Ostfalia - Hochschule für angewandte Wissenschaften
Fakultät Informatik
Studiengang Informatik

Master-Arbeit

Reactive Programming mit Quarkus

eingereicht bei Prof. Dr. B. Müller

von Erik Simonsen 70455429

Wolfenbüttel, den 28. Juni 2021

Zusammenfassung

Ehrenwörtliche Erklärung

Hiermit erkläre ich ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne unerlaubte fremde Hilfe angefertigt habe, andere als die angegebenen Quellen nicht benutzt und die benutzten Quellen wörtlich oder die inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Wolfenbüttel, den 28. Juni 2021

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Motivation

1.2 Aufbau

2 Grundlagen

2.1 Threads & Prozesse in Java

In der Java-Laufzeitumgebung sind Prozesse und Threads als Betriebssystem-Prozesse realisiert, sog. *native threads* oder auch *Kernelthreads*. Hierbei wird die Ausführungsreihenfolge, die Ausführungszeit und der Prozess- & Threadwechsel vom Scheduler & Dispatcher des Betriebssystems übernommen (Tanenbaum und Bos 2016). Die Threading-Abstraktion in Java bietet Entwicklern verhältnismäßig leichten Zugriff auf parallelle Programmierung und Synchronisation von Threads.
Servlet-Container binden üblicherweise jede vom Webserver weitergeleitete Anfrage an einen Thread[1] im Servlet-API, welcher die jeweilige Anfrage imperativ abarbeitet (daher auch *worker thread*).
Der auszuführende Code ist in diesem Ansatz an den jeweiligen Thread gekoppelt, dieser wartet bei asynchronen Ereignissen solange, bis er eine Antwort erhält und blockiert den Thread bis dahin.

Um jede Anfrage, und somit jeden Thread, scheinbar parallel zu bearbeiten wird vom Scheduler des Betriebssystems regelmäßig ein Kontextwechsel zwischen den Threads, ein *thread-switch*, durchgeführt. Während bei einem Prozesswechsel der gesamte Programmkontext (Adressraum, Inhalt der CPU-Register, Seitentabelle, geöffnete Dateien und Metainformationen) gewechselt werden muss, wird bei einem Threadwechsel lediglich der Inhalt der CPU-Register (inkl. Programmzähler) ersetzt(Brosenne 2018)(Mosberger und Eranian 2002). Da der Kontextwechsel, im Fall von *native threads*, durch Systemaufrufe, also vom Kernel des Betriebssystems, ausgeführt werden muss, entsteht auch bei einem Threadwechsel ein messbarer Zeitverlust.
Weitere Threadwechsel entstehen, wenn ein Thread die zugewiesene Rechenzeit nicht nutzen kann, da er noch durch ein asynchrones Ereignis (abgesetzte Datenbankabfragen oder weitere aufgerufene Webservices) blockiert, und diese einem anderen Thread zugeteilt wird.

Während dieser Zeitverlust für hoch frequentierte Anwendungen lange Zeit kein Problem darstellte, sind die Anforderungen an Webanwendungen in den letzten Jahren durch steigende Nutzerzahlen und Architekturen, die stark auf Client-Server-Kommunikation basieren, erheblich gestiegen. Ab einer gewissen Menge an Anfragen stellen die Kosten der Threadwechsel von *native threads* ein Performance Bottleneck (in Form von Durchsatz) dar.

[1] Aus einem, im vornherein erzeugten, Thread-Pool

2.2 Reaktive Programmierung

Reaktive Programmierung ist ein Programmierparadigma, bei dem der Programmablauf als Sequenz von asynchronen Ereignissen (Events), und Daten als -von außen- unveränderliche (immutable) Datenströme (Streams) dargestellt werden. Sobald es innerhalb des Datenstroms zu Änderungen kommt werden diese als Events durch einen Publisher veröffentlicht.[2] Die eigentliche Programmlogik wird in Funktionen ausgeführt, die auf die veröffentlichten Events hören (Subscriber), sie verarbeiten und wiederrum welche veröffentlichen können. Die Grundidee orientiert sich am Observer-Pattern und dessen Ausprägung: dem Publish-Subscribe Pattern, erweitert diese aber noch um die Benachrichtigungen des Subscribers:

1. Sobald keine Events mehr kommen
2. Wenn ein Fehler aufgetreten ist

Indem Änderungen direkt propagiert werden und Subscriber keine Kontrolle über den Datenfluss haben, sondern lediglich über Änderungen informiert werden, können Programme ohne jeglichen Zustand realisiert werden(Escoffier 2017).

Reaktive Programmierung verinnerlicht das Konzept von nicht-blockierender bzw. asynchroner Ein- und Ausgabe (I/O). Dabei wird, statt wie bei synchroner bzw. blockierender Ein- und Ausgabe die restliche Ausführung des Programms zu blockieren bis die Datenübertragung abgeschlossen ist, nach dem Start der Übertragung bereits begonnen die Teile des Programms auszuführen, die nicht von dem Ergebnis der I/O Operation abhängen.

Dadurch können mehrere parallele Anfragen von dem gleichen Thread bearbeitet werden. Methoden die blockierende I/O Operationen ausführen, wie Datenbankzugriffe oder Anfragen von externen Services, geben beim Aufruf unverzüglich einen Publisher zurück, auf dem sich der Aufrufer registriert (subscribe). Dadurch wird der bearbeitende Thread nicht blockiert, und kann die nächste Anfrage bearbeiten. Sobald das Ergebnis der I/O Operation bereit ist, wird es dem Publisher in Form eines Events mitgeteilt, von diesem veröffentlicht und die Anfrage kann vom Aufrufer bzw. Subscriber weiter bearbeitet werden. Da durch dieses Modell der auszuführende Code nicht mehr an den jeweils ausführenden Thread gebunden wird, erlaubt es die Nutzung eines einzigen Threads (*sog. IO Thread*) statt eines *Threadpools*. Dadurch ergeben sich folgende Vorteile:

1. Die Antwortenzeiten sind für eine hohe Last geringer, da deutlich weniger Threadwechsel gemacht werden müssen
2. Der Speicherverbrauch ist geringer, da weniger Threads genutzt werden
3. Der Grad der Paralellität ist nicht von der Anzahl der Threads begrenzt

Allerdings gibt es auch einige gravierende Nachteile:

1. Asynchroner Code ist schwieriger zu schreiben, lesen, testen und zu debuggen als imperativer Code

[2] Änderungen in Datenströmen sind der quasi der Stimulus

2. Sehr aufwendig in bestehende klassische Anwendungen zu integrieren
3. Reaktive Anwendungen müssen in jeder Schicht reaktiv sein (Transaktionen, Security, Datenbanktreiber)
4. Da nur ein IO-Thread genutzt wird resultieren blockierende I/O-Operationen in der Blockierung der gesamten Anwendung

Ein beliebtes Threading-Modell für die Verarbeitung von asynchronen Events ist die *Event Loop*. Sobald ein Event entsteht wird es einer Warteschlange (Queue) in der Event Loop hinzugefügt. Solange der ausführende Thread aktiv ist und die Queue noch Events enthält, wird in einer Schleife das nächste Event abgerufen und an den, für diesen Eventtyp, registrierten Event-Handler bzw. oben beschriebenen Subscriber weitergeleitet.

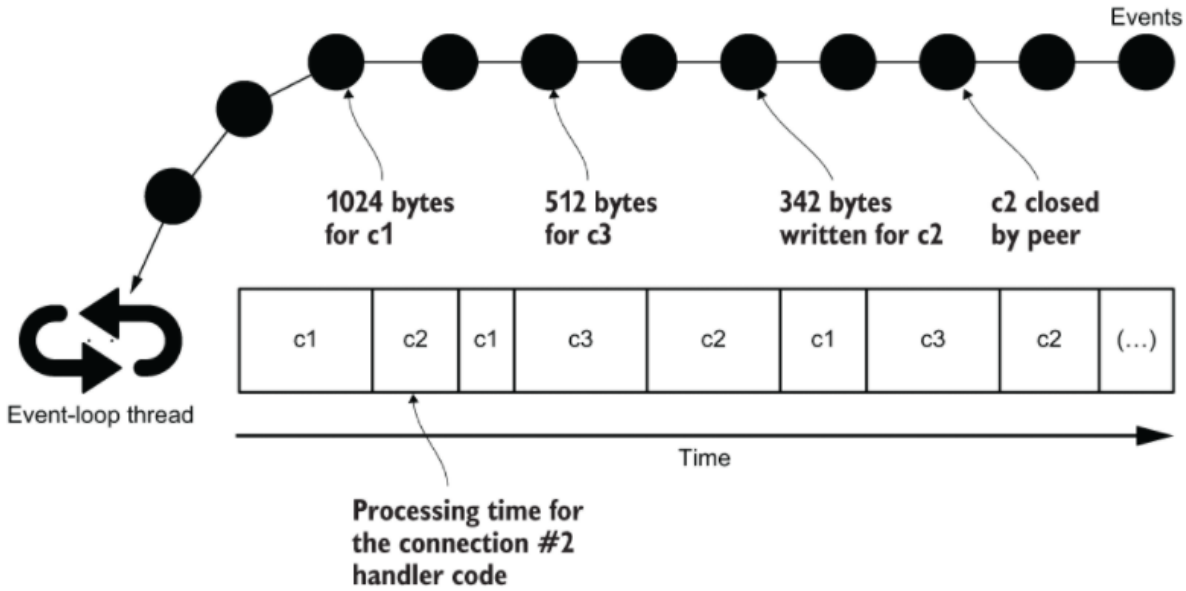

Abbildung 1: Exemplarische Abbildung einer Event Loop. (Ponge 2020, Kapitel 1.7)

Dies können beispielsweise I/O-Events sein, die signalisieren das Daten bereit zur Weiterverarbeitung sind, aber auch jegliches andere Event. Eine Event Loop wird in der Regel auf einem IO-Thread ausgeführt, daher darf das Verarbeiten von Events keine blockierenden, oder zeitintensiven Operationen beinhalten(Ponge 2020).

2.2.1 Alternativen

In Java 1.1 wurden Threads als sog. *Green Threads* implementiert. Dabei wurde die Möglichkeit Threads vom Betriebssystem verwalten zu lassen gar nicht genutzt. Stattdessen lief die komplette JVM in einem einzigen Prozess. *Green threads* waren als *user threads* implementiert [3], dabei ist die Funktionalität nicht im Kernel implementiert (wie bei *kernel-/native threads*), sondern in einer Programmbibliothek im *Userspace*. Da sich das Betriebssystem nicht um das Scheduling von *user threads* kümmert, wurde dies über

[3] Auch als *Fiber* oder *virtual thread* bezeichnet

einen eigenen Scheduling-Algorithmus der JVM geregelt.(*JDK 1.1 for Solaris Developer's Guide - Chapter 2 Multithreading* 2021) Ein *green thread* existiert lediglich als Objekt innerhalb der JVM, und durch die virtuelle Speicherverwaltung entfallen somit die aufwändigen Betriebssystemaufrufe beim Erstellen eines Threads, sowie bei Thread- bzw. Kontextwechseln, denn der ausführende Main-Thread bleibt gleich.

Die Threadwechsel der *user-threads* erfolgten ausschließlich innerhalb des Main-Threads, weswegen keine echte Parallelität realisiert werden konnte, da immer nur ein Prozessorkern genutzt wurde. Während der Vorteil dieses Modells darin lag, dass es keine 'echten' parallelen Zugriffe auf eine Resource innerhalb des JVM-Prozesses geben konnte und die Synchronisation von Datenzugriffen daher leicht war, überwog schließlich der Umstand, dass keine Nutzung von mehreren Prozessorkernen durch Multithreading möglich war, weswegen *Green Threads* ab Java 1.3 zugunsten von *native threads* entfernt wurden.

Mit dem OpenJDK Projekt *Project Loom* ist die Idee von *Green threads* wieder aufgegriffen worden, allerdings nun als Ergänzung (statt Alternative) zu *native threads*. Statt alle virtuellen Threads auf dem nativen Main-Thread auszuführen, werden diese von einer geringen Anzahl an nativen *worker threads*, die als Carrier eingesetzt werden, ausgeführt. Deren Anzahl ist so gewählt, dass alle CPU Kerne durch Multithreading dauerhaft benutzt werden können [4] , aber so wenig Kontextwechsel wie möglich ausgeführt werden müssen. (OpenJDK 2021) [5] Während ein nativer Thread in einer 64 Bit JVM ca. 1 MB für den Threadstack reserviert und zusätzlich noch Metadaten abspeichert, ist ein virtueller Thread lediglich ein Objekt im virtuellen Speicher der JVM und benötigt sehr wenig Resourcen (da er ja im Hintergrund von einem nativen Thread abgearbeitet wird). Aus diesem Grund können durchaus mehrere Millionen virtueller Threads erzeugt werden (bei entsprechendem allokierten Heap-Speicher der JVM), wohingegen die Erstellung von 10.000 nativen Threads entweder den allokierten Speicher weit überschreitet (wodurch der JVM-Prozess abstürzt) oder die Threadgrenze des Betriebssystems überschreitet.

Sobald ein virtueller Thread nun eine blockierende I/O Operation ausführt signalisiert er dem darunterliegenden nativen Thread, dass er momentan nichts machen kann außer Warten, und erlaubt dem nativen Thread zu einem anderen virtuellen Thread zu wechseln.

Das große Versprechen des Projektes ist außerdem, das Entwickler keine asynchronen Programmierparadigmen (wie u.A. *Reactive Programming*) nutzen müssen um die beschriebenen virtuellen Threads (und die damit einhergehenden wesentlichen Performanceverbesserungen) nutzen zu können. Um dieses Versprechen zu halten werden virtuelle Threads, statt als Bibliothek eines Drittanbieters, in Form einer eigenen JDK Version bereitgestellt. In dieser Version wurden viele Teile der Standardbibliothek die mit I/O-Operationen arbeiten so angepasst, dass virtuelle Threads statt native Threads genutzt werden. Auf diese Weise können I/O-Operationen, wie beispielsweise der Aufruf einer Netzwerkfunktionalität, ohne Änderungen am Programm die virtuellen Threads nutzen und blockieren den darunterliegenden nativen Thread nicht mehr.

[4] In der Praxis laufen natürlich noch andere Prozesse auf dem Server, deren Threads auch ausgeführt werden müssen.

[5] Im Idealfall würde auf jedem CPU Kern ein *worker thread* laufen, ohne jemals einen Thread- bzw. Kontextwechsel zu machen.

2.3 Reaktive Datenströme

In einer typischen asynchronen Verarbeitungskette von, potenziell unbegrenzten, Datenströmen bestehend aus einem Sender und Empfänger bzw. Publisher und Subscriber kann es vorkommen, dass der Sender Daten schneller an den Empfänger verschickt, als dieser sie verarbeiten kann. Zwei naive Ansätze mit einer Überlastung des Empfängers umzugehen wären:

1. Nur der Empfänger reagiert auf eine Überlast. Diese kann sich in einem Speicherüberlauf äußern oder, falls der Puffer des Empfängers eine Größenbeschränkung hat, im Verlust der empfangenen Daten

2. Der Sender begrenzt im vornherein die Datenmenge, die er an den Empfänger schickt. Da der Sender allerdings in der Regel nicht weiß wieviel der Empfänger verarbeiten kann, sendet er entweder zuviel (es entsteht eine Überlast), oder er sendet zuwenig wodurch der Durchsatz geringer ist als nötig

(Grammes und Schaal 2015) Die Lösung für dieses Problem wird *Backpressure* genannt. Dabei fordert der Empfänger die Daten entsprechend seiner Kapazitäten an, wodurch der Sender weiß wieviele Daten er maximal versenden darf. Diese Mitteilung muss asynchron geschehen, da bei einer synchronen Kommunikation der Backpressure die Vorteile der asynchronen, reaktiven Datenverarbeitung negiert würden. Da große Anwendungen aus mehreren Schichten (bspw. Routing-Schicht, Persistenzschicht, Geschäftslogik) bestehen und somit zwischen dem Sender und Empfänger mehrere Komponenten liegen können, muss jedes Element der Verarbeitungskette nichtblockierendes, asynchrones Verhalten implementieren, da ansonsten der Rest der Kette blockiert würde.

Aus der Intention einen Standard für die asynchrone Verarbeitung von Datenströmen mit nicht-blockierender *back pressure* zu schaffen, ging die *Reactive Streams*-Initiative hervor. Innerhalb dieser Initiative haben sich mehrere Arbeitsgruppen gebildet, welche die grundlegenden Semantiken erarbeitet haben und sie in Form einer eigenen Java-Spezifikation namens *Reactive Streams*-API implementiert und veröffentlicht haben.(*Reactive Streams* 2021) Diese API wurde anschließend in Java 9, als Schnittstelle namens *Flow-API* dem JDK hinzugefügt.

Die *Flow-API* des JDK entspricht der *Reactive Streams* Spezifikation und stellt (nur) Interfaces zur Verfügung mit denen eine asynchrone, nicht blockierende Verarbeitung von (unbegrenzten) Datenströmen mit *back pressure* auf der JVM implementiert werden kann. (Oracle 2021c). [6]

2.4 Reaktive Systeme

Anforderungen an große Softwaresysteme haben sich in den letzten Jahren stark verändert:

1. Antwortzeiten in Millisekunden statt im Sekundenbereich

2. Datengrößen in Petabytes statt Gigabytes

[6]Nicht zu Verwechseln mit den Java-Streams durch die Collection-API ab Java 8. Diese sind zur Auswertungszeit in ihrer Größe begrenzt und nach der Abarbeitung liegt statt eines Streams eine Collection

3. 100% Verfügbarkeit statt stundenlange Wartungsarbeiten

4. Deployment auf einer Vielzahl von Plattformen und cloud-basierten Clustern mit tausenden Multikernprozessoren

Unternehmen aus verschiedenen Bereichen haben sich voneinander unabhängig an diese Kriterien angepasst und Architekturmuster erarbeitet, mit denen robuste, belastbare und flexible Softwaresysteme entwickelt werden können, die die modernen Anforderungen erfüllen.

2014 wurde mit dem *Reactive Manifesto* versucht diese Ansätze des Systemdesigns in Form eines Manifests zusammenzuführen und daraus allgemeingültige Systemattribute abzuleiten. Laut dieses Manifests sind Systeme reaktiv wenn sie:

1. Reaktionsschnell

2. Widerstandsfähig (gegen Fehler)

3. Elastisch

4. Nachrichtengesteuert

sind. Solche Systeme sind, laut den Autoren, flexibler, stärker entkoppelt und würden besser skalieren als herkömmliche, nicht-reaktive Systeme. Dies mache sie leichter zu entwickeln, zugänglicher für Veränderungen und deutlich fehlertoleranter.

Die Autoren definieren die genannten Systemeigenschaften wie folgt:

Reaktionsschnelligkeit: Das System reagiert, falls überhaupt möglich, rechtzeitig. Reaktionsgeschwindigkeit ist dabei die Grundlage von Nutzen und Benutzbarkeit und ermöglicht das schnelle Erkennen und Behandeln von Fehlern. Der Fokus von reaktionsschnellen Systemen liegt auf konsistenten und schnellen Antwortszeiten. Darüber hinaus schaffen sie verlässliche Obergrenzen um eine konsistente Qualität zu erreichen. Dieses konsistente und verlässliche Verhalten simplifiziert Fehlerbehandlung, und erhöht das Vertrauen der Benutzer.

Widerstandsfähig/Fehlertolerant: Das System bleibt auch bei Fehlern reaktionsschnell. Das gilt nicht nur geschäftskritische, hochverfügbare Systeme - jedes System das nicht Fehlertolerant ist, wird nach Fehlern nicht mehr reaktionsfähig sein. Widerstandsfähigkeit wird durch Redundanz, Eingrenzung, Isolation und Delegation erreicht. Fehler werden innerhalb einer Komponente eingegrenzt und die Komponenten sind voneinander isoliert. Dadurch bleibt das Gesamtsystem stabil, selbst wenn eine einzelne Komponente versagt. Die Wiederherstellung jeder Komponente wird an eine andere (möglicherweise externe) Komponente deligiert, und Hochverfügbarkeit der Komponenten wird, wo notwendig, durch Redundanz gewährleistet.

Elastisch: Das System bleibt reaktionsschnell unter variierenden Arbeitslasten. Auf Änderungen der Arbeitslast wird durch das Anpassen der allokierten Ressourcen reagiert. Das impliziert ein Systemdesign das keine zentralen Performance-Bottlenecks oder Reibunspunkte hat, damit Komponenten problemlos repliziert und die Last darauf verteilt werden kann. Reaktive Systeme unterstützen prädiktive, skalierende Algorithmen zur Ressourcenberechnung, indem Sie die Live-Messungen von Performance relevanten Systemmetriken als Eingabe nutzen.

Nachrichtengesteuert: Reaktive Systeme basieren auf dem asynchronen Austausch von Nachrichten, um die Komponenten voneinander abzugrenzen und dadurch eine loose Kopplung, Isolation und eine transparente Lokalisierung der Komponenten zu ermöglichen. Aufgrund dieser Abgrenzung werden Fehler als Nachrichten an andere Komponenten delegiert. Der Ansatz jegliche Kommunikation der Komponenten durch das Übermitteln von Nachrichten zu implementieren erlaubt Elastizität, indem er das Verteilen der Arbeitslast, und die Kontrolle der Datenströme durch das Überwachen der Nachrichtenwarteschlangen (*message queues*) und, falls nötig, Anwenden von *back pressure*, erlaubt.(Bonér u. a. 2014)

2.5 Werkzeuge

2.5.1 Java Ökosystem

Im Java Ökosystem gibt es eine Vielzahl an Frameworks, Libaries und APIs mit denen Reaktive Programmierung und reaktive Systeme umgesetzt werden können. Um in Java einzelne, asynchrone Prozesse zu implementieren, wird vom JDK die Future-API zur Verfügung gestellt.(Oracle 2021b) Für die Verarbeitung von asynchronen (unbegrenzten) Datenströmen gibt es die Flow-API.(Oracle 2021c) Da die Flow-API lediglich Interfaces bereitstellt, gibt es mehrere Implementierungen von *reactive streams*.

Um reaktive Programmierung zu erleichtern, gibt es unter Anderem die folgenden Projekte:

1. RxJava
2. Spring Webflux
3. Mutiny

Jedes Projekt unterscheidet sich dabei in den verwendeten Klassennamen [7], Operatoren und dem Grad der funktionalen Programmierung.(*ReactiveX* 2021; SmallRye Mutiny 2021) Allerdings sind die meisten Frameworks und Bibliotheken interoperabel, da sie die *reactive streams* Spezifikation implementieren (und damit *back pressure*), und bieten Converter-Klassen an.

Für die Entwicklung von reaktiven Systemen bieten sich mehrere Toolkits und Frameworks an. Sie implementieren bereits Komponenten und Mechanismen wie Messaging, Event Loops, Dateizugriffe, nichtblockierende Netzwerkanwendungen, Web APIs und mehr. Zu den populärsten gehören:

1. Eclipse Vert.x
2. Akka
3. Project Reactor

(*Eclipse Vert.x* 2021; *akka* 2021; *Project Reactor* 2021)

Bei dem, in dieser Arbeit verwendeten, Framework Quarkus handelt es sich, laut Hersteller Red Hat, um ein benutzerfreundliches, auf Entwickler abgestimmtes Java

[7] RxJava - Observable; Mutiny - Uni, Multi; Spring Webflux - Mono, Flux

Framework, welches für Container-, Cloud- und Serverless-Umgebungen optimiert ist und nur wenig Konfiguration benötigt, sowie nur die besten und hochwertigsten Java-Bibliotheken und Standards nutzt. Dabei können die Anwendungen sowohl auf einer JVM (JVM-Mode) laufen, als auch, durch native Kompilierung mit vollständigem Stack, als *native executable* (native-mode). Dafür nutzt Quarkus eine, von Oracle entwickelte, Technologie namens GraalVM. Dabei handelt es sich um eine polyglotte, virtuelle Maschine und Laufzeitumgebung die auf dem OpenJDK basiert, und über JVMCI [8] den C2-Compiler der zugrundeliegenden HotSpot-JVM durch den Graal JIT-Compiler ersetzt.(Oracle 2021a)

Darüber hinaus verspricht Quarkus, durch seine Container-first-Philosophie, bis zu 300 Mal schnellere Startzeiten und nur ein Zehntel des Speicherbedarfs im Vergleich zum 'traditionellen' Java, wodurch es eine signifikante Reduzierung der benötigten Ressourcen im Cloud-Umfeld bewirkt.

Des Weiteren erlaubt Quarkus die Kombination des imperativen und des reaktiven, nicht-blockierenden Programmierparadigmas. Für die reaktive Programmierung bietet Quarkus die bereits genannte Bibliothek Mutiny an. Quarkus selber ist zudem auch reaktiv, denn 'angetrieben' wird es durch eine nicht-blockierende, reaktive Eclipse Vert.x Engine, die alle Netzwerk I/O-Operationen verarbeitet. (Red Hat 2021a; Red Hat 2021b)

3 Vergleich reaktive & imperative Anwendung

Um zu prüfen, ob Leistungsfähigkeit und Skalierbarkeit einer reaktiven Anwendung tatsächlich die einer traditionellen, imperativen Anwendung übertrifft, werden in diesem Kapitel beide Ansätze hinsichtlich verschiedener Metriken in einem festen Zeitintervall miteinander verglichen.

3.1 Implementierung & Systemaufbau

Die beiden Anwendungen implementieren mit dem Quarkus-Framework jeweils eine simple REST-Schnittstelle mit HTTP-CRUD Methoden und einer angebundenen PostgreSQL-Datenbank. Dabei ist vorallem die HTTP-Schicht von Interesse. Die HTTP-Unterstützung von Quarkus basiert auf einem reaktiven, nicht-blockierenden Unterbau: der Vert.x Engine. Jede HTTP-Anfrage wird auf einem der *event-loop threads* bzw. *IO threads* [9] verarbeitet und durch eine Routing-Schicht an den Anwendungscode weitergeleitet. Je nachdem welcher Ansatz zur Implementierung des jeweiligen HTTP-Endpunktes gewählt wurde, wird der Code dann auf einem blockierenden *worker thread* (Servlet, JAX-RS) oder einem der *IO threads* (Reactive Routes, Reactive Resteasy) ausgeführt. Die *IO threads* sind dafür zuständig alle IO-Operationen asynchron auszuführen und die jeweiligen EventListener bzw. Subscriber auszulösen sobald die Operationen abgeschlossen sind.

[8] Java Virtual Machine Compiler Interface

[9] Deren Anzahl hängt von der Anzahl der CPU-Kerne ab

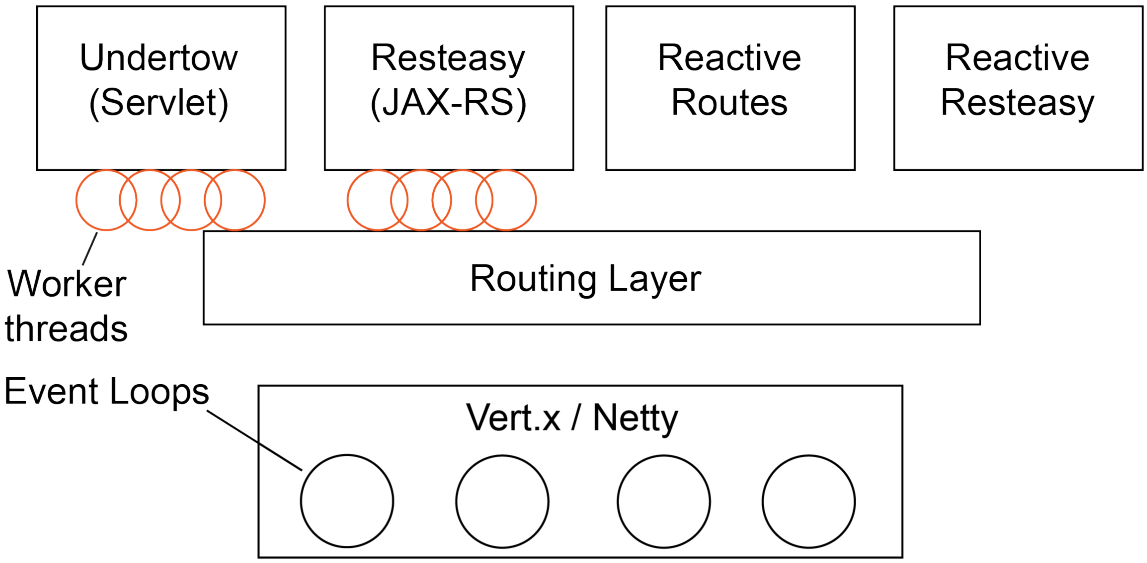

Abbildung 2: Quarkus HTTP-Schicht (Red Hat 2021c)

Damit sich beide Anwendungen nahe an einer realen Java-EE REST-API orientieren, haben Sie (zusätzlich zu den grundlegenden Abhängigkeiten des Quarkus-Frameworks) folgende Projekt-Abhängigkeiten:

1. JAX-RS Implementierung
2. JSON Unterstützung
3. Datenbanktreiber
4. JPA Implementierung

Diese Abhängigkeiten wurden vom Quarkus Maven-Repository sowohl in blockierender, als auch in nicht-blockierender, reaktiver Form bereitgestellt:

	Blockierend	Nicht-blockierend (reaktiv)
JAX-RS	Resteasy	Resteasy Reactive
JSON	Resteasy-Jackson	Resteasy-Reactive-Jackson
Datenbanktreiber	JDBC-Postgresql	Reactive-Pg-Client
JPA-/ORM	Hibernate-ORM	Hibernate-Reactive

Der Projektcode kann vom Gitlab-Server der Ostfalia unter //TODO Ostfalia Gitlab link? eingesehen und geklont werden.

Code-Beispiele für Reactive zeigen erwähnung von http und rest ? Verweis auf Github Repository

3.2 Testbedingungen

//TODO: Erklären welche Hardware Specs und Software Versionn nötig sind, Dinge wie SSH Zugriff

3.3 Vorgehen des Tests / Testaufbau

//TODO: Erklären wie das gesamte System getestet wird, welche Werkzeuge (wrk2 (warum wrk2, latenzmessung im gegensatz zu wrk), top, jbang etc. docker für reproduzierbare umgebung, lua histogramme dann durch javascript-script zu graphen) Systemaufbau (Client-Host, Server-Host, User-Host) //eine minute lang http anfragen an zwei Anwendungen, die genau dasselbe machen was genau gemessen wird (welche Metriken) und wie diees beeeinflusst werden können Jeden größeren Schritt erklären, Bauen der Anwendungen, Warm-Up (JIT), Workload

3.4 Test: Statische Daten

3.4.1 Systemablauf

//TODO: Grafik ähnlich zu Grafik in Implementierung aber mit Threadwechseln und exemplarisch mhrere Threads zeigen (auch angeben wie Ergebnisse mit komplizierteren Queries aussehen könnten)

3.4.2 Resultate

3.5 Test: Datenbankzugriffe

3.5.1 Systemablauf

//TODO: Grafik ähnlich zu Grafik in Implementierung aber ohne Threadwechsel dafür Main-Thread mit dahinterliegender Datenbank, das Nicht BLockieren bzw. Asynchronität verdeutlichen

3.5.2 Resultate

3.6 Auswertung

4 Fazit

Bezug auf in Einleitung angegebene Ziele

Literatur

akka (2021). URL: `https://akka.io/` (besucht am 22. 06. 2021).

Bonér, Jonas u. a. (16. Sep. 2014). *The Reactive Manifesto*. URL: `https://www.reactivemanifesto.org/` (besucht am 13. 06. 2021).

Brosenne, Dr. Henrik (2018). *Betriebssysteme*. URL: `https://user.informatik.uni-goettingen.de/~brosenne/vortraege/os2018ws/os20181030.pdf` (besucht am 05. 06. 2021).

Eclipse Vert.x (2021). URL: `https://vertx.io/` (besucht am 19. 06. 2021).

Escoffier, Clement (2017). *Building Reactive Microservices in Java - Asynchronous and Event-Based Application Design*. O'Reilly. ISBN: 9781491986288.

Grammes, Rüdiger und Kristine Schaal (2015). „Datenströme asynchron verarbeiten mit Reactive Streams“. In: *JavaSpektrum* 05/2015, S. 44–48.

JDK 1.1 for Solaris Developer's Guide - Chapter 2 Multithreading (2021). URL: `https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqe/` (besucht am 13. 06. 2021).

Mosberger, David und Stephane Eranian (2002). *IA-64 Linux Kernel: Design and Implementation*. Pearson. ISBN: 9780130610140.

OpenJDK (2021). *Loom - Fibers, Continuations, and Tail-Calls for the JVM*. URL: `https://openjdk.java.net/projects/loom/` (besucht am 09. 06. 2021).

Oracle (21. Juni 2021a). *Get Started with GraalVM*. URL: `https://www.graalvm.org/docs/getting-started/` (besucht am 21. 06. 2021).

— (2021b). *Class CompletableFuture<T>*. URL: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html` (besucht am 19. 06. 2021).

— (2021c). *Class Flow*. URL: `https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/Flow.html` (besucht am 19. 06. 2021).

Ponge, Julien (2020). *Vert.x in Action - Asynchronous and Reactive Java*. Manning. ISBN: 9781617295621.

Project Reactor (2021). URL: `https://projectreactor.io/` (besucht am 22. 06. 2021).

Reactive Streams (2021). URL: `http://www.reactive-streams.org/` (besucht am 13. 06. 2021).

ReactiveX (2021). URL: `http://reactivex.io/` (besucht am 19. 06. 2021).

Red Hat (2021a). *Getting started reactive*. URL: `https://quarkus.io/guides/getting-started-reactive` (besucht am 22. 06. 2021).

— (2021b). *Quarkus - Subersonic Subatomic Java*. URL: `https://quarkus.io/` (besucht am 19. 06. 2021).

— (2021c). *Using Reactive Routes*. URL: `https://quarkus.io/guides/reactive-routes` (besucht am 27. 06. 2021).

SmallRye Mutiny (2021). *Mutiny! - Intuitive Event-Driven Reactive Programming Library for Java*. URL: `https://smallrye.io/smallrye-mutiny/` (besucht am 19. 06. 2021).

Tanenbaum, Andrew S. und Herbert Bos (2016). *Moderne Betriebssysteme*. Pearson Deutschland GmbH. ISBN: 9783868942705.