutzern. Die CPU-Auslastung erreichte frühzeitig bei allen Implementierungen 100%. Ab 4530 Nutzern reagierten native Threads nicht mehr, während virtuelle Threads und WebFlux bis zum Ende reaktionsfähig blieben.

Der Speicherverbrauch war bei den virtuellen Threads mit einem Maximum von 1185 MB am höchsten. WebFlux zeigte einen ähnlichen Verbrauch mit einem Maximum von 1050 MB, während native Threads zwischen 500 MB und 960 MB verbrauchten.

5.3.3 Ergebnisse der hohen Konfiguration

Unter der hohen Konfiguration wiesen alle drei Implementierungen einen ähnlichen Durchsatz auf, wie in Abbildung 29 dargestellt. Über den gesamten Nutzerbereich schwankte die Anzahl der Anfragen zwischen 130 und 165 Anfragen pro Sekunde. Alle Anfragen konnten erfolgreich beantwortet werden.

Die mediane Antwortzeit stieg bei allen drei Anwendungen fast gleichmäßig an und erreichte bei 7500 Nutzern ein Maximum von ca. 21000ms. Die CPU-Auslastung erreichte bei allen Implementierungen ab etwa 1500 Nutzern 100%, wobei die Anwendungen bis zum Ende reaktionsfähig blieben.

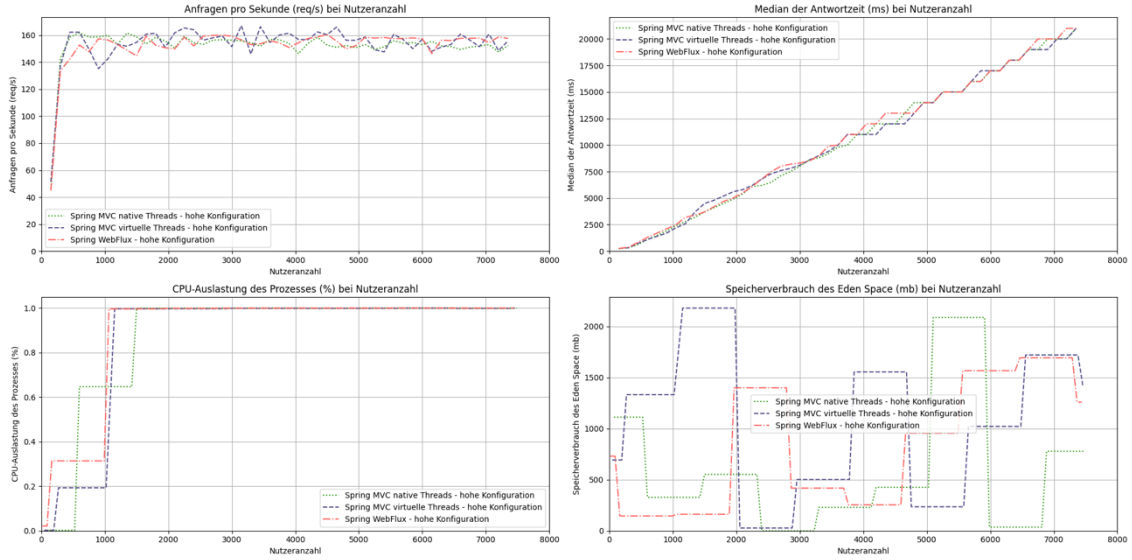

Abbildung 29 Ergebnisse der kombinierten Aufgabe mit der hohen Konfiguration

Der Speicherverbrauch war bei virtuellen Threads am höchsten mit bis zu 2178 MB, gefolgt von WebFlux mit maximal 1258 MB und nativen Threads mit bis zu 2086 MB. Alle drei Anwendungen zeigten starke Schwankungen im Speicherverbrauch.

Zusammenfassend liefern diese Ergebnisse wertvolle Einblicke in die Leistungsfähigkeit und das Verhalten von unterschiedlichen Technologien unter kombinierten Lastbedingungen. Sie bieten entscheidende Erkenntnisse für die Auswahl und Optimierung geeigneter Technologien in komplexen Anwendungsgebieten.

5.4 Zusammenfassung

In diesem Kapitel wurden umfangreiche Experimente durchgeführt, um die Effizienz und Skalierbarkeit verschiedener Implementierungsansätze in Spring-basierten Anwendungen zu bewerten. Die Analysen konzentrierten sich auf Spring MVC mit nativen Java-Threads, Spring MVC mit virtuellen Threads des Loom-Projekts und Spring WebFlux. Die Ergebnisse dieser Untersuchungen wurden in drei Hauptbereichen präsentiert: Datenbankaufgaben, CPU-intensive Aufgaben und kombinierte Aufgaben wurden jeweils in niedrige, mittlere und hohe Konfigurationen unterteilt.

Es wurde festgestellt, dass die Leistung der verschiedenen Technologien je nach Art der Aufgabe und Systemkonfiguration signifikant variiert. Obwohl einige Implementierungen in niedrigen Konfigurationen gut abschneiden, zeigen sie unter höheren Belastungen Schwächen.

In allen Szenarien wurden Durchsatz und Antwortzeiten als kritische Leistungsindikatoren identifiziert. Spring WebFlux und virtuelle Threads wiesen allgemein eine höhere Leistungsfähigkeit im Vergleich zu nativen Threads auf, insbesondere unter anspruchsvolleren Bedingungen.

Die CPU- und Speichernutzung sind entscheidende Faktoren, die die Leistungsfähigkeit verschiedener Technologien beeinflussen. Virtuelle Threads wurden oft umgesetzt und wiesen dabei eine höhere Speichernutzung auf, während WebFlux in einigen Szenarien ein effizienteres Handling des Arbeitsspeichers demonstrierte.

Die Stabilität und Zuverlässigkeit der verschiedenen Umsetzungen variierten unter höherer Belastung. In vielen Fällen hat sich WebFlux im Vergleich zu anderen Ansätzen als stabiler und zuverlässiger erwiesen.

Diese Erkenntnisse sind von entscheidender Bedeutung für die Analyse im nächsten Kapitel, in dem die praktische Anwendung dieser Technologien in realen Szenarien erörtert wird. Die gewonnenen Daten bieten eine fundierte Grundlage zur Beurteilung der Eignung jeder Technologie für spezifische Anwendungsfälle. Sie bieten wichtige Erkenntnisse für die notwendigen Überlegungen bei der Auswahl und Konfiguration von Technologien für effiziente und skalierbare Spring-basierte Anwendungen. In der folgenden Analyse werden diese Ergebnisse genutzt, um konkrete Empfehlungen für Softwareentwickler und Systemarchitekten zu formulieren, die darauf abzielen, die Leistung und Zuverlässigkeit ihrer Anwendungen in verschiedenen Betriebsumgebungen zu optimieren.

6 Analyse der Ergebnisse

In diesem Kapitel erfolgt die Auswertung der im vorherigen Abschnitt erhobenen Daten. Ziel ist es, die Bedeutung der Ergebnisse im Kontext der gesamten Arbeit zu verdeutlichen und ein tiefgreifendes Verständnis für die Leistung und Effizienz der verschiedenen Implementierungsansätze zu gewinnen. Das Kapitel gliedert sich in eine detaillierte Analyse der Ergebnisse, eine Diskussion zu ihren Implikationen sowie eine Reflexion hinsichtlich der Limitationen der vorliegenden Methodik. Abschließend wird ein Ausblick auf zukünftige Forschungsrichtungen gegeben, um das Gesamtbild der untersuchten Thematik abzurunden.

6.1 Detaillierte Analyse der Ergebnisse

Bei der Analyse von I/O-intensiven Aufgaben, insbesondere bei Datenbankaufgaben, erwiesen sich virtuelle Threads in verschiedenen Konfigurationen als am effizientesten. Dies äußert sich in einem höheren Durchsatz und einer besseren medianen Antwortzeit. Allerdings wurde bei virtuellen Threads in der niedrigsten Konfiguration eine deutliche Abnahme des Durchsatzes bei hohen Benutzerzahlen beobachtet, was auf mögliche Grenzen ihrer Skalierbarkeit hinweist. Im Vergleich dazu tendierte WebFlux dazu, sich bei einer CPU-Auslastung von etwa 80% auf einen bestimmten Durchsatz einzupendeln. Möglicherweise ist das auf die Backpressure-Mechanismen zurückzuführen, die eine Überlastung des Systems verhindern. Native Threads hingegen waren in allen Konfigurationen für I/O-intensive Aufgaben weniger geeignet.

Außerdem zeigen die Ergebnisse, dass WebFlux in den niedrigen und mittleren Konfigurationen sehr schnell eine CPU-Auslastung von 100% erreicht. Dies ist ein Hinweis darauf, dass die CPU-Ressourcen intensiv genutzt werden. Im Vergleich dazu verbrauchten die virtuellen Threads mehr CPU-Ressourcen als die nativen Threads, aber weniger als WebFlux, was auf eine ausgeglichenere Ressourcennutzung hindeutet. Dabei lässt die geringe CPU-Auslastung der nativen Threads darauf schließen, dass die Anwendung einen kleinen Thread-Pool verwendet, was wiederum auf eine ineffiziente Ressourcennutzung hinausläuft. Auffällig ist, dass sowohl Spring WebFlux als auch Spring MVC mit virtuellen Threads einen dynamischen Speicherbedarf zeigen. Dies spricht für eine effiziente Speichernutzung im Gegensatz zum geringeren, aber möglicherweise ungenutzten Speicherbedarf der nativen Threads. In diesem Zusammenhang ist anzumerken, dass virtuelle Threads und WebFlux zwar mehr Speicher benötigen, aber im Vergleich zu nativen Threads dadurch einen höheren Durchsatz und bessere Antwortzeiten bieten.

Hinsichtlich der CPU-intensiven Aufgabe zeigten Spring WebFlux und Spring MVC mit virtuellen Threads eine beeindruckende Leistung mit einem konstanten und hohen Durchsatz. Besonders deutlich wurde dies bei den mittleren und hohen Konfigurationen. Dies unterstreicht ihre Fähigkeit, CPU-intensive Aufgaben effizient zu bewältigen. Dabei weist die konstante Leistung der virtuellen Threads in verschiedenen Konfigurationen auf eine hohe Skalierbarkeit hin. Diese kon-

stante Leistung kann in Anwendungsszenarien mit schwankender CPU-Last von Vorteil sein. Darüber hinaus waren WebFlux und virtuelle Threads in der niedrigen und mittleren Konfiguration der Anwendung mit nativen Threads überlegen. In der hohen Konfiguration erreichte die Anwendung mit nativen Threads jedoch eine vergleichbare Leistung, was darauf hindeutet, dass native Threads für CPU-intensive Aufgaben besser skalieren, wenn ihnen mehr Ressourcen zugewiesen werden.

Die mediane Antwortzeit zeigt ein ähnliches Muster. Im Vergleich zu nativen Threads wiesen virtuelle Threads und WebFlux in der niedrigeren und mittleren Konfiguration eine deutlich bessere mediane Antwortzeit auf. Dies weist auf eine optimierte Nutzung von CPU-Ressourcen und effiziente Scheduling-Mechanismen hin. Zudem weisen virtuelle Threads und WebFlux in der hohen Konfiguration eine geringere mediane Antwortzeit auf und demonstrieren damit ihre Überlegenheit in Szenarien, in denen schnelle Antwortzeiten entscheidend sind. Die Tatsache, dass die CPU-Auslastung bei allen Implementierungen innerhalb kurzer Zeit 100% erreicht, weist eine vollständige Nutzung der verfügbaren CPU-Ressourcen hin, zeigt aber auch mögliche Engpässe bei der CPU-Leistung auf, die weiter untersucht werden sollten. Im Allgemeinen konnten WebFlux und virtuelle Threads besser mit der CPU-Last umgehen als native Threads. Der höhere Speicherverbrauch bei virtuellen Threads legt eine intensivere Nutzung der Speicherressourcen für die Verwaltung der Nebenläufigkeit nahe und sollte genauer untersucht werden, um mögliche Speicherengpässe zu identifizieren. Besonders bemerkenswert ist die Fähigkeit von virtuellen Threads und WebFlux, auch bei der kombinierten Aufgabe hohe Leistungen zu erbringen. Dies zeigt ihre Vielseitigkeit in komplexen Anwendungsszenarien. Allgemein weist WebFlux eine niedrigere Fehlerrate und höhere Stabilität auf. Daher eignet sich WebFlux besser für Anwendungen, bei denen die Zuverlässigkeit unter variablen Lastbedingungen entscheidend ist. Die Fähigkeit virtueller Threads und WebFlux, relativ kurze Antwortzeiten über einen großen Lastbereich aufrechtzuerhalten, unterstreicht ihre Effizienz. Diese Beobachtung ist von großer Bedeutung, da die Antwortzeiten ein Schlüsselfaktor für die Benutzererfahrung sind, insbesondere bei interaktiven Anwendungen. Die dynamische Speichernutzung in beiden Implementierungen impliziert eine adaptive Speicherverwaltung, die gegenüber nativen Threads für speichereffiziente Anwendungen vorteilhaft ist.

Zusammenfassend zeigen die Ergebnisse, dass virtuelle Threads und WebFlux insgesamt eine höhere Leistungsfähigkeit und Effizienz bieten als native Threads, vor allem in Szenarien mit variablen und hohen Lasten. Das legt nahe, dass diese Technologien besser für moderne, skalierbare Anwendungen geeignet sind, die eine hohe Leistung und Effizienz erfordern. Daraus lässt sich schließen, dass der Einsatz von virtuellen Threads die CPU-Auslastung und die Behandlung von I/O- und CPU-intensiven Tasks effizienter gestaltet. Daher sind sie besonders in Anwendungen mit hoher Nebenläufigkeit sinnvoll einsetzbar. WebFlux eignet sich besonders für Szenarien, die eine hohe Stabilität und effiziente Reaktion auf Backpressure erfordern, was für reaktive Systeme entscheidend ist. Die Analyse der Speichernutzung und die dynamische Anpassung bei beiden Technologien legen nahe, dass in zukünftigen Anwendungen ein Augenmerk auf die Speicheroptimierung gelegt werden sollte, um die volle Leistungsfähigkeit dieser Technologien zu nutzen. Es ist wichtig, den Nebenläufigkeitsansatz sorgfältig abhängig von den spezifischen Anforderungen der Anwendung und der zur Verfügung stehenden Hardwareressourcen zu wählen.

In zukünftigen Studien sollte dies weiter untersucht werden. Dadurch können detailliertere Richtlinien für die Auswahl der optimal geeigneten Technologie entwickelt werden.

Im nächsten Abschnitt werden die Forschungsfragen, die im Kapitel 1 gestellt wurden, beantwortet und bewertet. Hierbei werden die Ergebnisse dieser detaillierten Analyse berücksichtigt.

6.2 Bewertung der Forschungsfragen

Die in Kapitel 1 formulierten Forschungsfragen haben den Fokus der Untersuchung auf die Effizienz, Skalierbarkeit und Leistungsfähigkeit verschiedener Implementierungsansätze im Spring Framework gerichtet. Nach einer detaillierten Analyse der experimentellen Ergebnisse können nun diese Fragen beantwortet und die Hypothesen entsprechend bestätigt oder widerlegt werden.

1. Wie unterscheidet sich die Leistung von Spring MVC-Anwendungen hinsichtlich Antwortzeiten und Ressourcennutzung, wenn native Java-Threads im Vergleich zu virtuellen Threads des Loom-Projekts verwendet werden?

Die Untersuchungen zeigen, dass Spring MVC mit virtuellen Threads des Loom-Projekts in vielen Szenarien einen höheren Durchsatz und bessere mediane Antwortzeiten als die Implementierung mit nativen Threads erreicht. Dies ist besonders in Szenarien mit hohen Nutzerzahlen und in komplexen Aufgabenstellungen der Fall. Die Ergebnisse deuten darauf hin, dass virtuelle Threads insbesondere bei I/O-lastigen Aufgaben eine effizientere Ressourcennutzung und bessere Skalierbarkeit bieten. Damit wird die Hypothese bezüglich der Leistungsvorteile virtueller Threads in spezifischen Anwendungsszenarien durch diese Ergebnisse bestätigt. Obwohl Native Threads bei hohen Hardwarekonfigurationen in der Regel bei der gewählten CPU-lastigen Aufgaben mit virtuellen Threads mithalten können, bleiben sie allgemein dennoch im Nachteil.

2. Inwieweit verbessert die Anwendung reaktiver Programmierungsparadigmen mit Spring WebFlux die Skalierbarkeit und den Durchsatz von Webanwendungen unter variierenden Lastbedingungen im Vergleich zu synchronen Ansätzen?

Die Ergebnisse bestätigen, dass Spring WebFlux in vielen Fällen eine bessere Skalierbarkeit und höheren Durchsatz als native Threads bietet. Insbesondere unter variierenden Lastbedingungen zeigt Spring WebFlux eine überlegene Leistung. Das ist wahrscheinlich auf die effizientere Ressourcennutzung und die Fähigkeit, Backpressure zu handhaben, zurückzuführen. In den meisten Szenarien konnten virtuelle Threads jedoch mit dem reaktiven Programmierungsansatz mithalten. Die einzigen Leistungseinbußen wurden unter der niedrigen Hardwarekonfiguration verzeichnet, was auf fehlende Backpressure-Mechanismen zurückzuführen ist. Die Ergebnisse unterstützen die Hypothese, dass reaktive Ansätze in den meisten Szenarien den nativen Threads überlegen sind. Allerdings konnten virtuelle Threads in der Mehrzahl der Szenarien eine gleichwertige oder bessere Leistung erzielen.

3. Welche spezifischen Szenarien und Anwendungsfälle in der Java-basierten Webentwicklung profitieren am meisten von den virtuellen Threads, die durch das Projekt Loom eingeführt wurden, und wie können diese Erkenntnisse zur Optimierung bestehender Anwendungen eingesetzt werden?

Die Ergebnisse zeigen, dass virtuelle Threads in Szenarien mit hoher Nebenläufigkeit besonders effizient sind. Insbesondere bei I/O- und CPU-intensiven Aufgaben weisen sie im Vergleich zu nativen Threads eine verbesserte Leistung auf. Die Fähigkeit, schnell auf Laständerungen zu reagieren und dabei eine effiziente Speicher- und CPU-Nutzung zu gewährleisten, macht sie ideal für komplexe, skalierbare Anwendungen. Diese Erkenntnisse liefern wertvolle Informationen zur Optimierung bestehender Anwendungen unter Verwendung von virtuellen Threads. Vorsicht ist jedoch in Anwendungsumgebungen mit begrenzten Hardwareressourcen geboten. Wenn die Last zu hoch ist, kann die Anwendung möglicherweise nicht mehr alle Anfragen beantworten, da die Ressourcen des Systems aufgebraucht sind. Es ist empfehlenswert, entweder die Anzahl der virtuellen Threads über den gewählten Benutzer-Modus-Schedulers zu begrenzen oder auf leistungsstärkere Hardware umzusteigen.

Zusammenfassend liefern die vorliegenden Ergebnisse entscheidende Einblicke in die Leistungsfähigkeit und Effizienz der verschiedenen Implementierungsansätze im Spring Framework. Sie bestätigen, dass sowohl virtuelle Threads als auch reaktive Programmierungsparadigmen signifikante Vorteile in bestimmten Anwendungsszenarien bieten können. Die Wahl des geeigneten Ansatzes sollte daher sorgfältig abhängig von den spezifischen Anforderungen und Rahmenbedingungen jeder Anwendung getroffen werden.

6.3 Zusammenfassung

Die umfassende Analyse der experimentellen Ergebnisse hat tiefgreifende Einblicke in die Leistungsfähigkeit und Effizienz der verschiedenen Implementierungsansätze geliefert. Dieser Abschnitt fasst die zentralen Erkenntnisse zusammen und bildet eine Brücke zur Schlussfolgerung in dem nachfolgenden Kapitel. Die Ergebnisse zeigen deutlich, dass die Implementierung virtueller Threads aus dem Loom-Projekt in Spring MVC-Anwendungen zu einer signifikanten Verbesserung der Skalierbarkeit und Effizienz führt. Virtuelle Threads bieten eine ausgewogenere Ressourcennutzung und zeigen besonders bei hohen Nutzerzahlen ihre Stärken. Sie sind in der Lage, mit den Herausforderungen von den drei Aufgabentypen effizient umzugehen, was sie zu einer attraktiven Option für moderne Anwendungen macht. Obwohl Spring WebFlux seine Stärken in Szenarien mit variabler Last durch effektive Backpressure-Handhabung zeigt, können virtuelle Threads in vielen Fällen eine gleichwertige oder sogar bessere Leistung bieten. Dies stellt die Präferenz für reaktive Programmierung in bestimmten Anwendungsfällen in Frage.

Im anschließenden Kapitel werden diese Erkenntnisse weiter vertieft. Dort wird über die umfassenden Implikationen dieser Studie reflektiert, über ihre Limitationen diskutiert und praktische Empfehlungen sowie Anregungen für zukünftige Forschungsarbeiten abgeleitet. Dieser Abschnitt zielt darauf ab, einen zusammenhängenden Überblick über die gesamte Forschungsarbeit zu geben und die Bedeutung der Ergebnisse im Kontext der sich ständig weiterentwickelnden Landschaft der Softwareentwicklung zu betonen.

7 Schlussfolgerung

In dieser Arbeit wurden die Auswirkungen und Möglichkeiten von virtuellen Threads innerhalb des Spring-Frameworks eingehend untersucht und mit nativen Java-Threads und reaktiver Programmierung verglichen. Die Untersuchungen konzentrierten sich auf neue Wege zur Steigerung der Skalierbarkeit und Effizienz von Java-basierten Webanwendungen und lieferten aufschlussreiche Erkenntnisse.

Die wichtigsten Ergebnisse zeigen, dass die Integration der virtuellen Threads aus dem Loom-Projekt in Spring MVC-Anwendungen zu einer signifikanten Verbesserung der Skalierbarkeit und Effizienz führt. Diese Verbesserung wird erreicht, ohne dass die bestehende Anwendungsarchitektur grundlegend geändert werden muss, was für viele Entwickler eine einfache und kostengünstige Lösung darstellt. Die Studie hat auch gezeigt, dass virtuelle Threads eine leistungsfähige Alternative zu reaktiven Programmieransätzen bieten können.

Parallel dazu wurde die Rolle der reaktiven Programmierung unter den neuen Bedingungen von Projekt Loom untersucht. Es wurde festgestellt, dass reaktive Ansätze zwar in bestimmten Szenarien weiterhin ihre Berechtigung haben, die Einführung virtueller Threads jedoch den Bedarf an reaktiven Programmiermustern in vielen Anwendungsfällen reduzieren könnte. Dies stellt einen bemerkenswerten Wandel in der Java- Entwicklung dar und eröffnet neue Perspektiven für die Optimierung von Anwendungen.

In den folgenden Abschnitten werden die Bedeutung und die Implikationen dieser Ergebnisse eingehender erörtert, die Grenzen der durchgeführten Forschung diskutiert und Empfehlungen für die Praxis gegeben. Abschließend werden Vorschläge für zukünftige Forschungsarbeiten gemacht, die auf den gewonnenen Erkenntnissen aufbauen.

7.1 Bedeutung und Implikation

Die in dieser Arbeit vorgestellten Ergebnisse haben weitreichende Implikationen für die Entwicklung von Spring-Anwendungen sowie für den weiteren Bereich der Java-Entwicklung. Dabei verdienen zwei Hauptaspekte besondere Aufmerksamkeit.

Erstens zeigen die Untersuchungsergebnisse, dass eine Migration von Spring MVC-Anwendungen auf die virtuellen Threads des Loom-Projekts eine signifikante Verbesserung in Bezug auf Skalierbarkeit und Effizienz verspricht. Diese Verbesserung kann, wie die Ergebnisse nahelegen, ohne nennenswerten Aufwand realisiert werden. Die Implementierung virtueller Threads in bestehende Anwendungsstrukturen ermöglicht es, bestehende, imperativ programmierte Anwendungen zu optimieren, ohne die grundlegende Architektur zu verändern. Dies ist ein klarer Vorteil für Entwickler, die nach Möglichkeiten suchen, die Leistung ihrer Anwendungen zu verbessern, ohne sie komplett neu schreiben oder auf neue Programmierparadigmen umstellen zu müssen.

Ein weiterer wichtiger Aspekt, der sich aus den Ergebnissen ableiten lässt, ist die potenzielle Verringerung der Relevanz reaktiver Programmieransätze im Java-Umfeld durch die Einführung virtueller Threads. Die Studie hat gezeigt, dass virtuelle Threads in der Lage sind, imperativen Programmieransätzen eine mit reaktiven Ansätzen vergleichbare Skalierbarkeit und Effizienz zu verleihen. Dieses Ergebnis könnte die Präferenz für reaktive Programmierung in zukünftigen Java-Webanwendungen in Frage stellen, da virtuelle Threads eine ähnliche Leistung bieten, jedoch ohne die Notwendigkeit, sich mit reaktiven Programmiermustern vertraut zu machen. Dies kann insbesondere für Teams relevant sein, die bereits über umfangreiche Erfahrung in der imperativen Programmierung verfügen, aber dennoch skalierbare und effiziente Anwendungslösungen in Java anstreben.

Zusammenfassend kann festgestellt werden, dass die Integration von virtuellen Threads in die Java-Programmierung eine bedeutende Veränderung in der Herangehensweise an die Entwicklung von Anwendungen darstellt. Diese Entwicklung hat das Potenzial, etablierte Praktiken zu verändern und bietet eine neue Perspektive auf die Optimierung der Leistung von Anwendungen, die in der Programmiersprache Java entwickelt werden.

7.2 Limitationen der Forschung

Trotz sorgfältiger Durchführung und Analyse gibt es Begrenzungen in dieser Studie, die zukünftige Forschungen leiten könnten.

Obwohl in dieser Studie die Speichernutzung insbesondere im Eden Space der JVM betrachtet wurde, gibt es weitere relevante Speicherbereiche, die ebenfalls untersucht werden könnten. Beispielsweise könnten der Survivor Space, der Old Gen Space oder der Metaspace der JVM in zukünftigen Studien näher betrachtet werden. Eine detaillierte Analyse dieser Bereiche könnte ein tieferes Verständnis der Speichereffizienz und des Speicherverhaltens der untersuchten Ansätze, insbesondere unter verschiedenen Lastbedingungen und im Langzeitbetrieb, liefern.

Darüber hinaus basieren die Ergebnisse auf einer begrenzten Anzahl von Testdurchläufen. Eine Erweiterung der Testreihe könnte die Datenbasis verbreitern und zu noch repräsentativeren und robusteren Ergebnissen führen. Dies würde die Zuverlässigkeit und Genauigkeit der Schlussfolgerungen erhöhen und zu einem besseren Verständnis der Leistungsvariabilität unter verschiedenen Szenarien beitragen.

Außerdem wurden die aktuellen Tests unter spezifischen Lastbedingungen durchgeführt, die eine Vielzahl von Anwendungsfällen abdecken. Es wäre jedoch aufschlussreich, die Leistung der betrachteten Programmiermodelle unter noch höheren Belastungen zu untersuchen. Solche Extremszenarien könnten aufzeigen, wie die Systeme unter maximaler Stressbelastung reagieren und wo ihre Grenzen hinsichtlich Skalierbarkeit und Stabilität liegen.

Diese Limitationen sollten als Anregungen für zukünftige Forschungsarbeiten verstanden werden. Sie bieten wertvolle Perspektiven für weiterführende Untersuchungen in diesem dynamischen und sich ständig entwickelnden Bereich der Softwareentwicklung und können dazu beitragen, ein umfassenderes Verständnis der betrachteten Technologien zu erlangen.

7.3 Empfehlungen für die Praxis

Die Ergebnisse dieser Studie liefern praktische Hinweise für Softwareentwickler, die für die Entwicklung und Wartung von Spring-Anwendungen verantwortlich sind. Die folgenden Empfehlungen sollen helfen, die Leistung und Skalierbarkeit von Anwendungen zu optimieren.

Zunächst sollte man darüber nachdenken, native Threads in bestehenden Spring MVC oder anderen Java-Anwendungen durch virtuelle Threads zu ersetzen. Einer der bemerkenswertesten Vorteile von virtuellen Threads ist ihre einfache Integration in bestehende Spring MVC-Anwendungen. Durch eine einfache Konfigurationsänderung können native Threads durch virtuelle Threads ersetzt werden, was zu einer direkten Verbesserung der Skalierbarkeit und Effizienz führt. Diese Anpassung ermöglicht es Entwicklern, die Vorteile virtueller Threads zu nutzen, ohne die Anwendungslogik grundlegend ändern zu müssen.

Darüber hinaus wird generell empfohlen, virtuelle Threads gegenüber reaktiven Programmierparadigmen zu bevorzugen. Virtuelle Threads stellen eine attraktive Alternative zur reaktiven Programmierung dar, insbesondere wenn es um die Einfachheit der Implementierung geht. Abbildung 30 zeigt eine einfache Klassifizierung von nativen Threads, virtuellen Threads und reaktiver Programmierung.

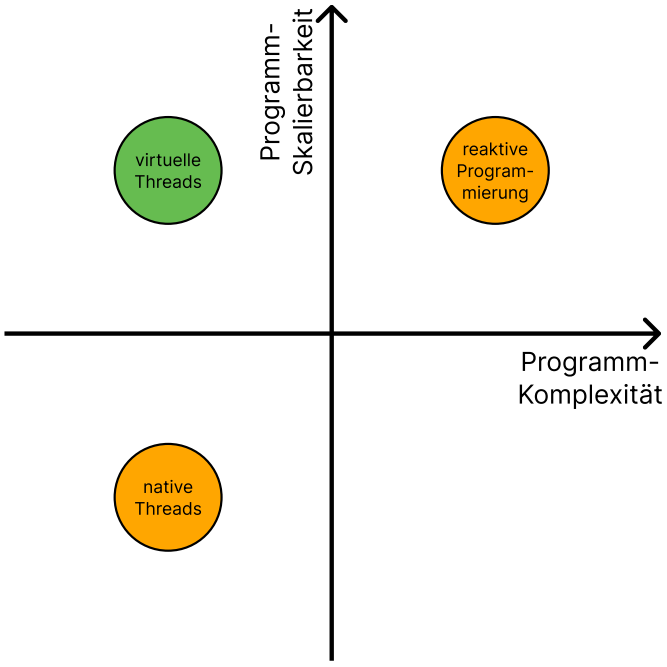

Abbildung 30 Programm Komplexität vs. Skalierbarkeit

Sie ermöglichen eine hochskalierbare und effiziente Anwendungsentwicklung, ohne dass komplexe reaktive Programmiermuster erforderlich sind. Entwickler sollten jedoch die Hardwarespezifikationen im Auge behalten und gegebenenfalls Begrenzungen vornehmen, um eine Ressourcenüberlastung zu vermeiden. Insbesondere ist es ratsam, die maximale Anzahl von Threads im Benutzer-Modus-Scheduler manuell zu setzen, wenn dies notwendig ist, um ein Gleichgewicht zwischen Leistung und Ressourcenverbrauch zu gewährleisten.

Obwohl virtuelle Threads in vielen Szenarien eine effektive Alternative darstellen, behält die reaktive Programmierung ihre Relevanz für bestimmte Aufgaben, insbesondere für Anwendungen, die mit kombinierten Lasten arbeiten. Es wird empfohlen, virtuelle Threads für das spezifische Anwendungsszenario zu testen, bevor eine vollständige Abkehr von reaktiven Ansätzen in Betracht gezogen wird. Eine solche Evaluierung kann helfen, die am besten geeignete Technologie für die spezifischen Anforderungen zu identifizieren.

Diese Empfehlungen spiegeln die Ergebnisse der vorliegenden Studie wider und sollen Softwareentwicklern, die die Vorteile moderner Programmieransätze in Java-basierten Anwendungen optimal nutzen wollen, als Leitfaden dienen. Wichtig ist jedoch, dass jede Entscheidung unter Berücksichtigung der spezifischen Anforderungen und Rahmenbedingungen der jeweiligen Anwendung getroffen wird.

7.4 Vorschläge für zukünftige Forschungen

Die Ergebnisse dieser Studie bieten interessante Ansatzpunkte für weitere Forschung im Bereich der Java-Softwareentwicklung. Im Folgenden werden einige Vorschläge diskutiert, die auf diesen Erkenntnissen aufbauen und neue Wege in der Technologie beschreiten könnten.

Die Verwendung von GraalVM[2] in Kombination mit virtuellen Threads wäre beispielsweise ein interessantes Forschungsfeld. GraalVM stellt einen bedeutenden Fortschritt in der Welt der virtuellen Maschinen dar. Als polyglotte virtuelle Maschine bietet sie nicht nur Unterstützung für eine Vielzahl von Programmiersprachen, sondern auch fortschrittliche Optimierungstechniken, die über die Fähigkeiten herkömmlicher JVMs hinausgehen. Zu ihren Hauptvorteilen zählen eine verbesserte Leistung durch einen hochleistungsfähigen JIT-Compiler, die Möglichkeit der Ahead-of-Time Kompilierung und eine verbesserte Speicherverwaltung. Diese Eigenschaften könnten in Kombination mit virtuellen Threads genutzt werden, um moderne, skalierbare Anwendungen zu realisieren, die von schnelleren Startzeiten, geringerem Speicherverbrauch und effizienterer Ausführung profitieren.

Ein zukünftiger Forschungsschwerpunkt könnte darin bestehen, zu untersuchen, wie sich virtuelle Threads in einer GraalVM-Umgebung im Vergleich zu einer Standard-JVM verhalten. Insbesondere wäre es interessant zu analysieren, wie die effiziente JIT-Kompilierung und Speicherverwaltung von GraalVM die Vorteile virtueller Threads, wie verbesserte Skalierbarkeit und geringerer Ressourcenverbrauch, weiter steigern könnte. Solche Studien könnten aufzeigen, wie GraalVM und virtuelle Threads zusammenwirken, um performante Anwendungen zu ermöglichen, die gleichzeitig ressourceneffizient sind.

Ein weiteres interessantes Forschungsfeld könnte die Erprobung verschiedener Benutzer-Modus-Scheduler sein. Die Einführung virtueller Threads in Java bietet die Möglichkeit, verschiedene Benutzer-Modus-Scheduler zu verwenden. Zukünftige Studien könnten verschiedene Scheduler-

[2] GraalVM: https://www.graalvm.org

Implementierungen testen, um herauszufinden, welche Konfigurationen die besten Ergebnisse in Bezug auf Durchsatz, Antwortzeiten und Ressourcennutzung liefern.

Darüber hinaus könnte es interessant sein, ein breiteres Spektrum von I/O- und CPU-basierten Aufgaben zu testen, um ein umfassenderes Verständnis der Leistung von virtuellen Threads und reaktiven Programmieransätzen zu erhalten. Dies würde helfen, die Anwendbarkeit und Effizienz der verschiedenen Ansätze unter einer Vielzahl realer Betriebsbedingungen zu bewerten.

Darauf aufbauend wäre die Untersuchung noch höherer Lasten sinnvoll. Die Durchführung von Tests unter hoher Last würde es ermöglichen, die Grenzen der Systeme zu ermitteln und einen direkten Vergleich zwischen reaktiver Programmierung und virtuellen Threads unter Stressbedingungen durchzuführen. Solche Tests von Extremszenarios sind entscheidend, um die Robustheit und Stabilität der verschiedenen Ansätze in Hochlastsituationen zu verstehen.

Diese Vorschläge zielen darauf ab, das Verständnis der Wechselwirkungen zwischen modernen Programmieransätzen und der zugrundeliegenden Infrastruktur zu vertiefen. Sie bieten spannende Möglichkeiten, die Grenzen heutiger Technologien auszuloten und neue Wege in der Softwareentwicklung zu beschreiten.

7.5 Abschließende Gedanken

Am Ende dieser Arbeit wird über die tiefe und komplexe Reise, die diese Forschung darstellte, reflektiert. Die Untersuchung der Integration von virtuellen Threads in das Spring Framework und die daraus resultierenden Auswirkungen auf die Skalierbarkeit und Effizienz von Java-Webanwendungen war nicht nur eine technische Herausforderung, sondern auch eine intellektuelle Bereicherung.

Diese Studie hat nicht nur bestehende Annahmen und Praktiken in der Java-Entwicklung in Frage gestellt, sondern auch neue Wege eröffnet, wie über Software-Leistung und -Effizienz nachgedacht werden könnte. Die Einführung virtueller Threads durch das Projekt Loom markiert einen Wendepunkt, der das Potenzial hat, die Art und Weise, wie Java-Anwendungen entwickelt und optimiert werden können, grundlegend zu verändern. Es ist ermutigend zu sehen, wie technologische Fortschritte neue Türen öffnen und gleichzeitig bestehende Paradigmen herausfordern.

Die Hoffnung dieser Arbeit ist, dass sie nicht nur als Sammlung von Erkenntnissen und Daten dient, sondern auch als Inspiration für andere Forscher und Praktiker, die Grenzen der Softwareentwicklung weiter zu erforschen und zu verschieben. Die Softwareentwicklung ist ein dynamisches Feld, in dem ständig neue Technologien und Methoden entstehen. Es ist von entscheidender Bedeutung, dass die Gemeinschaft offen für Veränderungen bleibt und bereit ist, neue Ansätze zu erproben und zu adaptieren.

Abschließend wird betont, dass diese Forschung erst am Anfang steht. Es gibt noch viele unerforschte Bereiche und potenzielle Anwendungen für virtuelle Threads und andere fortschrittliche Technologien in der Java-Entwicklung. Es wird sich darauf gefreut, wie zukünftige Forschung auf diesen Erkenntnissen aufbaut und innovative Lösungen für die Herausforderungen in der Softwareentwicklung findet.

Literaturverzeichnis

Apache Software Foundation. 2023. „Apache Tomcat 8 Configuration Reference (8.5.97) - The Executor (thread pool)“. https://tomcat.apache.org/tomcat-8.5-doc/config/executor.html (24. Dezember 2023).

Augsten, Stephan. und Koller, Dirk. 2021. „Services und DTOs in Spring-Boot-Anwendungen“. *Dev-Insider*. https://www.dev-insider.de/services-und-dtos-in-spring-boot-anwendungen-a-991082/ (7. November 2023).

Baeldung. 2018. „Guide to JNI (Java Native Interface) | Baeldung“. https://www.baeldung.com/jni (2. November 2023).

Baeldung. 2019. „OpenJDK Project Loom | Baeldung“. https://www.baeldung.com/openjdk-project-loom (13. Oktober 2023).

Baeldung. 2020. „Guide to Work Stealing in Java | Baeldung“. https://www.baeldung.com/java-work-stealing (27. Oktober 2023).

Baun, Christian. 2017. *Betriebssysteme kompakt*. Berlin, Heidelberg: Springer Berlin Heidelberg. https://opac.th-brandenburg.de/00/bvnr/BV044291200 (15. Dezember 2023).

Behler, Marco. 2022. „Understanding Java’s Project Loom“. https://www.marcobehler.com/guides/java-project-loom (27. Oktober 2023).

Bonér, Jonas. Farley, Dave. Kuhn, Roland. und Thomposn, Martin. 2014. „Das Reaktive Manifest“. https://www.reactivemanifesto.org/de (19. September 2023).

Carter, Kyle. 2022. „Java Virtual Threads, Millions of Threads Within Grasp“. https://blog.scaledcode.com/blog/java-virtual-threads-intro/# (24. Oktober 2023).

Chandrakant, Kumar. 2020a. „Concurrency in Spring WebFlux | Baeldung“. https://www.baeldung.com/spring-webflux-concurrency (24. Oktober 2023).

Chandrakant, Kumar. 2020b. „Light-Weight Concurrency in Java and Kotlin | Baeldung on Kotlin“. https://www.baeldung.com/kotlin/java-kotlin-lightweight-concurrency (2. November 2023).

Charousset, Dominik. 2020. „Scheduler — CAF 0.17.5 documentation“. https://actor-framework.readthedocs.io/en/0.17.5/Scheduler.html (27. Oktober 2023).

Chen, Kuo-Yi. Chang, Jien Morris. und Hou, Thing-Wei. 2010. „Multithreading in Java: Performance and Scalability on Multicore Systems“. https://ieeexplore.ieee.org/document/5661769.

Christensen, Ben. und Kuhn, Roland. 2015. „Reactive Streams“. https://www.reactive-streams.org/ (22. September 2023).

Ciocîrlan, Daniel. 2023. „The Ultimate Guide to Java Virtual Threads“. *Rock the JVM Blog*. https://blog.rockthejvm.com/ultimate-guide-to-java-virtual-threads/ (27. Oktober 2023).

D. Beronić, L. Modrić, B. Mihaljević, und A. Radovan. 2022. „Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines“. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, , 1466–71.

Dahlin, Karl. 2020. „An Evaluation of Spring WebFlux“. Master Thesis. MID Sweden University.

https://www.diva-portal.org/smash/get/diva2:1445480/FULLTEXT01.pdf.

Deinum, Marten. Rubio, Daniel. und Long, Josh. 2023. *Spring 6 Recipes: A Problem-Solution Approach to Spring Framework*. Berkeley, CA: Apress. https://opac.th-brandenburg.de/00/bvnr/BV048918980 (18. Dezember 2023).

Dijkstra, Edsger Wybe. 1965. „E.W.Dijkstra Archive: Cooperating sequential processes (EWD 123)". https://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html (16. Dezember 2023).

Grammes, Rüdiger. und Schaal, Kristine. 2015. „Datenströme asynchron verarbeiten mit Reactive Streams". https://www.sigs-datacom.de/uploads/tx_dmjournals/grammes_schaal_JS_05_15_vYon.pdf.

Gravelle, Robert. 2023. „Java Thread Methods: A Comprehensive Guide". *Developer.com*. https://www.developer.com/java/java-thread-methods/ (21. Oktober 2023).

Gupta, Lokesh. 2022. „Java Virtual Threads - Project Loom". *HowToDoInJava*. https://howtodoinjava.com/java/multi-threading/virtual-threads/ (21. Oktober 2023).

Iwanowski, Sebastian. und Kozieł, Grzegorz. 2022. „Comparative analysis of reactive and imperative approach in Java web application development". In Journal of Computer Sciences Institute, 242–49. https://doi.org/10.35784/jcsi.2999.

Jansen, Grace. und Jiang, Emily. 2020. „Defining the term reactive". *IBM Developer*. https://developer.ibm.com/articles/defining-the-term-reactive/ (22. September 2023).

Johnson, Rod u. a. 2023. „Spring Framework Reference Documentation". https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/overview.html# (31. Oktober 2023).

Long, Josh. 2020. *Reactive Spring*. Amazon Fulfillment. https://books.google.de/books?id=k4vYzQEACAAJ.

Maurer, Norman. Kuhn, Roland. und Dokuka, Oleh. 2023. „Reactive Streams Documentation". https://github.com/reactive-streams (27. Oktober 2023).

Nurkiewicz, Tomasz. 2022. „Project Loom: Revolution in Java Concurrency or Obscure Implementation Detail?" *InfoQ*. https://www.infoq.com/presentations/loom-java-concurrency/ (2. November 2023).

Öggl, Bernd. und Kofler, Michael. 2021. *Docker: Das Praxisbuch für Entwickler und DevOps-Teams*. Rheinwerk. https://opac.th-brandenburg.de/00/bvnr/BV047551808.

Oleschuk, Lucy. 2023. „Project Loom, Containers, and Jakarta EE: What Are The Java Trends in 2023?" *CodeGym*. https://codegym.cc/groups/posts/1012-project-loom-containers-and-jakarta-ee-what-are-the-java-trends-in-2023 (17. Oktober 2023).

Oracle. 2022. „Java Documentation". *Oracle Lesson: Concurrency*. https://docs.oracle.com/javase/tutorial/essential/concurrency/ (11. August 2023).

Oracle. 2023a. „Java Platform, Standard Edition 8 API Specification". https://docs.oracle.com/javase/8/docs/api/java/lang/package-summary.html (20. Oktober 2023).

Oracle. 2023b. „Java Platform, Standard Edition Core Libariers, Release 21". *Java Platform, Standard Edition Core Libariers, Release 21*. https://docs.oracle.com/en/java/javase/21/core/index.html (6. Oktober 2023).

Otta, Maxmilián. 2023. „What Is Reactive Programming? | Baeldung on Computer Science". https://www.baeldung.com/cs/reactive-programming (22. Oktober 2023).

P. Pufek, D. Beronić, B. Mihaljević, und A. Radovan. 2020. „Achieving Efficient Structured Concurrency through Lightweight Fibers in Java Virtual Machine“. In *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, , 1752–57.

Paraschiv, Eugen. 2018. „Spring MVC Tutorial“. https://www.baeldung.com/spring-mvc-tutorial (31. Oktober 2023).

Piazzolla, Gaetano. 2022. „Java Virtual Threads“. *Medium*. https://blog.devgenius.io/java-virtual-threads-715c162c6c39 (2. November 2023).

Piazzolla, Gaetano. 2023. „Working with Virtual Threads in Spring 6 | Baeldung“. https://www.baeldung.com/spring-6-virtual-threads (31. Oktober 2023).

Pressler, Ron. 2020. „Loom Proposal“. https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html (16. Oktober 2023).

Pressler, Ron. 2021. „Java’s Project Loom, Virtual Threads and Structured Concurrency with Ron Pressler“. *InfoQ*. https://www.infoq.com/podcasts/java-project-loom/ (27. Oktober 2023).

ReactiveX. 2023. „ReactiveX - Observable“. https://reactivex.io/documentation/observable.html# (24. Oktober 2023).

Royal, Peter. 2023. *Building Modern Business Applications: Reactive Cloud Architecture for Java, Spring, and PostgreSQL*. Berkeley, CA: Apress. https://opac.th-brandenburg.de/00/bvnr/BV048638948 (15. Dezember 2023).

Sadakath, Shazin. 2022. „Running 1 Million Threads in Java“. *Medium*. https://shazinsadakath.medium.com/running-1-million-threads-in-java-a427adc8f60a (21. Oktober 2023).

Schmidt, Douglas. und Vinoski, Steve. 1995. „Comparing Alternative Programming Techniques for Multi-Threaded Servers (Column 5)“.

Silberschatz, Abraham. Galvin, Peter Baer. und Gagne, Greg. 2018. 10 *Operating System Concepts*.

Sommerville, Ian. 2018. *Software Enginnering*. Hallbergmoos: Pearson. https://opac.th-brandenburg.de/00/bvnr/BV045245808.

StackOverflow. 2022. „Stack Overflow Developer Survey 2020“. *Stack Overflow*. https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020 (15. Dezember 2023).

Takeuchi, Satoru. 2019. „A Brief History of the Linux Kernel’s Process Scheduler: The Very First Scheduler“. *DEV Community*. https://dev.to/satorutakeuchi/a-brief-history-of-the-linux-kernel-s-process-scheduler-the-very-first-scheduler-v0-01-9e4 (7. November 2023).

Tanenbaum, Andrew. und Bos, Herbert. 2015. *Modern Operating Systems*. 4. ed. Boston: Prentice Hall.

Ullenboom, Christian. 2014. *Java ist auch eine Insel*. Bonn: Galileo Press. https://opac.th-brandenburg.de/00/bvnr/BV041866299 (29. September 2023).

Walls, Craig. 2012. *Spring Im Einsatz*. 2., überarb. Aufl. München: Hanser. https://opac.th-brandenburg.de/00/bvnr/BV040269450.

Weinländer, Markus. 1995. *Entwicklung paralleler Betriebssysteme*. Braunschweig: Vieweg. https://opac.th-brandenburg.de/00/bvnr/BV010349434.

Wolf, Jürgen. und Krooß, René. 2023. *C von A bis Z*. 3. Aufl. https://openbook.rheinwerk-verlag.de/c_von_a_bis_z/026_c_paralleles_rechnen_003.htm (7. November 2023).

Abbildungsverzeichnis

Quelltextverzeichnis

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt wurde.

Brandenburg, 01.02.2024
Ort, Datum

Unterschrift

(Maximilian Milz)