

B

C

Bachelorarbeit

Kriterien und Möglichkeiten für den Einsatz von Reaktiver Programmierung

S

Bachelorarbeit

Kriterien und Möglichkeiten für

den Einsatz von Reaktiver

Programmierung

von

Kirill Meng

zur Erlangung der akademischen Grades eines

Bachelor of Science

in angewandter Informatik

an der Hochschule Konstanz,

Matrikelnummer: 292154

Abgabedatum: 28 Februar 2019

1. Prüfer: **Prof. Dr. Marko Boger**

2. Prüfer: **Dipl.-Wirt.-Inf. Stefan Waldmann**

Eine elektronische Version dieses Dokuments ist hier zu finden <https://github.com/Menkir/reactive-streams-tex/thesis.pdf>.

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Kirill Meng, geboren am 25.08.1994 in Orenburg,

- (1) dass ich meine Bachelorarbeit mit dem Titel:

"Kriterien und Möglichkeiten von Reaktiver Programmierung"

in der Fakultät Informatik unter Anleitung von Professor Dr. Marko Boger selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angeführten Hilfen benutzt habe;

- (2) dass ich die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

- (3) dass die eingereichten Abgabe-Exemplare in Papierform und im PDF-Format vollständig übereinstimmen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 28.02.2019

Kirill Meng

Abstract

In der vorliegenden Thesis soll die Frage beantwortet werden, worin Möglichkeiten und Szenarien der reaktiven Programmierung liegen. Dazu wird der Begriff zunächst differenziert und abgegrenzt. Aus der Sicht der Architektur beschreibt die reaktive Programmierung das Design eines Systems, das die vier Qualitäten des reaktiven Manifests umsetzt. Ein reaktives System muss somit widerstandsfähig, elastisch, und nachrichtenorientiert sein, damit es eine hohe Antwortbereitschaft garantieren kann. Um die Einsatzszenarien solcher Systeme aufzuzeigen, werden Erfolgsgeschichten von Unternehmen wie Netflix, PayPal und Verizon betrachtet. Der Einsatz reaktiver Systeme vereinfacht zum einen die Entwicklung der Projekte, zum anderen steigert er den Umsatz der Unternehmen.

Aus programmatischer Sicht bezeichnet die reaktive Programmierung ein Programmierparadigma. Die Technologien Reactive Streams, Futures und Spreadsheets sind mögliche Ausprägungen davon. Die Reactive Streams werden in dieser Arbeit näher untersucht und Implementierungen von verschiedenen Anbietern verglichen. Unter den Bibliotheken wie RxJava, Akka-Streams und Reactor stellt sich Letzteres als besonders einfach in der Handhabung heraus. Mithilfe dieser Bibliothek wurde anschließend eine reaktive Applikation entwickelt und hinsichtlich der Performance mit einer äquivalenten klassischen, synchronen Variante verglichen, wobei sich die reaktive Applikation als schneller erweist.

Die reaktive Programmierung ist folglich ein Begriff, der sowohl in der Systemebene als auch in der Anwendungsebene vertreten ist. Unter den Möglichkeiten in der Anwendungsebene dominiert Reactive Streams, die in vielen Programmiersprachen und Implementierungen angeboten werden. Reactive Streams erlauben die Entwicklung von Anwendungen, die einen bedeutend höheren Durchsatz erzielen. Aus architektonischer Sicht ermöglicht die reaktive Programmierung Systeme zu entwickeln, die schnell, skalierbar und fehlertolerant sind und den Unternehmen eine Steigerung von Umsatz und Produktivität bescheren.

Inhaltsverzeichnis

Abbildungsverzeichnis	v
Tabellenverzeichnis	vi
Listingverzeichnis	vii
Glossar	viii
1 Einleitung	1
2 Reaktivität in Software	3
3 Reaktivität auf der Systemebene	5
3.1 Einführung	5
3.2 Das reaktive Manifest	6
4 Reaktivität auf der Anwendungsebene	8
4.1 Einführung	8
4.2 Reactive Streams	9
4.2.1 Überblick	9
4.2.2 Back Pressure	11
4.2.3 Streams oder Futures	12
4.2.4 Collections API und Reactive Streams	13
4.2.5 Hot und Cold Streams	14
4.3 Reaktive Programmierung in reaktiven Systemen	14
5 Evaluierung	16
5.1 Wahl der zu betrachtenden Frameworks	16
5.2 Bibliotheken	18
5.2.1 Methodiken	18
5.2.2 RxJava	19
5.2.3 Reactor	27
5.2.4 Akka-Streams	32

5.3	Bewertung	38
5.4	Frameworks	39
5.4.1	Spring Webflux	39
5.4.2	Akka	39
5.4.3	Eclipse Vert.x	40
6	Einsatzszenarien von reaktiver Programmierung	42
6.1	Netflix	42
6.2	PayPal	42
6.3	Verizon	43
7	Konzeption einer reaktiven Anwendung	45
7.1	Anforderungen	45
7.2	Die Wahl der Technologien	46
8	Umsetzung	47
8.1	Die Umsetzung in RSocket und Reactor	47
8.1.1	Der RSocket Server	49
8.1.2	Der RSocket Client	51
8.2	Die Umsetzung in Socket	53
8.3	Performance Benchmark	58
8.3.1	Die Testumgebung	58
8.3.2	Der Benchmark	58
8.3.3	Simulation in Scala	60
8.3.4	Auswertung der Testergebnisse	62
9	Fazit	69
	Anhang	71
A.1	Quellcode	71
A.1.1	Reactive Server Implementierung	71
A.1.2	Reactive Client Implementierung	73
A.1.3	Socket Server Implementierung	75
A.1.4	Socket Client Implementierung	77
A.1.5	Test der reaktiven Variante	81
A.1.6	Test der synchronen Variante	83
	Quellenverzeichnis	86

Abbildungsverzeichnis

4.1	Lebenszyklus eines Reactive Streams	10
5.1	Jaxcenter Umfrage: Application Frameworks von 2016-2018	17
5.2	Überblick der reaktiven Bibliotheken und Frameworks	18
5.3	Erhebung der Daten auf StackOverflow	25
5.4	Akka-Stream Beispiel	33
5.5	Ausschnitt aus der Dokumentation der Funktionen scan und fold	34
5.6	Spring 5 Module	39
6.1	Gegenüberstellung des alten und neuen Systems	44
8.1	Ausschnitt aus der JIT Ausgabe	59
8.2	Messung mit Zeitverzögerung	62
8.3	Reaktive Car und Server Kommunikation mit Sendeverzögerung	64
8.4	Synchrone Car und Server Kommunikation mit Sendeverzögerung	64
8.5	Messung ohne Zeitverzögerung	65
8.6	Reaktive Car und Server Kommunikation ohne Sendeverzögerung	67
8.7	Synchrone Car und Server Kommunikation ohne Sendeverzögerung	67

Tabellenverzeichnis

4.1	Szenarien und empfohlene Technologien	13
5.1	Flow Typen & RxJava Typen (unvollständig)	20
5.2	Ausschnitt aus den Releases	24
5.3	Schnappschuss aktiv diskutierter Fragen (Stand 03.12.2018)	25
5.4	GitHub Statistik	26
5.5	Flow & Reactor (unvollständig)	27
5.6	Versionsgleise und Versionsnummern	29
5.7	Ausschnitt der Releases	30
5.8	GitHub Statistik	31
5.9	Schnappschuss aktiv diskutierter Fragen	31
5.10	Flow & Akka-Streams	32
5.11	Auszug aus den Releases von Akka und Alpakka	36
5.12	GitHub Statistik	37
5.13	Schnappschuss aktiv diskutierter Fragen	37
5.14	Gewichtete Evaluationsmatrix	38
5.15	Unvollständige Liste der Akka Module	40
8.1	Klassen aus dem eigenen Prototype-Package	48
8.2	Klassen aus dem java.net Package	54
8.3	Messung mit Zeitverzögerung (tabellarisch)	63
8.4	Messung ohne Zeitverzögerun (tabellarisch)	66

Listingverzeichnis

5.1	Synchron: RxJava Hello World	21
5.2	Ausgabe vom synchronen Code	22
5.3	Asynchron: RxJava Hello World (asynchron)	22
5.4	RxJava Hello World debug	23
5.5	Ausgabe	23
5.6	Synchron: Reactor Hello World!	28
5.7	Asynchron: Reactor Hello World!	29
5.8	Akka Streams Hello World	35
8.1	Reactive Server: Relevante Imports	49
8.2	Reactive Server: Receive-Methode aus [async.server.Server	50
8.3	Reactive Car: Relevante Imports	51
8.4	Reactive Car: Connect-Methode aus [async.client.Car	52
8.5	Send Methode aus [async.client.Car	52
8.6	Close Methode aus async.client.Car	53
8.7	Receive Methode aus sync.server.Server	55
8.8	Connect Methode aus sync.client.Car	56
8.9	Send Methode aus sync.client.Car	56
8.10	SendData Methode aus sync.client.Car	57
8.11	Close Methode aus sync.client.Car	57
8.12	Run Methode aus der AsyncSimulation Klasse	60
8.13	Warmup Methode aus der AsyncSimulation Klasse	60
8.14	Benchmark Methode aus der AsyncSimulation Klasse	61
8.15	Main Methode aus der AsyncSimulation Klasse	61
A.1	Reactive Server Implementation	71
A.2	Reactive Client Implementation	73
A.3	Socket Server Implementation	75
A.4	Socket Client Implementation	77
A.5	Scalatest der reaktiven Variante	81
A.6	Scalatest der synchronen Variante	83

Glossar

Antwortzeit	Die Zeit die eine Anwendung braucht um auf eine Anfrage zu antworten.
-------------	---

Asynchronität	Die Auslagerung einer Task, die zeitintensiv ist, auf einem anderen Thread, um blockierendes warten zu vermeiden. z. B. um die UI bei einem Datenbank Request nicht zu blockieren.
---------------	--

Delegation	Die Weiterleitung der Verantwortung über Systemkomponenten im Fehlerfall.
------------	---

Homogen	Aus gleichartigem zusammengesetzt.
---------	------------------------------------

Interoperabilität	Zusammenarbeitsfähigkeit.
-------------------	---------------------------

Konversionsrate	Eine Rate die die Anzahl an Umwandlungen von Interessent zu Kunde beschreibt.
-----------------	---

Replikation	Die Vervielfältigung eines Dienstes z. B. durch Docker.
-------------	---

Ressourcen	Limitierte Softwareeinheiten wie z. B. Micro-services, Threads u. v. m..
------------	--

1

Einleitung

In Anwendungen ist das Paradigma der reaktiven Programmierung nicht neu, doch hält es derzeit vermehrt Einzug in das Java-Ökosystem und etabliert sich zunehmend als Standard. Die Firma doubleSlash möchte in dieser Arbeit näher untersuchen lassen wie und an welcher Stelle die reaktive Programmierung von Vorteil ist, wann sie Sinn ergibt und wann nicht.

Da reaktive Programmierung nicht nur in höheren Programmiersprachen vertreten ist, sondern auch in Systemarchitekturen, kann doubleSlash die Erkenntnisse dieser Arbeit für künftige Projekte in Zusammenarbeit mit anderen Firmen nutzen, um schnelle und skalierbare Software zu entwickeln.

Da sich die reaktive Programmierung immer größerer Beliebtheit erfreut, muss sich doubleSlash ebenfalls mit dieser Technologie beschäftigen, um auch künftig konkurrenzfähig zu bleiben. Reaktive Systeme können langfristig die klassischen Request-Response Modelle ergänzen, da die Menge der Daten im Zeitalter von Big Data immer weiter zunimmt.

Der Begriff der reaktiven Programmierung wird in modernen Applikationen häufig genutzt. Als problematisch erweist sich für das Unternehmen doubleSlash bislang das Fehlen von klaren Vorstellungen zu Einsatzszenarien und Möglichkeiten der reaktiven Programmierung.

Ziel dieser Arbeit ist es daher, das Thema der reaktiven Programmierung ausführlich aus einer technologischen Perspektive heraus zu diskutieren und zu evaluieren, um Entscheidungen bezüglich künftiger Einsätze in Projekten zu vereinfachen.

Die Thesis gliedert sich in neun Kapitel. Die Kapitel **zwei**, **drei** und **vier** befassen sich mit grundlegenden Informationen zur Reaktivität und reaktiver Programmierung.

In Kapitel **fünf** werden reaktive Technologien evaluiert und gewichtet. Das anschließende Kapitel **sechs** diskutiert Einsatzszenarien reaktiver Technologien anhand von Erfolgsgeschichten von Unternehmen wie Netflix und Paypal.

In Kapitel **sieben** und **acht** wird eine reaktive Anwendung prototypisch in zwei Varianten umgesetzt und getestet: zum einen in einer reaktiven- und zum anderen in einer klassischen, blockierenden Variante.

Das Fazit in Kapitel **neun** fasst die Ergebnisse dieser Untersuchung zusammen.

2

Reaktivität in Software

Grundsätzlich beschreibt Reaktivität einen Vorgang, bei dem etwas auf einen Reiz oder Impuls reagiert. Im Kontext der Softwareentwicklung bedeutet dies, dass eine Anwendung auf Reize in Events respektive Signale reagiert. Um den Begriff Reaktivität zu differenzieren, wird zwischen Anwendungsebene und Systemebene unterschieden.¹

Die Reaktivität in der Systemebene meint die Steuerung und Verwaltung der **Resourcen** in verteilten Systemen.² Ein Beispiel hierfür sind Microservices, die verteilt werden. Unklar bleibt dabei, was passiert, wenn ein Microservice ausfällt oder wenn die Anzahl der Nutzeranfragen stark ansteigt. Der Service muss auf solche Änderungen reagieren und schnelle **Antwortzeiten** liefern. Die Reaktivität meint daher auch die Reaktion auf Last und Fehler. Im Jahr 2013 wurde ein Manifest veröffentlicht, das ein reaktives System anhand von vier Qualitäten beschreibt. In Kapitel 3 wird dieses Manifest näher erläutert.

Die Reaktivität in der Anwendungsebene spiegelt die Reaktion einer Anwendung auf ein oder mehrere Events. Ein Beispiel hierfür ist Java Swing. So lässt sich beispielsweise für jedes JComponent ein Listener implementieren, das auf Mausbewegungen oder Klicks reagiert. Bei Reaktivität in der Anwendungsebene ist von reaktiver Programmierung die Rede. Diese Art der Programmierung befasst sich mit der asynchronen, nicht blockierenden Verarbeitung von Events mittels Datenströmen (Streams). Wie diese Programmierung praktisch umgesetzt ist, wird in Kapitel 4 diskutiert.

In beiden Ebenen ist die Reaktivität definiert, jedoch ist es notwendig, den Unterschied zwischen den Ebenen zu beachten, denn ein Softwarearchitekt versteht unter der Reaktivität etwas anderes, als ein Anwendungsentwickler. Allerdings sind die Gren-

¹vgl. Malawski, Abs. 2 [Mal17]

²vgl. Bonér & Klang, 2017, S. 15 [BK17]

zen zwischen diesen Ebenen fließend. So können Teile eines reaktiven Systems mit reaktiven Programmiertechniken implementiert werden.

Aktuelle Frameworks, mit denen sich reaktive Systeme bauen lassen, werden im Kapitel 5.4 erläutert. Die Vorteile von reaktiven Systemen werden in Erfolgsgeschichten von bekannten Unternehmen in Kapitel 6 gezeigt.

Da der Fokus dieser Thesis darauf liegt, zu ermitteln, wann reaktive Programmierung sinnvoll ist und was für Möglichkeiten sie mit sich bringt, werden zunächst reaktive Programmiertechniken in Kapitel 4 vorgestellt und anschließend die zugehörigen Bibliotheken in Kapitel 5 evaluiert.

3

Reaktivität auf der Systemebene

3.1. Einführung

Es folgen die drei Hauptgründe dafür, weshalb Reaktivität in Softwaresystemen zunehmende Bedeutung erhält:

1. Big Data: Die stetig steigende Datenmenge befindet sich inzwischen im Petabyte-Bereich. ¹
2. Applikationen laufen heutzutage nicht nur auf Smartphones, sondern auch in Clustern mit der Leistung von tausenden von Multikernprozessoren. Dieser hohe Grad an Verteilung muss entsprechend verwaltet werden. ¹
3. Die Nutzer erwarten Antwortzeiten im Millisekunden Bereich und einen andauernden Betrieb ohne Aussetzer. ¹

Zieht man in Betracht, dass Software auch in Kleinstgeräten integriert ist (IoT), nimmt die zu verarbeitende Datenmenge weiter zu. ¹ Solche Hürden sind nicht mit traditionellen Software-Architekturen vereinbar. Nicht reaktive Systeme können solche Datenmengen nicht effizient verarbeiten, ohne einen hohen Grad an Komplexität in ein Projekt zu bringen. So müssen beispielsweise Fehler geeignet behandelt und Komponenten neu gestartet werden, was Aussetzer (Down Times) verursacht. In reaktiven Systemen dagegen werden Anfragen an eine defekte Komponente - ohne dass es dem Nutzer auffällt - zu einer funktionierenden Komponente delegiert, während die defekte Komponente im Hintergrund wiederhergestellt wird.

¹vgl. Urma et al., S. 417 [[Rao18a](#)]

Im Jahr 2013 stellte eine Gruppe von Programmierern das reaktive Manifest vor, das notwendige Qualitäten für ein System beschreibt in dem kurze Antwortzeiten, unter hohen Lasten und Fehlern gewährleistet werden.

3.2. Das reaktive Manifest

Das reaktive Manifest definiert vier Qualitäten, die ein reaktives System ausmachen.

Das oberste Ziel eines reaktiven Systems ist die **Antwortbereitschaft**. Es gilt, sie im Zeichen der Qualität so hoch wie möglich zu halten, denn dies schafft ein besseres Nutzererlebnis. Eine hohe Antwortbereitschaft fördert weitere Interaktionen des Nutzers mit der Software, da nicht unnötig gewartet werden muss.² Um eine hohe Antwortbereitschaft zu erreichen, benötigt es eine nachrichtenorientierte, elastische und widerstandsfähige Architektur.

Die **Elastizität** ist die Einhaltung der Antwortbereitschaft unter sich wechselnden Lastbedingungen. So muss beispielsweise das System unter einer hohen Anzahl an Requests dieselbe Antwortzeit einhalten wie unter einer geringen Anzahl. Dies wird durch die dynamische **Replikation** von Diensten ermöglicht. Das heißt, dass das reaktive System in der Lage sein muss, die aufkommende Last zur Laufzeit zu erkennen, um die Dienste zu **replizieren**.² Ein gutes Beispiel dafür, die Elastizität zu erreichen, ist die Containersoftware Docker mit der Verwaltungssoftware Kubernetes zur Orchestrierung. Hierbei können die Dienste in separaten virtuellen Softwareumgebungen repliziert werden.

Kein System kann fehlerfrei gebaut werden. Die **Widerstandsfähigkeit** beschreibt vor diesem Kontext, wie auf Fehler reagiert wird. Um auf Software- und Hardwareausfälle zu reagieren, müssen die Dienste replizierbar (Elastizität), vollständig isoliert und **delegierend** sein. Die Isolation von (replizierten) Diensten hat den Vorteil, dass die restlichen Softwarekomponenten im Fehlerfall unberührt bleiben und weiter ausgeführt werden können. Des Weiteren müssen aufgefallene Dienste Ihre Verantwortung an überliegende Dienste delegieren, die im Anschluss den vorherigen Zustand wiederherstellen können.² Die Delegation lässt sich beispielsweise mit Kubernetes realisieren.

Die Kommunikation zwischen Diensten sowie die Delegation und Isolation müssen über eine **nachrichtenorientierte** Kommunikation erfolgen. Dies erlaubt erst die Isolation der Dienste und damit einhergehend die Elastizität und Widerstandsfähigkeit. Nachrichten ermöglichen eine Verteilung der Dienste und ebenso eine sprachunabhängige Formulierung des Programms.² So können Programmiersprachen beispielsweise

²vgl. Bonér et al., Das reaktive Manifest, 2014 [al14]

entsprechend ihrer Eigenschaften genutzt werden. Berechnungen u. s. w können in Sprachen wie C, C++ oder Rust geschrieben werden, während die grafische Oberfläche oder Datenbankenschnittstellen in Java und Scala geschrieben werden können. Beispiele für nachrichtenorientierte Kommunikation sind SOAP und das Aktorenmodell.

Zusammengefasst bildet die nachrichtenorientierte Kommunikation die Basis eines reaktiven Systems, denn sie gewährt eine elastische und widerstandsfähige Architektur und somit die Antwortbereitschaft. Die Qualitäten bringen Vorteile wie Isolation, Replikation und Delegation und ermöglichen es den Entwicklern, sich mehr auf die eigentliche Anwendung zu konzentrieren. Die Anwendung wiederum kann reaktiv implementiert werden, jedoch muss Reaktivität hier anders verstanden werden: als Reaktivität auf Anwendungsebene.

4

Reaktivität auf der Anwendungsebene

4.1. Einführung

Reaktivität auf der Anwendungsebene bedeutet die reaktive Programmierung einer Anwendung oder Teile einer Anwendung. Per definitionem beschreibt die reaktive Programmierung einen Programmierstil, in dem ein Problem in Teilschritte zerlegt wird, wobei jeder Teilschritt eventbasiert, **asynchron** und nicht-blockierend verarbeitet werden kann. ¹

Als Beispiel dient hier das Programm Excel. Hier ist eine Summenzelle definiert, die die Summe aus dem Inhalt einer Spalte berechnet. Wenn nun eine Zelle der Spalte sich in ihrem Wert ändert, wird ein Event angestoßen, sodass die Summenzelle die Summe neu berechnet. ²

Der Hauptgrund für die reaktive Programmierung ist, dass blockierendes Warten durch asynchrone Verarbeitung vermieden wird. Durch die Auslagerung der Operationen auf einen anderen Thread wird zum einen der Mainthread nicht blockiert, was zur Folge hat, dass die Anwendung nach wie vor bedienbar bleibt. Zum anderen können diese Threads auf Multikernprozessoren parallelisiert werden und die Gesamtauslastung somit optimieren. ³

Konzepte, die die reaktive Programmierung umsetzen sind: Reactive Streams, Futures & Promises und Dataflow-Variablen (beispielsweise Excel). ³

¹vgl. Jog, S. 14 [[Jog17](#)]

²vgl. Wikipedia, 2018 [[Wik18a](#)]

³vgl. Bonér und Klang, S. 6-8 [[BK17](#)]

Die nachfolgenden Kapitel befassen sich mit der Umsetzung der reaktiven Programmierung sowie der Frage, wie sich die reaktive Programmierung von den reaktiven Systemen abgrenzt.

4.2. Reactive Streams

4.2.1. Überblick

Reactive Streams bezeichnen einen Standard für asynchrone Verarbeitung mit Back Pressure. Im Jahr 2013 entwickelten Mitarbeiter von u. a. Netflix, Lightbend und Pivotal diese Standardisierung. Die Standardisierung definiert Java Interfaces und bietet ein Technology Compatibility Kit zur Kompatibilitätsprüfung eigener Reactive Streams Implementierungen an.⁴

Die Reactive Streams entstanden aus dem Problem heraus, dass in asynchronen Anwendungen Daten stocken können. Dies passiert genau dann, wenn Daten schneller zur Verfügung stehen, als sie verarbeitet werden. Das wäre zum Beispiel der Fall, wenn Echtzeitdaten über einen Stream von einer langsamen Berechnung gestaut werden. Meist hat der verarbeitende Teil einen Puffer, um mit solchen Problemen umzugehen. Allerdings kann dieser Puffer überlaufen und somit einen Datenverlust verursachen. Um dieses Problem zu lösen, definiert die Reactive Streams Specification, wie Informationen mittels Back Pressure übertragen werden. Back Pressure ermöglicht eine regulierte Übertragung der Daten.⁵ Im Kapitel 4.2.2 wird Back Pressure ausführlicher behandelt.

Die definierten Java Interfaces müssen nach der Spezifikation implementiert werden, um Back Pressure zu ermöglichen. Die Spezifikation ist eine Sammlung von Regeln für Implementierer. Mit der Version 9 von Java wurden die Reactive Streams Interfaces ins eigene Repertoire aufgenommen und befinden sich im Modul `java.base` im Package `java.util.concurrent`. Die Interfaces entsprechen exakt den Interfaces der Reactive Streams Specification.⁵

⁴vgl. Wikipedia, 2018, Reactive Streams [Wik18c]

⁵vgl. Reactive Streams, The Problem; Working Groups []

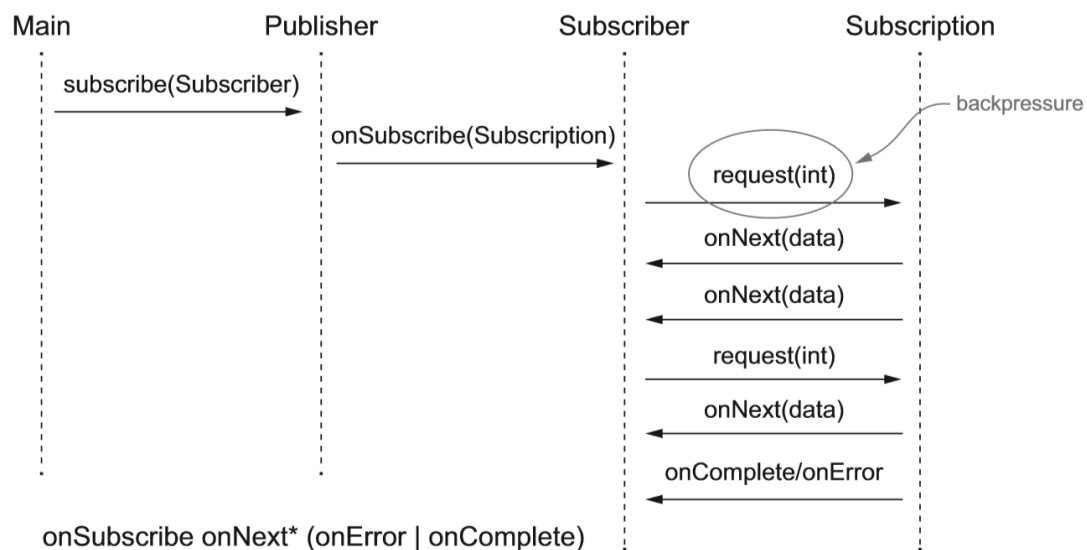
Prinzipiell besteht das Set aus vier Interfaces.

1. Ein Publisher, der den eigentlichen Stream definiert.
2. Ein Subscriber, der die Daten aus dem Stream empfängt.
3. Eine Subscription, die einen Subscriber einem Publisher zuordnet und den Datentransfer umsetzt.
4. Ein Processor, der die Eigenschaften eines Publishers und eines Subscribers in sich vereint.

6

Das folgende Sequenzdiagramm zeigt den Lebenszyklus eines Reactive Streams, wobei nur die Beziehung zwischen Publisher und Subscriber betrachtet wird.

Abbildung 4.1: Lebenszyklus eines Reactive Streams



7

Innerhalb des Mainthreads wird die Subscribe-Funktion, mit dem Subscriber als Argument, beim Publisher aufgerufen. Der Publisher ruft auf dem übergebenen Subscriber die Funktion `onSubscribe` auf, wobei die eine Subscription-Instanz des Publishers als Argument übergeben wird. Der Subscriber fordert eine bestimmte Anzahl an Datensätzen über die Request-Methode an. Die Subscription-Instanz ruft für die Anzahl

⁶vgl. Github, Specification [Git17]

⁷vgl. Urma et al., S.424 [Rao18a]

an benötigten Daten die `onNext`-Methode des Subscribers auf. Dies geschieht, solange die Subscription keine `onComplete`- oder `onError`- Methode aufruft, denn damit ist die Subscription offiziell beendet. Die `Request`-Methode spielt hierbei eine besondere Rolle. Über diese lässt sich der Back-Pressure-Mechanismus realisieren, was im Kapitel 4.2.2 genauer betrachtet wird.

Anbieter wie z. B. RxJava, Reactor oder Akka implementieren die Reactive Streams Specifications. So heißen die Reactive Streams in RxJava `Flowable`, in Akka-Streams `Source` und in Reactor `Flux`. Eine eingehendere Behandlung erfahren diese Begriffe in Kapitel 5.

4.2.2. Back Pressure

Unter Back Pressure versteht man die Regulierung des Datenflusses. In asynchronen Anwendungen kann es häufig zu einem Stau von Daten kommen, wenn beispielsweise der Publisher schneller Daten produziert als der Subscriber sie verarbeitet. Es gibt prinzipiell zwei Möglichkeiten, einen solchen Stau ohne Back Pressure zu vermeiden.⁸

1. Der Subscriber reagiert auf den Stau und ordnet die Daten in eine Warteschlange ein.⁸
2. Der Publisher sendet kontinuierlich eine begrenzte Menge an Daten, sodass der Subscriber die Menge staufrei verarbeiten kann.⁸

Bei Möglichkeit 1 kommen zwei Probleme auf. Wenn erstens die Größe der Warteschlange unbegrenzt ist, wird der Hauptspeicher irgendwann voll sein und einen Speicherüberlauf auslösen. Zweitens kann eine limitierte Größe der Warteschlange Datenverlust verursachen, wenn die Schlange voll ist.⁸

Bei Möglichkeit 2 kann es passieren, dass die Leistung des Subscribers nicht vollständig ausgenutzt wird, wenn der Publisher zu wenig Daten produziert.⁸

Back Pressure ermöglicht Überlastschutz über eine dynamische Anpassung des Datenflusses. Die Umsetzung gelingt über einen Push-Pull-Mechanismus. Push bezeichnet hier die Propagierung von Daten. Pull bezeichnet hingegen das Abholen von Daten. So wäre ein Push sinnvoll, wenn der Subscriber schneller als der Publisher ist, und ein Pull im gegenteiligen Fall.⁸

Der Mechanismus des Back Pressure funktioniert so, dass zu Beginn davon ausgegangen wird, dass der Subscriber die Daten genauso schnell verarbeitet, wie der Publisher sie bereitstellt. Entsteht nun ein Stau, wechselt der Betrieb von Push zu Pull,

⁸vgl. Grammes & Schaal, 2015, S.44-45 [Dr 15]

was bedeutet, dass der Subscriber nun die Datenmenge anfordert, die er auch verarbeiten kann. So wird stets garantiert, dass zum einen kein Stau entsteht und zum anderen der Subscriber voll ausgelastet ist.⁸

4.2.3. Streams oder Futures

Reactive Streams sind eine Spezifikation mit der Funktion, Informationen asynchron, nicht-blockierend und regulierend (Back Pressure) zu übertragen. Die Idee zur asynchronen, nicht-blockierenden Verarbeitung ist nicht neu und wurde bereits mit anderen Konzepten umgesetzt.

Futures sind Technologien die, wie Reactive Streams, reaktive Programmierung umsetzen. Hierbei sind Futures ein Feature, das es bereits seit der Java-Version 5 gibt und im Package `java.util.concurrent` zu finden ist. Das Konzept dabei ist, eine Task asynchron durchzuführen, um blockierendes Warten zu vermeiden. Beim Aufruf wird direkt ein Future, also eine Referenz auf ein Ergebnis, geliefert, während die Task asynchron verarbeitet wird. Das Problem hierbei ist, dass einerseits der Status des Futures abgefragt werden muss, um zu erfahren, ob ein Ergebnis vorliegt. Andererseits kann es passieren, dass die asynchrone Task das Ergebnis nicht berechnet (Fehlerfall) und das Ergebnis somit nicht mehr verfügbar ist. Des Weiteren stellt es sich als schwer heraus, Ergebnisse von Futures aneinander zu reihen, da jedes Future wiederum auf Verfügbarkeit geprüft werden muss.⁹

Seit der Java Version-8 wurden die `CompletableFuture` (`java.util.concurrent`), als konzeptuelle Erweiterung eingeführt. Die `CompletableFuture` können, im Gegensatz zu den Futures, verknüpft werden. Wenn eine asynchrone Operation von dem Ergebnis einer anderen abhängt, muss nun nicht-blockierend gewartet werden, bis das Ergebnis vorliegt, sondern kann direkt weitergeleitet werden.⁹

Sowohl Reactive Streams als auch Futures haben das Ziel, Tasks asynchron und nicht blockierend auszuführen. Der wesentliche Unterschied dieser Konzepte ist, dass Futures immer ein Element verarbeiten. So können zum Beispiel mehrere HTTP-Requests je in einem Future gemacht werden, während ein Stream keine theoretische Begrenzung hat. Des Weiteren sind Reactive Streams in der Lage, mittels Back Pressure Datenstaus zu vermeiden.

Die folgende Tabelle¹⁰ zeigt die Kardinalität von Technologien im asynchronen und synchronen Kontext:

⁹vgl. Urma et al., S.390 [Rao18b]

¹⁰vgl. Nurkiewicz und Christensen, Cardinality [Tom17a]

Tabelle 4.1: Szenarien und empfohlene Technologien

	Ein Datensatz	Viele Datensätze
Synchron	Objekt	Iterable, Java Streams
Asynchron	Callback, Future, CompletableFuture	Reactive Stream, Callback

Für einfache synchrone Verarbeitungen mit einem Datensatz können simple Getter-Methoden genutzt werden, um das jeweilige Objekt zu erhalten. Falls viele Daten synchron verarbeitet werden müssen, eignen sich Iterable Typen respektive Streams. Im asynchronen Fall bieten sich Futures an. Selbst wenn es mehrere asynchrone Requests braucht, um eine Tasks zu verarbeiten, können dennoch CompleteableFutures verwendet werden. Bei vielen, asynchronen Datensätzen eignen sich vor allem die Reactive Streams.

4.2.4. Collections API und Reactive Streams

Der Begriff Stream kommt nicht nur in der reaktiven Programmierung vor, sondern auch in Java als native Bibliothek. Die Collections (in `java.util.stream`) unterstützen Streams als zusätzliche Abstraktionsebene, um deklarativ Informationen zu verarbeiten. Der so entstehende Vorteil ist, dass nun auf den imperativen Code von iterativen Schleifen verzichtet werden kann, was den Code allgemein wartbarer und verständlicher macht. Allerdings sind die Java Streams auf bestimmte Szenarien optimiert. Wie die Tabelle 4.1 zeigt, liegt die Stärke der Streams darin, Informationen synchron zu verarbeiten. Zwar ist es möglich, asynchrone Tasks zu verarbeiten, jedoch bietet die Streams-API keine Möglichkeiten an die asynchronen Ergebnisse zu sortieren und zu synchronisieren. Vor allem in Situationen, in denen die asynchronen Tasks Latenzen haben oder undefinierte Zeitabstände vorweisen. Dies müsste manuell gelöst werden und würde die Komplexität des Codes stark erhöhen. Zudem gibt es keinen Mechanismus, der die Streams vor Überbelastung schützt.

Der Grund dafür, weshalb Java Streams in einen Kontext mit Reactive Streams gebracht werden, ist zum einen der Bezeichner Stream und zum anderen die deklarative Programmierung. Im späteren Teil dieser Thesis werden Implementierungen von Reactive Streams vorgestellt, die sich ähnlich deklarativ programmieren lassen wie die Java Streams. Die Möglichkeit der deklarativen Programmierung ist kein verpflichtendes Feature der Reactive Streams Specification und ist somit auch kein Teil von ihr.

Zusammengefasst liegt der Unterschied in dem Einsatzzweck: Java Streams sind für statische Daten geeignet, die einmal transformiert werden. Reactive Streams dagegen sind für dynamische, asynchrone Daten geeignet.

4.2.5. Hot und Cold Streams

In vielen Implementierungen von Reactive Streams gibt es zwei Arten von Streams. Die Standard-Art ist der Cold Stream. Er reagiert mit dem Versenden der Daten, wenn eine Subscription erstellt wurde. Diese Art der Reaktion wird auch als lazy evaluation bezeichnet. Eine weitere Besonderheit ist, dass ein Cold Stream immer die gesamten Informationen für jede Subscription versendet. Ein Cold Stream ist somit nicht für Echtzeitdaten geeignet. Dieser Stream-Typ eignet sich besonders dann, wenn Informationen ein definiertes Ende haben, wie z. B. beim Inhalt einer Textdatei. ¹¹

Ein Hot Stream dagegen versendet Daten unabhängig von Subscriptions. Das heißt, dass ein Subscriber immer die aktuellen Daten des Streams anstelle des gesamten Datensatzes empfängt. Hot Streams eignen sich dann, wenn die Informationen kein definiertes Ende haben, wie z. B. bei Twitter-Feeds. ¹¹

4.3. Reaktive Programmierung in reaktiven Systemen

Ein reaktives System ist ein Modell, das anhand von Qualitäten ein modernes, antwortbereites System beschreibt. Die reaktive Programmierung ist ein Paradigma, das Datenstrukturen in Streams asynchron, und nicht-blockierend verarbeitet. Beide Kontexte lassen sich miteinander vereinen, jedoch eignen sich die meisten reaktiven Bibliotheken nicht dafür, reaktive Systeme zu bauen.

Der Grund dafür ist die eventbasierte Kommunikation. Events haben keinen klaren Empfänger und stellen lediglich Fakten oder Signale dar. Die Widerstandsfähigkeit und Elastizität sind nicht ohne Weiteres realisierbar, da eine nachrichtenorientierte Kommunikation notwendig ist (siehe Kapitel 3.2). Ein isolierter, ausgefallener Service muss

¹¹vgl. Davis, Hot and Cold [Dav19]

adressierbar sein; falls er das nicht ist, weiß das System nicht, welcher Service wiederhergestellt werden soll.

Reaktive Programmiertechniken sind auf zeitlicher Ebene gut skalierbar, das heißt, sie sind für Nebenläufigkeit optimiert, während reaktive Systeme auf örtlicher Ebene skalierbar und daher verteilbar sind. Um ein reaktives System aus reaktiven Programmiertechnologien wie Streams und Futures zu bauen, benötigt es somit ergänzende Technologien wie z. B. Nachrichtenbusse, Fehlererkennungstechnologien (bspw. Hystrix) und Replikationstechniken (bspw. Docker). ¹²

Ein prominentes Beispiel für die Verwendung von reaktiver Programmierung in reaktiven Systemen ist das Akka Toolkit. Die interne Verarbeitung der Daten und die Weiterleitung an Operatoren wird über eine Reactive-Streams-Implementierung realisiert. Das [Akka Toolkit](#) enthält noch weitere Technologien wie z. B. eine Implementierung des Aktorenmodells über die sich Nachrichten zwischen verschiedenen Diensten austauschen lassen. Es ist möglich, vollständige reaktive Systeme mit dem Toolkit zu bauen, da notwendige Technologien für die Streams bereits integriert sind. Anders als bei reinen Streams-Implementierungen wie RxJava und Reactor werden somit keine externen Abhängigkeiten (wie z. B. Message Bus und Verteilungssoftware) benötigt, was die [Homogenität](#) des gesamten Systems erhöht.

Es bietet sich an, die Dienste eines reaktiven Systems mit reaktiven Programmiertechniken zu bauen, denn auf der Anwendungsebene ist die reaktive Programmierung eine Alternative zu synchronen und blockierenden Programmen. In Kapitel 8 wird eine Anwendung reaktiv und klassisch implementiert, was im Anschluss in Performancetests miteinander verglichen wird.

¹²vgl. Benedikt Stemmildt [[Ste17](#)]; Bonér und Klang, S. 15 [[BK17](#)]

5

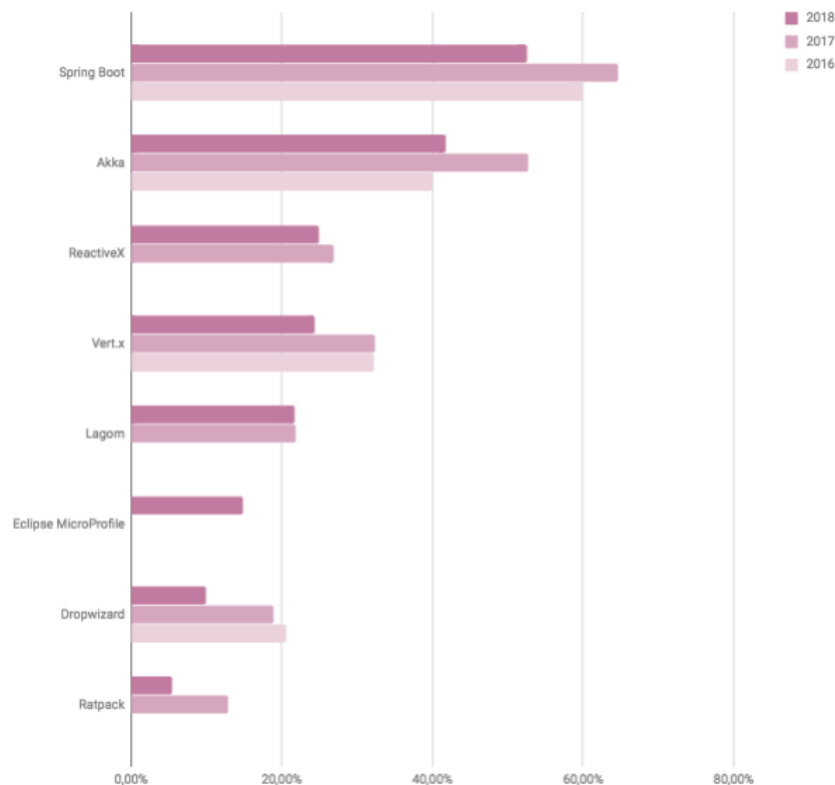
Evaluierung

In diesem Kapitel werden reaktive Bibliotheken evaluiert und Frameworks betrachtet, die diese Bibliotheken nutzen. Es werden Kriterien evaluiert, die für das Unternehmen doubleSlash essentiell sind. Am Ende der Evaluierung werden die Kriterien und die Bewertung in einer gewichteten Evaluationsmatrix zusammengefasst. Die Gewichtung richtet sich nach den Prioritäten des Unternehmens. Die Matrix gilt als Ausgangspunkt für die Wahl der Bibliothek bei der Konzeption und Entwicklung der prototypischen Anwendung.

5.1. Wahl der zu betrachtenden Frameworks

Konkret umfasst die Evaluierung drei reaktive Bibliotheken. Die Wahl der Bibliotheken basiert auf einer Umfrage der Seite jaxcenter.de. Die folgende Abbildung zeigt eine Umfrage zu Application Frameworks.

Abbildung 5.1: Jaxcenter Umfrage: Application Frameworks von 2016-2018



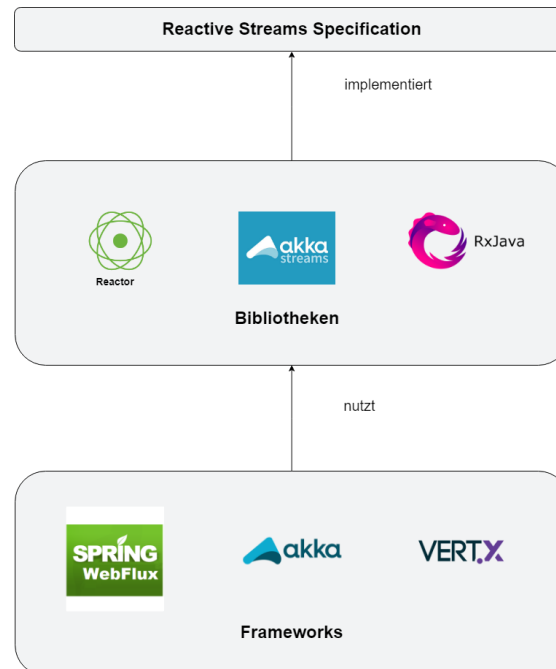
1

Gemäß den Technologietrends 2018 werden die Top 3 Spring-Boot, Akka und Vert.x betrachtet. Die korrespondierenden reaktiven Bibliotheken sind **RxJava**, **Reactor** und **Akka-Streams**. ReactiveX wird nicht als vollwertiges Framework wie Akka oder Spring betrachtet. Der Grund hierfür ist, dass ReactiveX sich selbst als API bezeichnet und nicht den technischen Umfang der anderen Frameworks aufweist. ReactiveX wird in vielen Programmiersprachen implementiert, weshalb die Verbreitung der ReactiveX API in Frontend (RxJs) und Backend (RxJava) stattfindet. Da Spring-Boot einen großen Umfang an Modulen besitzt, wird speziell das reaktive Web Framework WebFlux betrachtet.

¹Schlosser und Richters, JAXenter-Survey: Application Frameworks - 2018, 2017, 2016 [SR]

Die folgende Abbildung zeigt einen Überblick über die bekanntesten reaktiven Frameworks/Toolkits und die dazugehörigen Bibliotheken.

Abbildung 5.2: Überblick der reaktiven Bibliotheken und Frameworks



5.2. Bibliotheken

5.2.1. Methodiken

Die Kriterien werden mittels der folgenden Methodiken evaluiert. Hierbei entsprechen die Kriterien den Anforderungen der Firma doubleSlash.

Lizenz und Kosten

Es ist zu prüfen ob die Bibliothek eine kommerzielle Nutzung zulässt und ob gegebenenfalls Kosten entstehen.

Dokumentation

Hierbei wird untersucht ob die Dokumentation umfassend, korrekt und aktuell ist.

Handhabung der Bibliothek

Es soll bewertet werden ob Programme kurz, prägnant und verständlich geschrieben werden können und ob eine falsche Bedienung verhindert bzw. erschwert wird.

Weiterentwicklung

Es werden hierbei die Release-Daten untersucht und ermittelt ob Upgrades, Updates und Bugfixes regelmäßig erscheinen.

Verbreitung

Die Ermittlung der Verbreitung wird zum einen über die Popularität auf GitHub verglichen und zum anderen über die Präsenz in der Entwickler-Community StackOverflow.

Die Bewertung der Kriterien beläuft sich auf ein Punkteschema. Jedes Kriterium wird zu Beginn mit zehn Punkten initialisiert. Für jedes negative Merkmal, dass bei der Betrachtung auffällt, gibt es Punktabzug. Die Menge an Punkten, die abgezogen werden, richten sich nach der Schwere des Merkmals.

5.2.2. RxJava

RxJava steht für Reactive Extension (ReactiveX) Java. Die Reactive Extension wurde 2007 von Microsoft für .NET entwickelt. Die zunehmende Beliebtheit sorgte dafür, dass die API auch für andere Programmiersprachen implementiert wurde wie z. B. Javascript, C#, Scala, Swift, Clojure, C/C++, Python und viele mehr. Das Unternehmen Netflix hat RxJava implementiert und nutzt die Bibliothek für die Interprozesskommunikation und Verbindung verschiedener Services. In der Android Welt erfreut sich RxJava großer Beliebtheit. So nutzen Firmen wie SoundCloud, Square, NYT und Seatgeek die Bibliothek in ihren Apps.²

RxJava gibt es in zwei Versionen. Version eins unterstützt nicht die Reactive Streams Specification und damit weder Back Pressure noch die Interkompatibilität mit anderen Bibliotheken. Version zwei dagegen ist eine Reimplementierung und unterstützt die Reactive Streams Specification. Im Rahmen dieser Thesis ist nur RxJava2 relevant. Die Bibliothek wird ab der Java Version 6 unterstützt.

Die Besonderheit von RxJava ist, dass die Daten der Publisher über Funktionen höherer Ordnung komponiert, gefiltert oder anderweitig transformiert werden können. Dieser funktionale Aspekt ermöglicht eine deklarative Programmierung. Unter dem Kriterium [Handhabung der Bibliothek](#) wird gezeigt wie Nebenläufigkeit auf deklarative Art

²vgl. Nurkiewicz und Christensen, Abs. 10 [[Tom17b](#)]

und Weise realisiert werden kann.

Sowohl die Flow-API als auch RxJava richten sich nach der Reactive Streams Specification. Folgende Tabelle zeigt die Flow-Typen und die RxJava Implementierungen:

Tabelle 5.1: Flow Typen & RxJava Typen (unvollständig)

Flow	RxJava
Publisher	Observable, Flowable
Subscriber	Observer, Single
Processor	Subject
Subscription	Disposable

Eine vollständige Liste der Typen in RxJava sind in der [Javadocs](#) zu finden.

Lizenz und Kosten

RxJava ist Open Source und besitzt die Apache-Lizenz 2.0 und ist daher für den kostenfreien, kommerziellen Gebrauch zulässig.

Da eine kommerzielle Nutzung erlaubt ist und keinerlei Kosten entstehen erhält RxJava zehn Punkte.

Punkte	10
--------	----

Dokumentation

Auf der [Homepage des Projekts](#)³ befindet sich eine [Referenzdokumentation](#), die die grundsätzlichen Typen und Mechanismen erklärt und illustriert. Diese Dokumentation soll primär die Kernkonzepte hinter ReactiveX vermitteln. Es handelt sich hierbei um eine allgemeine Referenz, die nicht programmiersprachenspezifisch ist.

Eine zweite Dokumentation gibt es auf GitHub in Form eines [Wiki](#)⁴. Das Wiki ist gut gegliedert und baut semantisch aufeinander auf. Für Java-Entwickler wird das Wiki für den Einstieg empfohlen, da alle relevanten Themen und weiterführende Informationen dort abgedeckt sind. Allerdings sind die Themen teils unvollständig dokumentiert.

³GitHub, ReactiveX Website [[Maic](#)]

⁴Github, ReactiveX Wiki [[Maia](#)]

Die [Javadocs](#)⁵ lassen sich über das Wiki finden und sind auf dem aktuellen Stand des Releases. Die Dokumentation ist detailliert und enthält sämtliche Beschreibungen von Methoden, Klassen und Interfaces. Des Weiteren gibt es zu jeder Methode illustrierte Beispiele wie sie in der Praxis genutzt werden können.

Es gibt drei Punkte Abzug für die Unvollständigkeit der ReactiveX Referenzdokumentation. Die Überschriften im Wiki sind irreführend. Das Kapitel [reactive Types of RxJava](#)⁶ suggeriert eine Auflistung der Typen, stattdessen wird auf die Referenzdokumentation verlinkt und der Observable Typ erklärt.

Punkte	7
--------	---

Handhabung der Bibliothek

Publisher und Subscriber können in RxJava kurz und prägnant ausgedrückt werden. Allerdings gibt es einige Aspekte, die eine falsche Benutzung ermöglichen. RxJava bietet eine Vielfalt an Typen für Publisher, Subscriber und Processor an. Das Problem ist, dass die große Menge an Typen eine umfangreichere Einarbeitung mit der Dokumentation benötigen. So wird in der Dokumentation als einziger reaktiver Typ das Observable genannt obwohl es weitaus mehr Typen für verschiedene Einsatzzwecke gibt. Das Flowable unterstützt Back Pressure während das Observable das nicht unterstützt. Die Verwechslung dieser Typen kann verheerend für das Programm sein.

Das folgende Listing zeigt ein Publisher (Flowable), welcher nacheinander die Events HELLO, WORLD und ! sendet. Der Subscriber (Observer) wird über zwei Methodenreferenzen und einer Lambda Funktion realisiert, welche implizit für die onNext, onError und onComplete Funktionen stehen.

Listing 5.1: Synchron: RxJava Hello World

```
1 Flowable<String> flowable = Flowable.just("hello", "world", "!")
2                                   .map(String::toUpperCase);
3 flowable.subscribe((next) ->
4                     System.out.println(Thread.currentThread.getName()
5                                         + " " + next), // onNext
6                     System.err::println,           // onError
7                     () -> System.out.println("done")); // onComplete
```

⁵ReactiveX, Javadoc[\[Maib\]](#)

⁶GitHub, ReactiveX Wiki [\[Maib\]](#)

Das Programm erzeugt die folgende Ausgabe:

Listing 5.2: Ausgabe vom synchronen Code

```
1 main HELLO
2 main WORLD
3 main !
4 done
```

Standardmäßig sind alle Publisher Typen (Observable, Flowable u. s. w.) synchron. Damit eine Subscription asynchron läuft, muss ein Threadpool respektiver Scheduler definiert werden, damit dieser auf einem Thread ausgelagert werden kann. Die Konfiguration hierfür ist allerdings simpel, da RxJava ein Set an Scheduler anbietet, die optimal auf das System abgestimmt sind. Das folgende Beispiel zeigt eine asynchrone Variante von [Listing 5.1](#).

Listing 5.3: Asynchron: RxJava Hello World (asynchron)

```
1 Scheduler scheduler = Schedulers.computation();
2 Disposable disposable = Flowable.just("hello", "world", "!")
3   .subscribeOn(scheduler) // Subscription auf einem Thread des Scheduler
4   .map(String::toUpperCase);
5   .subscribe(System.out::println,
6               System.err::println,
7               () -> System.out.println("done"));
8 while (!disposable.isDisposed()) {}
```

Probleme in der Handhabung bei asynchroner Programmierung treten bei der Auswahl der Funktionen auf. RxJava bietet zwei Funktionen zur asynchronen Ausführung an. Zum einen `subscribeOn` (wie in [Listing 5.3](#)) und zum anderen `observeOn`. Mit `subscribeOn` kann definiert werden mit welchem Scheduler die Subscription ausgelagert wird. Das heißt, dass alle Elemente auf einen anderen Thread geschickt werden und dort vom Subscriber über `onNext`, `onError` u. s. w. verarbeitet werden.

Die Funktion `observeOn` dagegen, definiert, auf welchen Thread einzelne Operationen ausgeführt werden können. Das heißt, dass alle Funktionen, die nach dem Aufruf von `observeOn` aufgerufen werden, auf einem anderen Thread laufen.

Das folgende Listing zeigt den Ausschnitt einer asynchronen Variante mit `subscribeOn` und `observeOn`. Über die Funktion `doOnNext` können schrittweise die Elemente ausgegeben werden. In diesem Beispiel wird der aktuelle Thread und das dazugehörige Element ausgegeben.

Listing 5.4: RxJava Hello World debug

```

1 Flowable.just("hello", "world", "!")
2 .map(String::toUpperCase)
3 .doOnNext((element) -> System.out.println(Thread.currentThread()
4                                           .getName()+ " " + element))
5 .subscribeOn(scheduler)
6 .observeOn(scheduler) // Nachfolgende Funktionen auf anderem Thread des
   Scheduler
7 .filter(word -> word.length() > 1)
8 .subscribe((next) -> System.out.println(Thread.currentThread()
9                                           .getName()+ " "+next) ,
10           System.err::println ,
11           () -> System.out.println("done"));

```

Aufgrund dessen, dass das Programm multithreaded läuft, ist das nachfolgende Listing ein Beispiel:

Listing 5.5: Ausgabe

```

1 RxComputationThreadPool-2 HELLO
2 RxComputationThreadPool-2 WORLD
3 RxComputationThreadPool-2 !
4 RxComputationThreadPool-1 HELLO
5 RxComputationThreadPool-1 WORLD
6 done

```

Die Ausgabe zeigt, dass die Verarbeitung des Flowables nicht auf dem Main Thread stattfindet, sondern auf dem RxComputationThreadPool-2 Thread. Die Funktionen, die nach dem Aufruf von `observeOn` folgen, werden auf dem RxComputationThreadPool-1 Thread ausgeführt wie z. B. der Aufruf der Filterfunktion. Das bedeutet, dass es entscheidend ist, wann die `observeOn` Funktion aufgerufen wird. Je nachdem wann die Funktion aufgerufen wird, werden alle folgenden Funktionen auf einem anderen Thread ausgelagert. Bei `subscribeOn` spielt die Reihenfolge keine Rolle, da die Subscription asynchron läuft und somit implizit der gesamte Stream mit allen Funktionen verlagert wird.

Bei der asynchronen Programmierung kann es schnell zur falschen Benutzung kommen, wenn die Anwendungsfälle der Funktionen nicht klar sind. Die freie Benutzung der Threadpools und Scheduler soll eine maximale Transparenz und eine freie Auswahl bezüglich Scheduler bieten. Für Anfänger wirkt das nicht intuitiv und verleitet zur falschen Benutzung.

Insgesamt gibt es zwei Punkte Abzug für die große Anzahl an Typen, die die API unübersichtlichen machen und eine falsche Benutzung der Typen ermöglichen. Des

Weiteren gibt es noch zwei Punkte Abzug für das unkonventionelle Multithreading. Die deklarative Art erleichtert zwar die nebenläufige Programmierung, allerdings muss der Entwickler aufpassen in welchen Kontext, welche Funktion verwendet wird. Ein zu früh platzierter `observeOn`-Aufruf verlagert zu viele Operatoren auf einen anderen Thread und kann ihn überlasten. In der Praxis wird die `observeOn`-Funktion hinter der `Subscribe`-funktion aufgerufen. Somit kann der Entwickler bestimmen, auf welchem Thread der Subscriber die Events verarbeitet.

Punkte	6
--------	---

Weiterentwicklung

Um die Weiterentwicklung einer Bibliothek qualitativ zu bewerten werden Statistiken von dem Repository auf GitHub verglichen.

Der folgende Ausschnitt zeigt die Release-Zyklen des Projekts.

Tabelle 5.2: Ausschnitt aus den Releases

Version	Datum
2.2.4	23.11.2018
2.2.3	23.10.2018
2.2.2	06.09.2018
2.2.1	23.08.2018
2.2.0	29.07.2018
2.1.17	23.07.2018

Es wird jeden Monat mindestens eine neue Version veröffentlicht. Die Nebenversionen werden hingegen unterschiedlich veröffentlicht. Die monatlichen Releases enthalten Bugfixes, Dokumentation und API-Erweiterungen.

Insgesamt wird die Bibliothek aktiv weiterentwickelt und erhält regelmäßige Updates. Das Kriterium ist vollständig erfüllt.

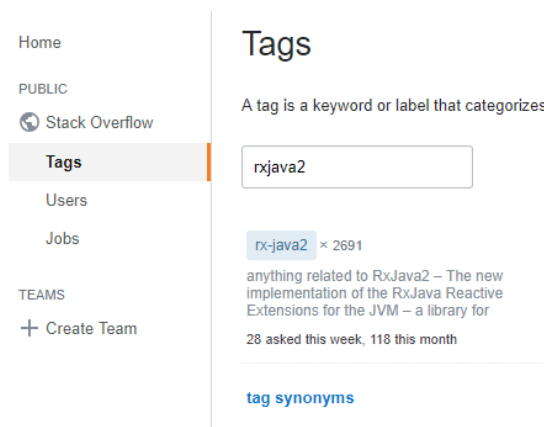
Punkte	10
--------	----

Verbreitung

Um die Verbreitung einer Bibliothek zu messen werden verschiedene Metriken verwendet. Zum einen wird die Anzahl an getaggten Fragen auf StackOverflow verglichen. Die Anzahl sagt aus wie häufig Fragen bezüglich der Bibliothek gestellt werden. Die statistischen Werte auf GitHub sagen aus wie viele Personen sich für die Bibliothek interessieren.

Die folgende Abbildung zeigt wie die Daten auf StackOverflow erhoben werden:

Abbildung 5.3: Erhebung der Daten auf StackOverflow



Die folgende Tabelle zeigt einen Schnappschuss aus StackOverflow⁷, wobei die Anzahl aktiv diskutierter Fragen im Kontext der jeweiligen Bibliothek betrachtet wird:

Tabelle 5.3: Schnappschuss aktiv diskutierter Fragen (Stand 03.12.2018)

Bibliothek	Diskutierte Fragen
RxJava	2628
Reactor	762
Akka Streams	1131

⁷Quelle: [Stab], [Staa], [Staa]

Es zeigt sich, dass RxJava die am meisten diskutierte Bibliothek auf StackOverflow ist.

Die zweite Metrik stützt sich auf statistische Daten der Open Source Plattform GitHub. Hierbei wird die Anzahl an öffentlichen Repositories⁸ sowie die Anzahl an Stars, die die Bibliothek hat, betrachtet. Die Stars eines Projektes sind als Wertschätzung für die Maintainer zu verstehen. GitHub nutzt diese Zahl zum globalen Ranking von Projekten.⁹ Damit wird die Popularität und Bekanntheit innerhalb der Community verstanden.

Tabelle 5.4: GitHub Statistik

	RxJava	Reactor	Akka Stream
Repositories	9005	876	836
Projekt Stars	36772	2761	9331
Ranking	48	3743	955

10

Das Ranking wird über die Seite www.gitstar-ranking.com¹¹ ermittelt.

Die Auswertung zeigt, dass RxJava mit über 8000 Repositories am häufigsten in Teilen von Projekten vorkommt. Des Weiteren hat die Bibliothek selbst über 36600 Stars, was unter den weltweiten Projekten (ca. 28 Millionen öffentliche Repositories)¹² den 48. Platz belegt und somit weit vor Reactor und Akka steht.

Insgesamt ist RxJava auf StackOverflow die am meisten diskutierte Bibliothek. Auf GitHub werden statistisch mehr Projekte mit RxJava erstellt, das Repository der Bibliothek hat weitaus mehr Stars und belegt unter den Open Source Projekten den höchsten Platz. RxJava ist somit die am weitesten verbreitetste reaktive Bibliothek. Es gibt somit keinen Punktabzug.

Punkte	10
--------	----

⁸ Github Quelle zu den Repositories [Gita] [Gitb], [Gitc]

⁹ vgl. GitHub, About Stars [Gita]

¹⁰ Da die Akka-Streams Bibliothek in Scala programmiert ist, wird die Anzahl Repositories unter der Sprache Scala betrachtet.

¹¹ Gitstar Quelle zum Ranking: [gita], [gitb], [gitc]

¹² Wikipedia [Wik18b]

5.2.3. Reactor

Die Bibliothek Reactor wird von Pivotal & Spring vertrieben und implementiert die Reactive Streams Specification. Reactor wurde in Kollaboration mit Spring für das Webflux Framework entwickelt und ist ein Kernbestandteil davon. Aufgrund dessen das WebFlux relativ neu ist, unterstützt Reactor Java ab Version 8. Die folgende Tabelle zeigt eine Gegenüberstellung der Typen.

Tabelle 5.5: Flow & Reactor (unvollständig)

Flow	Reactor
Publisher	Flux, Mono
Subscriber	Subscriber
Processor	FluxProcessor
Subscription	Disposable

Weitere Typen und nähere Beschreibungen sind in der [Javadoc](#)¹³ zu finden.

Lizenz und Kosten

Reactor ist Open Source und besitzt die Apache-Lizenz 2.0 und ist daher für den kostenfreien, kommerziellen Gebrauch zulässig.

Da eine kommerzielle Nutzung erlaubt ist und keinerlei Kosten entstehen, erhält Reactor keinen Abzug.

Punkte	10
--------	----

Dokumentation

Reactor besitzt eine umfangreiche [Referenzdokumentation](#)¹⁴ und eine [Javadoc](#)¹³. Die Referenz deckt grundlegende Fragen zur reaktiven Programmierung ab und führt in die Thematik ein. Des Weiteren wird auf die Installation, Kernkonzepte und weiter führende Themen eingegangen. Insgesamt ist die Referenz informativ und deckt alle Fragen für Neu- und Quereinsteiger ab.

¹³Reactor, Javadoc [[Maif](#)]

¹⁴Reactor, Referenz [[Maie](#)]

Die Javadocs sind auf dem aktuellen Stand des Releases. Es werden in den Javadocs ebenfalls Illustrationen für die Funktionen verwendet.

Aufgrunddessen, dass Reactor eine geringe Anzahl an Typen hat, wirkt die API schlank und übersichtlich und ermöglicht schnelle Designentscheidungen im Bezug auf die Wahl der richtigen Typen.

Insgesamt ist die Referenzdokumentation als auch die Javadocs sehr übersichtlich und klar strukturiert. Die Reactor Bibliothek erhält die volle Punktzahl.

Punkte	10
--------	----

Handhabung der Bibliothek

In Reactor können Publisher und Subscriber kurz und prägnant geschrieben werden. Die API ist der von RxJava sehr ähnlich. Der wesentliche Unterschied liegt darin, dass Reactor weniger Typen anbietet. Im Gegensatz zu RxJava bietet Reactor zwei Publisher-Typen an: Mono und Flux. Beide unterstützen standardmäßig das Back Pressure. Mono und Flux stehen jeweils für eine bestimmte Kardinalität. Mono verarbeitet Events in Anzahl 0 bis 1 während Flux 0 bis n verarbeitet.

Das folgende Beispiel zeigt einen Flux, dass nacheinander HELLO, WORLD und ! emittiert.

Listing 5.6: Synchron: Reactor Hello World!

```
1 Flux<String> flux = Flux.just("hello", "world", "!")
2                       .map(String::toUpperCase);
3 flux.subscribe((next) -> System.out.println(Thread.currentThread().getName() +
4         " " + next),
5         System.err::println,
6         () -> System.out.println("done"));
```

Der Quellcode ist, bis auf die Namen, identisch mit dem Quellcode aus 5.1 in RxJava. Die Ausgabe ist äquivalent zur Ausgabe in 5.2.

Bei der asynchronen Ausführung gibt es nur namentliche Unterschiede. Von der Menge des Codes und von der Semantik ist die Reactorimplementierung identisch mit der asynchronen Variante von RxJava in 5.3.

Listing 5.7: Asynchron: Reactor Hello World!

```

1 Scheduler scheduler = Schedulers.parallel();
2 Disposable disposable = Flux.just("hello", "world", "!")
3     .map(String::toUpperCase)
4     .subscribeOn(scheduler)
5     .subscribe(
6         System.out::println,
7         System.err::println,
8         () -> System.out.println("done"));
9 while(!disposable.isDisposed()) {}

```

Reactor hat, wie RxJava, ebenfalls zwei Methoden zur asynchronen Ausführung. Zum einen `subscribeOn`, welches dem `subscribeOn` in RxJava entspricht und zum anderen `publishOn`, was dem `observeOn` entspricht. Das Problem hierbei ist, wie bei RxJava, dass die Funktionen falsch benutzt werden können.¹⁵

Zusammengefasst ist es relativ schwierig die Bibliothek falsch zu bedienen. Die schlanke API lässt es kaum zu, falsche Typen zu verwenden. Lediglich bei der asynchronen Programmierung können dieselben Probleme auftreten wie bei RxJava, weshalb die Bibliothek diesbezüglich zwei Punkte abgezogen bekommt.

Punkte	8
--------	---

Weiterentwicklung

Das Reactor Projekt wird in GitHub unter drei Versionen vertrieben. Version eins und zwei sind veraltet und werden nicht mehr aktiv weiterentwickelt. Für Version drei nutzen die Maintainer ein dreigleisiges Versionsschema. Die folgende Tabelle die Versionierung von Reactor3.

Tabelle 5.6: Versionsgleise und Versionsnummern

Versions-Gleis	Versionsnummer
Aluminium	3.0.x
Bismuth	3.1.x
Californium	3.2.x

¹⁵vgl. Kapitel 5.2.2

Die Versionsnummern entsprechen der Version der reactor-core Bibliothek. Die Reactor Bibliothek enthält zusätzlich zur Core Bibliothek noch einen Http und TCP Server (Reactor Netty) und Reactor Addons für die Interoperabilität zwischen verschiedenen Bibliotheken. Somit besteht die Reactor Bibliothek aus drei Unterbibliotheken.

Die Aluminium Version wurde seit zwei Jahren nicht mehr aktualisiert. Die Bismuth Version wird noch aktiv weiterentwickelt und erhält regelmäßig Updates. Die Hauptunterschiede zwischen den Nebenversionen sind Features und Namensänderungen. Die Versionsgleise sind somit nicht miteinander kompatibel.

Tabelle 5.7: Ausschnitt der Releases

Version	Datum
3.2.3	23.11.2018
3.2.2	31.10.2018
3.2.1	10.10.2018
3.2.0	21.09.2018
3.1.0	25.09.2017
3.0.0	23.08.2016

Die Nebenversionen (Versionsgleise) werden kontinuierlich in Jahreszyklen veröffentlicht. Revisionen werden monatlich veröffentlicht.

Insgesamt wird die Bibliothek aktiv weiterentwickelt und erhält regelmäßig Updates. Das Kriterium ist vollständig erfüllt.

Punkte	10
--------	----

Verbreitung

Tabelle 5.8: GitHub Statistik

	RxJava	Reactor	Akka Stream
Repositories	9005	876	836
Projekt Stars	36772	2761	9331
Ranking	48	3743	955

Tabelle 5.9: Schnappschuss aktiv diskutierter Fragen

Bibliothek	Diskutierte Fragen
RxJava	2628
Reactor	762
Akka Streams	1131

Gemäß der Tabelle 5.8 erreicht Reactor die niedrigste Platzierung der drei Bibliotheken. Auf StackOverflow erreicht Reactor ebenfalls eine geringere Anzahl an Fragen pro Tag, wie in 5.9 zu sehen ist. Die Auswertung sagt nicht aus, dass die Bibliothek nicht verbreitet ist sondern, dass sie weniger verbreitet ist als Akka-Streams oder RxJava.

Da Reactor Kernbestandteil von Webflux ist, wird es implizit in allen Spring Webflux Anwendungen genutzt. Weiterhin bietet Spring zusätzlich eine Vielzahl an Integration an wie z. B. in Spring Data und Spring Security.

Allerdings ist Reactor eine Bibliothek, die stark an Popularität zunimmt. Der Chefentwickler von RxJava Dave Karnok antwortet auf die Frage, in welchen Situationen Reactor oder RxJava genutzt werden soll mit folgendem Satz:

Use Reactor 3 if you are allowed to use Java 8+, use RxJava 2 if you are stuck on Java 6+ or need your functions to throw checked exceptions. ¹⁶

¹⁶David Karnok, Twitter Posting [Kar]

Insgesamt ist die Reactor Bibliothek, die am wenigsten verbreitete. Ausgehend von drei Plätzen, werden für jeden Platz 3 Punkte vergeben. Der Erste erhält 4 Punkte dem zweiten gegenüber. Somit bekommt die Bibliothek 7 Punkte Abzug.

Punkte	3
--------	---

5.2.4. Akka-Streams

Akka-Streams ist die Stream Bibliothek des Akka Toolkits und wird ab der Javaversion 8 unterstützt. Die Besonderheit der Bibliothek ist, dass die Flow Typen nicht direkt als Schnittstelle angeboten werden sondern ausschließlich über die Implementierungen zur Verfügung stehen. Das heißt, dass die Publisher, Subscriber u. s. w. nicht direkt zugänglich sind. Die Entwickler der Akka-Streams betrachten die Reactive Streams Specification nicht als End-User API sondern als SPI (Service Provider Interface).¹⁷ Das heißt, dass die Schnittstelle ausschließlich für die Entwickler freigegeben wird und nicht für die Nutzer der Bibliothek. Die schwer zu implementierenden Interfaces sollen somit verborgen bleiben und über einfachere Abstraktionen bedienbar sein. Bibliotheken wie RxJava und Reactor bieten auch Abstraktionen zu Publishern und Subscriber an (Observable, Flux), allerdings kann der Entwickler auch eigene Publisher implementieren.

Die folgende Tabelle stellt die Flow Typen den Akka-Streams Typen gegenüber:

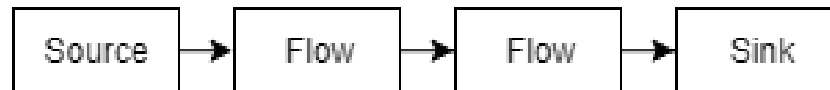
Tabelle 5.10: Flow & Akka-Streams

Flow	Akka-Streams
Publisher	Source
Subscriber	Sink
Processor	Flow
Subscription	Materializer

¹⁷vgl. Lightbend, Inc., Relationship with Reactive Streams, 2018 [Lig18c]

Ein Typischer Akka Stream sieht wie folgt aus:

Abbildung 5.4: Akka-Stream Beispiel



Eine oder mehrere Sources definieren die Datenquelle und können Nachrichten aus primitiven Datentypen, Collections und Arrays erzeugen. Am Ende der Verarbeitungskette befinden sich eine oder mehrere Sinks, welche die ankommenden Daten verarbeiten. Zwischen Source und Sink können Flows platziert werden, die weitere Verarbeitungsschritte umsetzen. Der Materializer alloziert die notwendigen Ressourcen der Prozessschritte und führt diese aus. Diese Art der Verarbeitung ermöglicht eine Entkoppelung der Verarbeitungsschritte und damit eine Wiederverwendung.¹⁸

In Akka-Streams werden Verarbeitungsketten von Source, Flows und Sinks als Graphen bezeichnet. Ein Graph beschreibt die Topologie der Verarbeitungsschritte. Der Stream aus [Abbildung 5.4](#) ist ein trivialer Graph. Komplexe Graphen können auch aus mehreren Eingabe- und Ausgabekanälen bestehen.

Lizenz und Kosten

Akka-Stream ist Open Source und besitzt die Apache-Lizenz 2.0 und ist daher für den kostenfreien, kommerziellen Gebrauch zulässig.

Da eine kommerzielle Nutzung erlaubt ist und keinerlei Kosten entstehen erhält Akka-Stream 10 Punkte.

Punkte	10
--------	----

Dokumentation

Akka bietet eine übersichtliche [Referenzdokumentation](#)¹⁹ für alle Module an. Hierbei können Code-Beispiele wahlweise in Scala oder Java betrachtet werden. Module, die Teil von Akka sind, entsprechen immer der Repository Version. Die aktuelle Akka Version ist 2.5.19 (Stand 10.12.2018). Die [Akka-Stream Referenz](#)²⁰ bietet eine gegliederte Übersicht über alle Teilaspekte der Streams an wie z. B. grundlegende Datentypen wie Source und Sink, IO Verarbeitung, Graphen u. v. m. .

¹⁸vgl. Lightbend Inc., Core Concepts [[Lig18b](#)]

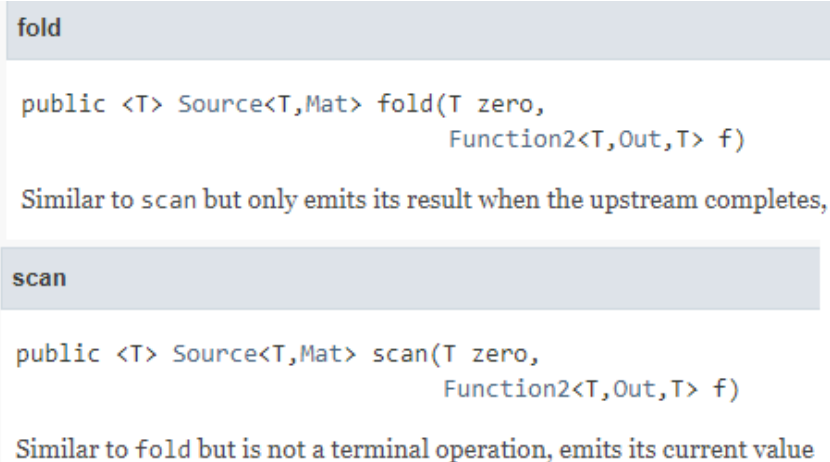
¹⁹Lightbend, Docs [[Lig18a](#)]

²⁰Lightbend, Referenz [[Lig18d](#)]

Die **Javadocs**²¹ sind auf dem aktuellen Stand des Releases. Ein Problem der Dokumentation ist, dass sie nicht nur die Streams enthält, sondern alle restlichen Akka Module wie z. B. Actor, Cluster und Persistence. Das hat zur Folge, dass die Doku sehr umfangreich und unübersichtlich ist. Die Beschreibung ist detailliert und teilweise mit Code-Beispielen versehen.

Insgesamt sind die Javadocs gut und hilfreich, allerdings werden einige Funktionen wie z. B. fold und scan rein textuell erklärt und weder mit Code-Beispielen noch mit Diagrammen vereinfacht dargestellt. Hierfür gibt es 2 Punkte Abzug. Die folgende Abbildung zeigt ein Beispiel der schlecht dokumentierten Funktion scan und fold:

Abbildung 5.5: Ausschnitt aus der Dokumentation der Funktionen scan und fold



The image shows a screenshot of the Akka Javadoc for the `fold` and `scan` functions. The `fold` section includes the signature `public <T> Source<T,Mat> fold(T zero, Function2<T,Out,T> f)` and the description "Similar to scan but only emits its result when the upstream completes,". The `scan` section includes the signature `public <T> Source<T,Mat> scan(T zero, Function2<T,Out,T> f)` and the description "Similar to fold but is not a terminal operation, emits its current value".

```
fold

public <T> Source<T,Mat> fold(T zero,
                             Function2<T,Out,T> f)

Similar to scan but only emits its result when the upstream completes,

scan

public <T> Source<T,Mat> scan(T zero,
                             Function2<T,Out,T> f)

Similar to fold but is not a terminal operation, emits its current value
```

Die Referenzdokumentation hat relativ schwer nachvollziehbare Codebeispiele, die das Wesentliche kompliziert darstellen. Hierfür gibt es drei Punkte Abzug, da für das Kapitel **Working with Graphs** einiges an Recherche gemacht werden muss, die nicht in den Grundlagen erklärt werden.

Punkte	5
--------	---

Handhabung der Bibliothek

Publisher und Subscriber können bedingt kurz und prägnant ausgedrückt werden. Grundsätzlich unterscheidet sich die Akka-Stream Bibliothek sehr stark von RxJava und Reactor. Das Vokabular rund um Akka-Stream ist deutlich größer als das von RxJava oder

²¹Lightbend, Javadoc [[Ligc](#)]

Reactor. Der Code sieht unübersichtlich aus und enthält viele Typen. Allerdings entfällt die Unübersichtlichkeit und der Boilerplate Code, wenn Scala statt Java verwendet wird. Akka-Stream ist eine Bibliothek, die in erster Linie in Scala entwickelt wurde und die Eigenschaften dieser Sprache optimal nutzt. In Java wirkt der Code deutlich komplexer, was auch die Handhabung beeinträchtigt.

Das folgende Beispiel demonstriert das (asynchrone) Emittieren von HELLO, WORLD und !.

Listing 5.8: Akka Streams Hello World

```
1 ActorSystem actorsystem = ActorSystem.create();
2 Materializer materializer = ActorMaterializer.create(system);
3 Source<String, NotUsed> source = Source
4     .from(Arrays.asList("hello",
5                         "world",
6                         "!"))
7     .map(String::toUpperCase);
8 Sink<String, CompletionStage<Done>> sink = Sink
9     .foreach(System.out::println);
10 CompletionStage<Done> done = source.runWith(sink, materializer);
11 done.thenRun(() -> {
12     System.out.println("done");
13     actorsystem.terminate();});
```

Die Ausgabe ist äquivalent zur Ausgabe in [Listing 5.2](#).

In Akka erfolgt die Kommunikation zwischen asynchronen Endpunkten immer über Nachrichten mehrerer Aktoren. Damit ein Programm die Aktoren nutzen kann, benötigt es eine Laufzeitumgebung für die Aktoren, dem Aktorensystem (Zeile 1). Wie in [Unterabschnitt 5.2.4](#) bereits beschrieben, werden Definition und Ausführung getrennt. In Zeile 2 wird der Materializer aus dem Aktorensystem erzeugt. Der Materializer alloziert die Ressourcen für die Verarbeitungsschritte und führt diese aus. Die Verarbeitungsschritte entsprechen in Listing 5.8 Source (Zeile 3) und Sink (Zeile 8). Über die Methode `runWith` wird die Source mit dem Sink verknüpft und über den Materializer ausgeführt. Da standardmäßig die Streams asynchron ausgeführt werden, muss das Aktorensystem terminiert werden, sobald das Programmende erreicht wurde. Der erste generische Typ der Source gibt den Typen des Ausgabeparameters an. Der zweite gibt Hilfsinformationen an, wie z. B. Netzwerkinformationen Ports, IP-Adressen und ähnliches. In Anwendungen in denen keine Hilfsinformationen genutzt werden, wird der Typ `NotUsed` verwendet.²² Im Sink entspricht der erste Typ dem Eingabeparameter und der zweite dem Eingabeparameter für Hilfsinformationen. Das `CompletionStage` ist ein Typ für asynchrone Verarbeitung aus dem `java.util.concurrent` Package.

²²vgl. Lightbend Inc., Streams Quickstart Guide [[Ligd](#)]

Die Handhabung der Akka-Streams Bibliothek ist unkompliziert. Der Java-Code hat einiges an Boilerplate Code und viele generische Typen, dass zum einen die Lesbarkeit erschwert und zum anderen komplexer wirkt im Vergleich zu anderen Bibliotheken wie RxJava und Reactor. Eine falsche Bedienung ist schwer zu bewerkstelligen. Zwar wird das Aktorensystem und der Materializer benötigt, allerdings sind diese Pflichtbestandteil eines jeden Akka-Streams-Programm. Aufgrund dessen, dass Java die Programmiersprache in der Betrachtung ist, erhält Akka-Streams vier Punkte Abzug in der Handhabung.

Punkte	6
--------	---

Weiterentwicklung

Da es kein gesondertes Repository für die Akka-Stream Bibliothek gibt, werden zur Evaluierung die Repositories Alpakka und Akka betrachtet. Alpakka ist eine ergänzende Bibliothek und stellt Adapter Klassen für verschiedene Technologien bereit um mit Akka-Streams zu arbeiten. Somit lässt sich Alpakka mit den Reactor Addons vergleichen. Es gibt noch ein Contributor Repository für Akka-Streams. Allerdings ist dieser primär als Erweiterung für die Akka-Streams API gedacht und ist nicht Teil der Core Bibliothek.

Die folgende Tabelle zeigt einen Auszug aus den Realeases von Akka und Alpakka.

Tabelle 5.11: Auszug aus den Releases von Akka und Alpakka

Akka		Alpakka	
Version	Datum	Version	Datum
2.5.19	07.12.2018	1.0.0	06.11.2018
2.5.18	07.11.2018	0.2.0	04.07.2018
2.5.17	27.09.2018	0.19.0	09.05.2018
2.0.0	06.03.2012	0.18.0	28.03.2018
1.0.0	15.02.2011	0.17.9	19.2.2018

Bei Akka werden Revisionen monatlich aktualisiert. Hauptversionen werden in unregelmäßigen Zeitabständen veröffentlicht. So liegt zwischen Version 1.0 und 2.0 ca.

ein Jahr, während zwischen 2.0 und 2.5 ca. fünf Jahre liegen. Nebenversionen werden ebenfalls unterschiedlich veröffentlicht.

Im Akka Repository werden mithilfe von Labels die jeweiligen Module gekennzeichnet, um Issues und Commits zu referenzieren. Die Akka-Stream Bibliothek wird unter dem Label `t:stream` weiterentwickelt.

Die Alpakka Bibliothek erhielt alle ein bis zwei Monate ein Update auf die nächste Nebenversion. Version 1.0 ist der aktuelle Release (Stand 10.12.2018).

Es lässt sich sagen, dass Akka-Streams (Akka) und Alpakka weiterentwickelt werden. Die Updatezyklen der Haupt- und Nebenversionen sind relativ ungenau zu bestimmen, jedoch kommen regelmäßige Updates und Bugfixes (siehe Revisionen in [Tabelle 5.11](#)). Somit bekommt die Bibliothek 10 Punkte.

Punkte	10
--------	----

Verbreitung

Tabelle 5.12: GitHub Statistik

	RxJava	Reactor	Akka Stream
Repositories	9005	876	836
Projekt Stars	36772	2761	9331
Ranking	48	3743	955

Tabelle 5.13: Schnappschuss aktiv diskutierter Fragen

Bibliothek	Diskutierte Fragen
RxJava	2628
Reactor	762
Akka Streams	1131

Gemäß der [Tabelle 5.12](#) belegt Akka den 2. Platz. Die Wertung bezieht sich allerdings auf das Akka Repository und nicht auf die Akka-Stream Bibliothek. Das liegt daran, dass die Akka-Stream Bibliothek ein Teil von dem Akka Repository ist.

Auf StackOverflow (siehe [Tabelle 5.13](#)) werden Akka-Streams am zweithäufigsten diskutiert. Aufgrund dessen, dass auf StackOverflow gezielt nach Tags gesucht werden kann, wird hier nach Akka-Stream gesucht.

Zusammengefasst belegt Akka-Stream den zweiten Platz unter den drei evaluierten Bibliotheken. Somit werden der Akka-Stream Bibliothek 4 Punkte abgezogen.

Punkte	6
--------	---

5.3. Bewertung

Die Auswertung erfolgt mittels einer gewichteten Evaluationsmatrix. Die Kriterien und die Gewichtung entsprechen den Vorgaben des Unternehmens doubleSlash.

Tabelle 5.14: Gewichtete Evaluationsmatrix

		RxJava		Project Reactor		Akka-Streams	
Kriterium	Gewichtung	Punkte	gew. Punkte	Punkte	gew. Punkte	Punkte	gew. Punkte
Lizenz in Geschäftsanwendungen	3	10	30	10	30	10	30
Dokumentation	3	7	21	10	30	5	15
Handhabung der Bibliothek	2	6	12	8	16	6	12
Weiterentwicklung	1	10	10	10	10	10	10
Verbreitung	1	10	10	3	3	6	6
Ergebnis			83		89		73

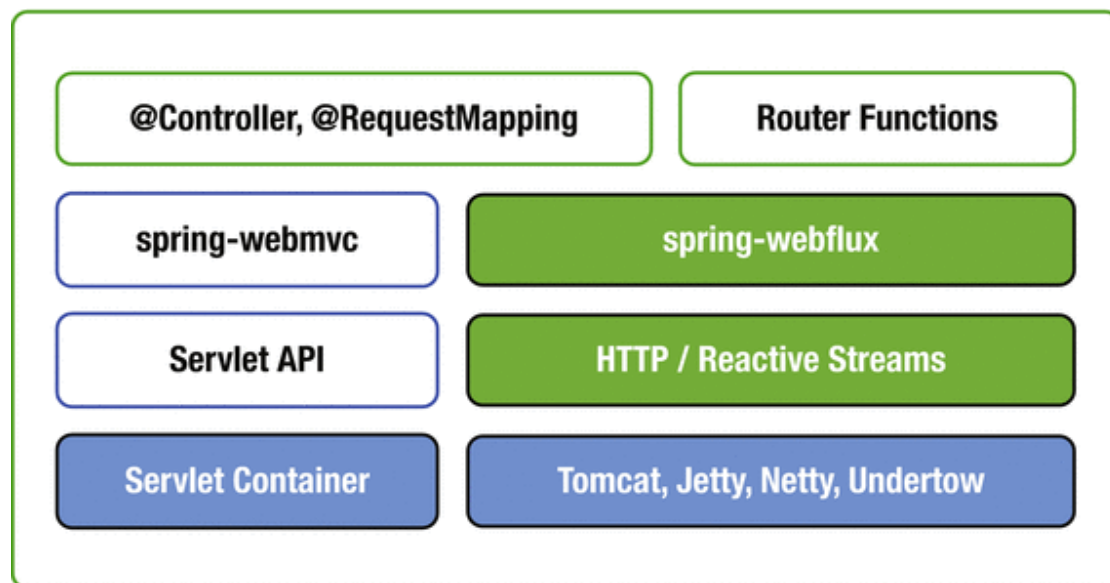
Die Evaluationsmatrix zeigt, dass Reactor knapp vor RxJava und weit vor Akka-Streams liegt. Das liegt daran, dass Reactor eine bessere Dokumentation und eine bessere Handhabung aufweist. Für die Konzeption der prototypischen Anwendung wird daher die Reactor Bibliothek verwendet.

5.4. Frameworks

5.4.1. Spring Webflux

Mit Version 5 des Spring Frameworks, wurde das WebFlux Modul eingeführt, dass eine nicht-blockierende, reaktive Alternative zum klassischen Spring-MVC Modul ist. Kernbestandteil von Webflux ist die Reactor Bibliothek, welche eine Implementierung für Reactive Streams ist. Die folgende Abbildung zeigt das Spring 5 Framework mit den neu hinzugekommenen Modulen (grün markiert).²³

Abbildung 5.6: Spring 5 Module



²³

WebFlux läuft grundsätzlich auf asynchronen Laufzeitumgebungen wie Tomcat, Jetty, Undertow und Netty. Es werden nur Servlet Container unterstützt, die mindestens die Servlet-Spezifikation V. 3.1 implementieren.²³

5.4.2. Akka

Akka ist ein Toolkit, daher eine Sammlung von Bibliotheken, mit dem sich Reactive Systeme realisieren lassen. Das heißt, dass Technologien die entsprechend dem Ab-

²³vgl. Reddy, Reactive Web Applications Using Spring WebFlux [Red17]

schnitt 3.2 notwendig sind, bereits im Repertoire von Akka sind. Die folgende Tabelle fasst einige Akka Module zusammen.

Tabelle 5.15: Unvollständige Liste der Akka Module

Akka Modul	Beschreibung
Actors	Implementierung des Aktorenmodells zur nachrichtenbasierten Kommunikation
Streams	Für nicht-blockierende und nebenläufige Verarbeitung von Daten
Cluster	Ermöglich Skalierung und Widerstandsfähigkeit durch Verteilung auf mehreren Knoten
Distributed Data	Ermöglicht einen Lese- und Schreibzugriffe mit geringer Latenzzeit auf Knoten.

²⁴

Auf der Anwendungsebene nutzt Akka die Akka-Streams als Reactive Streams Implementierung. Prominente Unternehmen, die Akka aktiv in ihrem Production Code nutzen sind unter anderem Paypal, Twitter, Samsung, Intel, Verizon und LinkedIn.

5.4.3. Eclipse Vert.x

Vert.X (ausgesprochen Vertex) ist ein Toolkit für nebenläufige und nicht-blockierende Anwendungen für die JVM. Das Toolkit basiert auf dem Netty Project und bietet eine High-Level API für das zugrunde liegende Netty Framework. Weiterhin ist Vert.X mehrsprachig, also ein Toolkit, dass mehrere Sprachen unterstützt, die auf der JVM laufen. Somit gibt es Support für Java, Scala, Kotlin, Groovy, Javascript, Ruby und Ceylon. ²⁵

²⁴vgl. Akka Docs, [Lig18a]

²⁵vgl. Ponge, Segismont & Viet et al. , Seite 2-6, [PSa18]

Konzeptuell hat Vert.X zwei Kernaspekte. Zum einen das Verticle und zum anderen den Eventbus. Ein Verticle ist eine Verarbeitungseinheit. Es beinhaltet eine Eventloop, in der Events entgegengenommen werden. Per Default werden zwei Verticle pro CPU Kern zur Verfügung gestellt. Der Eventbus ist für die asynchrone Kommunikation zwischen den Verticles zuständig. Zwischen Verticles wird über Nachrichten kommuniziert. Ähnlich zu Akka kann mit dem Vert.X Toolkit reaktive Systeme gebaut werden. ²⁵

Vert.X bietet zwei Arten von Reactive Streams an. Zum einen die Reactive Streams Integration für die Read/Write Streams, wobei es sich es um eine Überbrückung von Read/Write Stream zu Publisher/Subscriber handelt. ²⁶ Zum Anderen die RxJava- Streams, allerdings wurde die Bibliothek entsprechend dem Toolkit angepasst. Das heißt es wurden einige Funktionen entfernt, die nicht mit den Modulen aus Vert.X arbeiten können. Umgekehrt wurde auch die Vert.X API an RxJava angepasst, der sogenannten Rxified API.

Bekannte Unternehmen wie Bosch, VmWare, Hulu, Zalando, Deutsche Börse Group, Red Hat und viele andere nutzen Vert.X in ihren Projekten. ²⁷

²⁶vgl. tsegismont, StackOverflow Kommentar [[tse](#)]

²⁷vgl. Vert.X Homepage, [[Ver](#)]

6

Einsatzszenarien von reaktiver Programmierung

6.1. Netflix

Netflix ist maßgeblich bei der Entwicklung von RxJava beteiligt. Die Gründe für die Entwicklung von RxJava sind im Entwicklerblog ¹ des Unternehmens zu finden. Der wesentliche Grund für die Entwicklung war, dass Netflix die eigene API für nebenläufige Verarbeitung optimieren wollte. Das Problem von klassischen Lösungen wie z. B. Futures und Callbacks ist, dass diese Techniken nur für einen bestimmten Grad an Nebenläufigkeit sinnvoll ist. Futures können zwar verknüpft werden und über Callbacks ein blockierendes Warten vermeiden, allerdings nimmt die Lesbarkeit mit zunehmender Komplexität stark ab. Dies bezeichnet man als Callback Hell. Da die Programmierung mit Futures und Callbacks keine langfristige Lösung für hochgradig antwortbereite Systeme ist, implementierte das Entwicklerteam von Netflix die ReactiveX API für Java.

6.2. PayPal

Die Firma Lightbend (Akka) fasst in ihrer Fallstudie² den Technologiewechsel des Unternehmens PayPal zusammen. PayPal nutzt für die Verarbeitung der Transaktionen die

¹vgl. Christensen et al., Reactive Programming in the Netflix API with RxJava [CH13]

²vgl. lightbend case studies paypal [Ligb]

eigens entwickelte Plattform Squbs. Die Plattform ist in Scala geschrieben und nutzt das Akka Toolkit. Die Vorteile, die Paypal durch die neue Plattform hat, sind Geschwindigkeit und eine effizientere Auslastung der Hardware. Durch die hohe Geschwindigkeit können pro Tag bis zu einer Milliarde Transaktionen verarbeitet werden. Dies erreichen sie durch acht virtuelle Maschinen mit je zwei CPU-Kernen.

Ein weiterer Vorteil ist, dass der Code im Vergleich zur Java Implementierung um ca. 80% reduziert wurde da Scala durch seine ausdrucksstarke Syntax den Code kürzt.

Für PayPal sind unter anderem das Aktoerenmodell und die Streams Bibliothek die Schlüsseltechnologien um die hohen Lasten zu verarbeiten.

6.3. Verizon

Lightbend hat in einer Fallstudie³ den Technologiewechsel des Telekommunikationsunternehmens Verizon zusammengefasst. Verizon nutzt die reaktive Plattform von Lightbend (Akka u. v. m.) für ihr Onlineangebot und hat laut Angaben die Verkäufe verdoppelt. Die Performance der Software wurde bei halber Hardwareauslastung verdoppelt. Mit 2,5 Milliarden Transaktionen pro Jahr ist Verizon die am sechst häufigsten besuchte Website in den USA.

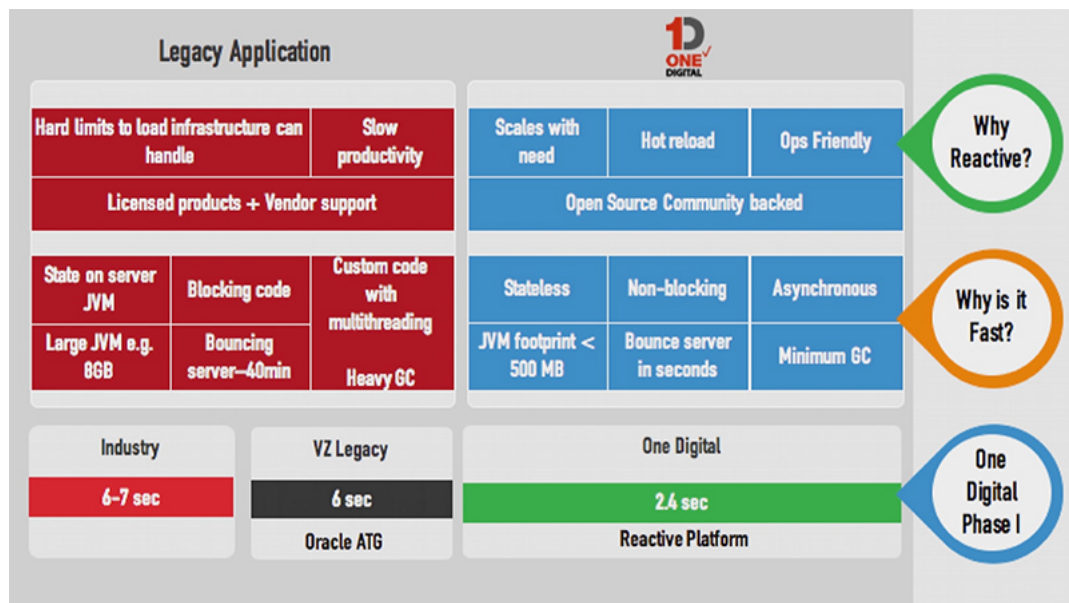
Das alte System basierte ursprünglich auf der Oracle Commerce Platform. Ein Problem war, dass das System eine monolithische Architektur hatte. Das heißt, dass die Codebase sehr komplex und groß war. Builds und Updates mussten über Nacht kompiliert werden. Die Testumgebungen für neue Features beanspruchten zwischen fünf und zehn Tagen an Vorbereitung. In Notfällen betrug die Zeit zum Einspielen der Fixes 24 Stunden. Trotz der Probleme nahm die Anzahl an Transaktionen und Neukunden zu. Das Entwicklerteam benötigte im Schnitt zwei Monate um neue Features zu veröffentlichen.

Verizon engagierte ein Team aus Softwarearchitekten zur Erkundung von neuen Technologien als Alternative zur Oracle Commerce Platform. Zur Auswahl standen Spring Boot und Lightbend's reaktive Plattform (Akka, Play u. v. m.). Das Team entschied sich für Lightbend's reaktive Plattform da sie technische Vorteile in asynchroner Verarbeitung und I/O bieten.

Die folgende Abbildung zeigt die Gegenüberstellung des alten und neuen Systems:

³vgl. lightbend case studies verizon [[Liga](#)]

Abbildung 6.1: Gegenüberstellung des alten und neuen Systems



Die starre Infrastruktur wurde durch eine skalierbare Infrastruktur ersetzt, die nach Bedarf Ressourcen bereitstellt. Das Entwicklungstempo wurde durch Hot Reload erhöht und ermöglicht eine insgesamt höhere Produktivität. Auf Lizenzen und Support wurde verzichtet und durch ein quelloffenes Backend ersetzt.

Die erhöhte Verarbeitungsgeschwindigkeit der Server wurde durch eine zustandlose, nicht blockierende und asynchrone Kommunikation erreicht. Der Speicherverbrauch wurde dadurch von 8 GB auf unter 500 MB gesenkt. Die Startzeit des Systems wurde von 40 Minuten auf wenige Sekunden gesenkt. Durch die Ersetzung der eigenen Multithreading Implementierung durch eine asynchrone Abstraktion wurde die Garbagecollection reduziert.

Im direkten Performancevergleich sank die durchschnittliche Antwortzeit des neuen Systems von 6 Sekunden auf 2,4 Sekunden.

Die Verkäufe nahmen um 235% zu und die **Konversionsrate** stieg um 197%. Aus technischer Sicht benötigt das neue System 40% weniger Bauzeit und hat insgesamt 50% weniger Kosten als das Vorgängersystem.

7

Konzeption einer reaktiven Anwendung

Die Anwendung basiert auf der Idee der Kommunikation zwischen autonomen Fahrzeugen und Servern. Ein Fahrzeug misst die Signalstärke eines Signals zum nächsten Mobilfunkmast und sendet einem Server die Signalstärke sowie die aktuelle Position in Form von Koordinaten. Der Server verarbeitet die Daten und sendet dem Fahrzeug eine Antwort. Mit den gewonnenen Daten (Signalstärke und Koordinate) lassen sich nun Heatmaps erzeugen, also bspw. eine Stadtkarte die farblich markierte Bereiche anzeigt, in denen autonome Fahrzeuge, ohne Verbindungsabbruch, fahren können.

Die Anwendung soll die Kommunikation zwischen Server und Fahrzeug reaktiv umsetzen. Anschließend sollen ein Vergleich zu einer blockierenden Variante gezogen werden.

7.1. Anforderungen

Die prototypische Anwendung muss folgende Anforderungen erfüllen:

Client: Ein Auto fährt eine vordefinierte Route ab. Währenddessen misst es die Signalstärke zum nächsten Mast und sendet diese zusammen mit der Position an den Server. Zwischen jeder versendeten Messung wartet das Auto 200 ms.

Server: Empfängt eine Positionskoordinate und die zugehörige Signalstärke, verarbeitet diese und sendet eine Antwort an das Auto zurück

7.2. Die Wahl der Technologien

Die Applikation wird in zwei Varianten implementiert. Zum einen eine reaktive und zum anderen eine synchrone, blockierende Variante.

Im Rahmen dieser Thesis wird TCP anstelle von HTTP als Übertragungsprotokoll verwendet. Der Grund dafür ist, dass eine triviale Datenstruktur (Positionskoordinate und Signalstärke) verwendet wird. Der Mehraufwand, der durch die Nutzung eines Http-Frameworks entsteht ist in diesem Fall nicht gerechtfertigt.

Für die reaktive Anwendung wird die Reactor Bibliothek verwendet, da sie unter den bekanntesten Bibliotheken am besten abschneidet (siehe Kapitel 5.3). Als Kommunikationstechnologie wird das RSocket¹ Protokoll verwendet. Das RSocket Protokoll nutzt die Reactive Streams Semantik um Daten über verschiedenste Protokolle (TCP, UDP uvm.) zu übertragen. Die Schnittstellen sind mit denen von Reactor kompatibel.

Für die blockierende Variante wird die Socket API aus dem java.net Package verwendet, da die reaktive Version ebenfalls eine Socket Implementierung ist.

¹www.rsocket.io

8

Umsetzung

8.1. Die Umsetzung in RSocket und Reactor

Der gesamte Quellcode zur Anwendung ist auf einem Github Repository unter <https://github.com/Menkir/reactive-streams> zu finden.

Die folgende Tabelle zeigt die wichtigsten Klassen:

Tabelle 8.1: Klassen aus dem eigenen Prototype-Package

Klasse	Beschreibung
CarServer	Server der Car-Verbindungen akzeptiert und die Messungen als Measurement Objekt verarbeitet.
Car	Client der eine Verbindung zum Server herstellt und Measurements sendet und die Antworten vom Server empfängt.
CarConfiguration	Dient zu Konfiguration von der Zeitverzögerung zwischen zwei Measurements und der Route
Measurement	Kapselt eine Koordinate als Tupel und eine Signalstärke als Integer Zahl. Die Signalstärke wird zwischen 0 (kein Signal) und 10 (volles Signal) definiert.
IRoute	Modelliert eine Route (Abfolge von Koordinaten), die ein Auto (Car) abfährt.
Serializer	Transformiert Measurement in einen Typ für den Transport zwischen Car und Server und umgekehrt.

Anmerkung zur Car Klasse: Die reaktive als auch synchrone Variante enthalten einen internen Zähler: `flowrate`. Die `Flowrate`-Variable dient dazu, zu ermitteln wie viele erfolgreiche Messungen gesendet wurden. Eine Messung ist genau dann erfolgreich, wenn eine `Car` Instanz eine Messung zum Server sendet und eine Antwort erhält.

Die folgende Auflistung zeigt die wichtigsten Klassen der `RSocket` Bibliothek:

- `RSocketFactory`: Instantiiert Server oder Client
- `ServerRSocketFactory`: Instantiiert einen Server `RSocket`
- `SocketAcceptor`: Handler für ankommende Verbindungen
- `AbstractSocket`: Definiert wie Daten übertragen werden: `Request/Response`, `Request/Stream`, `Channel` oder `Fire-and-Forget`.
- `ClientRSocketFactory`: Instantiiert Client `RSocket`
- `RSocket`: Der Socket der eine Verbindung zwischen Server und Client hält.
- `Payload`: Ist ein Typ der beim Transport zwischen Client und Server verwendet wird.

8.1.1. Der RSocket Server

Die `Server` Klasse implementiert zwei Methoden: Zum einen die `Receive`-Methode und zum anderen die `Close`-Methode. Die folgende zwei Listings zeigen zum einen die relevanten importierten Klassen, sowie die Implementierung des `RSocket`-Servers:

Listing 8.1: Reactive Server: Relevante Imports

```
1 import io.rsocket.AbstractRSocket;
2 import io.rsocket.Payload;
3 import io.rsocket.RSocketFactory;
4 import io.rsocket.transport.netty.server.TcpServerTransport;
5 import org.reactivestreams.Publisher;
6 import reactor.core.Disposable;
7 import reactor.core.publisher.Flux;
8 import reactor.core.publisher.Mono;
```

Listing 8.2:]Reactive Server: Receive-Methode aus [async.server.Server]

```

1      public void receive () {
2          channel = RSocketFactory.receive ()
3              .acceptor ((setupPayload, reactiveSocket) ->
4                  Mono.just (new AbstractRSocket () {
5                      @Override
6                      public Flux<Payload> requestChannel (Publisher<
7                          Payload> payloads) {
8                          return Flux.from (payloads)
9                              .flatMapSequential (
10                                  payload -> Flux.just (payload)
11                                      .delaySequence (Duration.
12                                          ofMillis (10))) );
13                      }
14                  })))
15          .transport (TcpServerTransport.create (socketAddress.
16              getHostName (), socketAddress.getPort ()))
17          .start ()
18          .block ();
19      }

```

Im Konstruktor der Klasse wird Objekt der `InetSocketAddress` Klasse instantiiert. In dem Objekt sind Informationen bezüglich IP-Adresse und Port. Über die `RSocketFactory` Klasse wird der Socket konfiguriert. Die Methode `acceptor` (Zeile 3) enthält die Parameter der `Accept-Methode` aus der `SocketAcceptor` Klasse. Der Rückgabewert des Lambdaausdrucks muss ein `Mono<RSocket>` sein, da es der Rückgabewert der impliziten `Accept-Methode` aus `SocketAcceptor` ist. In Zeile 4 wird ein `Mono` zurückgeliefert. Das `Mono` enthält eine anonyme Klasse von `AbstractSocket`. Die anonyme `AbstractSocket` Klasse überschreibt die `Requestchannel-Methode` (Zeile 6) und gibt einen `Flux` vom Typ `Payload` zurück. Dieser `Flux` wird im Client subscribed.

Über die Methode `transport` (Zeile 12) wird das Protokoll für die Kommunikation konfiguriert. Für die Anwendung wird TCP verwendet. Die Methode `start` startet die Event-Loop von `RSocket`. Da die Event-Loop asynchron gestartet wird, kann Zeit vergehen bis der Server tatsächlich gestartet wurde. Aus diesem Grund wird blockierend über die Methode `block` auf einen erfolgreichen Start gewartet.

Die überschriebene Methode `channelRequest` implementiert das Channel Interaktionsmodellen von RSocket. Insgesamt bietet RSocket vier Modelle an:

- Request und Response
- Request und Stream
- Fire-and-Forget
- Channel

Das Channel Modell wurde gewählt da es einen bidirektionalen Kommunikationskanal zwischen Client und Server bildet. Somit lässt sich auch der Durchsatz für einen Client messen. Mit Modellen wie Request und Response oder Request und Stream wird stets ein Element übertragen und ein oder mehrere Elemente als Antwort gesendet. Das Fire-and-Forget Modell passt ebenfalls nicht in den Anwendungsfall da nur ein Element übertragen wird ohne, dass der Client eine Antwort erhält.

Die Requestchannel-Methode (Zeile 6) wird clientseitig aufgerufen. Aufgrund dessen, dass das Argument der Methode ein Publisher ist, muss er in einen allgemeinen Flux umgewandelt werden um die komfortable Reactor API zu nutzen. Über die Methode `from` (Zeile 7) wird ein Flux mit den Elementen aus dem Publisher erzeugt. Die Funktion `flatMapSequential` bewirkt, dass die Payload in ein `Flux<Payload>` umgewandelt wird. Dadurch ist es möglich dem inneren Flux eine Zeitverzögerung von 10 ms als Bearbeitungszeit zu geben. Die Bearbeitung eines Measurement ist daher nur eine Zeitverzögerung. Im Anschluss wird das Flux schließlich flach, daher von `Flux<Flux<Payload>` zu `Flux<Payload>` und dem Aufrufer (Car), zurückgeliefert.

8.1.2. Der RSocket Client

Die Car Klasse implementiert die Connect-, Send- und Close-Methode. Die Connect-Methode stellt die Verbindung zwischen Car und Server her, wie folgendes Listing zeigt:

Listing 8.3: Reactive Car: Relevante Imports

```
1 import io.rsocket.RSocket;
2 import io.rsocket.RSocketFactory;
3 import io.rsocket.transport.netty.client.TcpClientTransport;
4 import io.rsocket.util.DefaultPayload;
5 import prototype.utility.Serializer;
6 import reactor.core.Disposable;
7 import reactor.core.publisher.Flux;
```

Listing 8.4:]Reactive Car: Connect-Methode aus [async.client.Car]

```

1      public void connect() {
2          this.client = RSocketFactory
3              .connect()
4              .keepAliveAckTimeout(Duration.ofMinutes(30))
5              .transport(TcpClientTransport.create(socketAddress.
6                  getHostName(), socketAddress.getPort()))
7              .start()
8              .block();
9      }

```

Über die Factory-Methode wird der Socket konfiguriert. Die KeepAliveAckTimeout-Methode setzt den Timeout für eine offene Verbindung auf 30 Minuten um sicher zu gehen, dass die späteren Benchmarks ohne Verbindungsunterbrechung laufen. Als Transportprotokoll wird TCP verwendet und über ein Objekt der Klasse TcpClientTransport konfiguriert. Die Methode start und block stellen eine Verbindung her und warten blockierend (da nur eine Verbindung) auf eine erfolgreiche Verbindungsherstellung.

Die Send-Methode der reaktiven Klasse Car emittiert die Messungen von einem Car, wie folgendes Listing zeigt:

Listing 8.5:]Send Methode aus [async.client.Car]

```

1      public void send() {
2          serverEndpoint = client.requestChannel(
3              Flux.fromIterable(routingFactory.getRoutingType(
4                  carConfiguration.ROUTETYPE).getRouteAsList())
5                  .repeat(100_000)
6                  .delayElements(carConfiguration.DELAY)
7                  .doOnNext(measurements -> measurements.
8                      setSignalStrength(((int) (Math.random() * 10))))
9                  .map(measurements -> DefaultPayload.create(Serializer
10                      .serialize(measurements)))
11                  .share()
12          ).subscribe(payload -> ++flowrate);
13      }

```

Der Client RSocket fragt über die Methode channelRequest einen bidirektionalen Kanal an. Diese Methode muss serverseitig implementiert sein (Siehe Kapitel 8.1.1). Als Argument wird ein Publisher übergeben, der Elemente vom Typ Payload emittiert. Da Flux eine Implementierung des Publishers ist, wird eine Flux Instanz übergeben. Das Flux erzeugt aus einem Iterable ein Flux<Coordinate>. Das Iterable ist eine Liste von statisch definierten Koordinaten (List<Coordinate>). Über die Methode repeat (Zeile 4) wird eine Runde der Route 100.000 Mal wiederholt. Somit wird ein lang andauernder Flux simuliert, jedoch nicht unendlich. Im Rahmen dieser Anwendung wird eine festgelegte Größe emittiert da die äquivalente Socket Implementierung ebenfalls

eine endliche Anzahl an Messungen versendet, denn endlose Emissionen über Sockets sind nicht möglich. Sockets benötigen stets ein Terminal, das signalisiert, dass keine Datenelemente mehr folgen.

Die Methode `delayElements` (Zeile 5) verzögert die Emission der Messungen um eine definierte Zeit. Das heißt, dass zwischen zwei Messungen immer eine gewisse Zeit gewartet wird. Im Kapitel 8.3 wird ein Vergleich zwischen verzögerten und nicht verzögerten Varianten gezogen.

Die `doOnNext` Methode (Zeile 6) setzt für jede Messung eine Signalstärke. Die Signalstärke wird durch eine Zahl zwischen 0 und 10 simuliert. In Zeile 7 wird jede Koordinate in eine Payload umgewandelt. Im Anschluss wird die `share` Methode aufgerufen, die das Flux von einem Cold-Stream in einen Hot-Stream umwandelt und somit $n \geq 1$ Subscriber akzeptiert und die Payload im Multicast emittiert. Diese Eigenschaft ist genau dann notwendig, wenn andere Teile der Software die Daten empfangen möchten wie z. B. eine grafische Benutzeroberfläche.

Die `Subscribe`-Methode in Zeile 9 erzeugt eine Subscription auf den Rückgabewert (Flux<Measurement>). Der Flux ist der Rückgabewert der auf dem Server definierten `RequestChannel`-Methode (siehe Listing 8.2 Zeile 6).

Die `Close` Methode der reaktiven Klasse `Client` beendet die Subscription zwischen Client und Server:

Listing 8.6: Close Methode aus `async.client.Car`

```
1      public void close () {  
2          serverEndpoint.dispose ();  
3      }
```

8.2. Die Umsetzung in Socket

Die folgende Tabelle zeigt die wichtigsten Klassen aus dem `java.net` Package:

Tabelle 8.2: Klassen aus dem java.net Package

Klasse	Beschreibung
ServerSocket	Socket für den Host
Socket	Socket für den Client
ObjectInputStream	Eingangskanal für Objekte
ObjectOutputStream	Ausgangskanal für Objekte

Die Klassen des eignen prototype Package sind in Kapitel 8.1 zu finden.

Der blockierende Server implementiert die receive und close Methode. Das folgende Listing zeigt die receive Methode:

Listing 8.7: Receive Methode aus sync.server.Server

```

1      public void receive() throws IOException {
2          this.serverSocket = new ServerSocket();
3          serverSocket.bind(socketAddress);
4          CompletableFuture.runAsync(() -> {
5              while(!serverSocket.isClosed()){
6                  Socket clientSocket;
7                  try {
8                      clientSocket = serverSocket.accept();
9                  } catch (IOException e) {
10                     System.err.println("[CarServer] " + e.getMessage());
11                     return;
12                 }
13
14                 Socket finalClientSocket = clientSocket;
15                 CompletableFuture.runAsync(() -> {
16                     ObjectInputStream ois;
17                     ObjectOutputStream oos;
18                     while(true) {
19                         try {
20                             assert finalClientSocket != null;
21                             ois = new ObjectInputStream(new
22                                 BufferedInputStream(finalClientSocket.
23                                     getInputStream()));
24                             Measurement measurement = (Measurement) ois.
25                                 readObject();
26                             // do expensive work
27                             Thread.sleep(10);
28                             oos = new ObjectOutputStream(new
29                                 BufferedOutputStream(finalClientSocket.
30                                     getOutputStream()));
31                             oos.writeObject(measurement);
32                             oos.flush();
33                         } catch (IOException | ClassNotFoundException |
34                             InterruptedException e) {
35                             System.err.println("[CarServer] Connection to
36                                 Client was dropped " + e.getMessage());
37                             break;
38                         }
39                     }
40                 }, executorService);
41             }
42         }, executorService);
43     }

```

Der Server lagert jeden Request in einem Future aus. Das heißt, dass eingehende Socket-Verbindungen über Threads asynchron verarbeitet werden. Innerhalb des Future werden über den InputStream vom Server die Messungen blockierend und synchron empfangen. Anschließend werden die Messungen unverändert an den Client über den OutputStream des Server zurückgesendet.

Die Car Klasse implementiert die connect, send und close Methode. Das folgende Listing zeigt die connect Methode:

Listing 8.8: Connect Methode aus sync.client.Car

```
1      public void connect() {
2          clientSocket = new Socket();
3          try {
4              clientSocket.connect(socketAddress);
5          } catch (IOException e) {
6              System.err.println("[Car] cannot connect Host " + e.getMessage());
7              ;
8          }
9      }
```

Zuerst wird eine Socket Instanz erzeugt und anschließend über die Methode connect mit dem Server verbunden. Der Paramter in der connect Methode enthält die Informationen zu IP und Port.

Das folgende Listing zeigt die öffentliche send und die private sendData Methode:

Listing 8.9: Send Methode aus sync.client.Car

```
1      public void send() {
2          int deliveredElements = 0;
3          int MAXELEMENTS = 100_000_000;
4          do{
5              for (Measurement measurement : route) {
6                  measurement.setSignalStrength((int)(Math.random()*10));
7                  sendData(measurement);
8                  delay();
9                  deliveredElements++;
10             }
11         } while(deliveredElements < MAXELEMENTS && !done);
12     }
```

Listing 8.10: SendData Methode aus sync.client.Car

```

1      private void sendData(Measurement measurement){
2          try {
3              // send
4              ObjectOutputStream oos = new ObjectOutputStream(new
5                  BufferedOutputStream(clientSocket.getOutputStream()));
6              oos.writeObject(measurement);
7              oos.flush();
8
9              // receive
10             ObjectInputStream ois = new ObjectInputStream(new
11                 BufferedInputStream(clientSocket.getInputStream()));
12             Measurement c = (Measurement) ois.readObject();
13
14             // increment flowrate
15             ++flowrate;
16         } catch (IOException | ClassNotFoundException e) {
17             e.printStackTrace();
18         }
19     }

```

Die send Methode emittiert hundert Millionen Koordinaten. In jedem Schleifendurchlauf wird eine Messung versendet, aktiv gewartet und anschließend die Prüfvariable inkrementiert. Innerhalb der sendData Methode wird ein OutputStream geöffnet und dem Server die Messung gesendet, anschließend wird über ein InputStream auf die Antwort des Servers blockierend gewartet und das Ergebnis gelesen. Zum Schluss wird die flowrate inkrementiert, die für den Benchmark benötigt wird.

Die close Methode schließt den Socket, der mit dem Server verbunden ist, wie folgendes Listing zeigt:

Listing 8.11: Close Methode aus sync.client.Car

```

1      public void close() {
2          if(clientSocket != null){
3              done = true;
4              try {
5                  clientSocket.close();
6              } catch (IOException e) {
7                  e.printStackTrace();
8              }
9          }
10     }

```

Wenn die Variable clientSocket eine gültige Instanz der Klasse Socket besitzt, dann darf der Socket auch über die Methode close geschlossen werden.

8.3. Performance Benchmark

Um einen qualitativen Vergleich zwischen den Varianten zu bewerkstelligen, wird der durchschnittliche Durchsatz pro Sekunde ermittelt. Hierbei entspricht der Durchsatz der Anzahl an erfolgreich übertragenen Requests. Ein Request ist genau dann erfolgreich, wenn auf ein Request eine Response empfangen wurde.

8.3.1. Die Testumgebung

Die Clients/ Fahrzeuge wurden auf einem Windows 10 Enterprise x64 Computer mit einem Intel I7 3,6 GHz Prozessor (acht CPU-Kerne) und 16 GB RAM simuliert. Der Server lief entfernt auf einem Macbook AIR6,2 (anfang 2014) mit Intel I5 1,4GHz Prozessor (vier CPU-Kernen) und 4 GB RAM. Das Testprogramm als auch der Server liefen unter der Oracle JDK Version 1.8_162.

8.3.2. Der Benchmark

Es werden insgesamt zwei Benchmarks je Variante (synchron/ reaktiv) ausgeführt. Zum einen ein Benchmark mit einprogrammierter Zeitverzögerung und einmal ohne Zeitverzögerung. Der Grund weshalb Zeitverzögerungen verwendet werden ist, dass ein realistischer Test gemacht werden kann. Ein Fahrzeug entspricht einem Client. Wenn nun ein Client die maximal mögliche Anzahl an Koordinaten sendet, hat dies keine praktische Relevanz für eine Auswertung da sonst zu viele identische Messungen gesendet werden.

Der Benchmark ist in zwei Phasen unterteilt. Zum einen in eine Warmlauf-Phase und zum anderen in eine Test-Phase. In der Warmlauf-Phase werden Optimierungen vom Hotspot JIT-Compiler vorgenommen um den Code zu beschleunigen. Würde man in dieser Phase messen, zuerst anhand von nicht-optimierten und später mittels optimierten Methoden gemessen, was dazu führen würde, dass die Messungen verfälscht werden.

Um zu gewährleisten, dass nur optimierte Methoden getestet werden, wird die JVM mit zwei Argumenten gestartet. Zum einen `-client` und zum anderen `-XX:+PrintCompilation`. Das erste Argument bewirkt, dass der Threshold¹, für die Methoden auf 1500 gesetzt wird. Das heißt, dass eine Methode 1500 mal aufgerufen werden muss um optimiert zu werden. Das zweite Argument gibt in Echtzeit die optimierten Methoden aus dem JIT-

¹Bezeichnet die Anzahl an Aufrufen, die eine Methode braucht um vom JIT-Compiler optimiert zu werden.

Compiler aus.² Die Ausgabe des JIT-Compilers dient zur Verifizierung ob die Methode tatsächlich optimiert wurde.

Konkret müssen genau die Methoden optimiert werden, die für die Emission der Messungen zuständig sind da diese den Durchsatz generieren. Wenn bspw. 1000 Messungen emittiert werden, wird keine Optimierung vom JIT-Compiler vorgenommen. Der entstandene Durchsatz ist somit verfälscht.

Da das Flux die Payload über die Reactive Streams Semantik emittiert, muss die `onNext` Methode optimiert werden. Jedoch wird die `onNext` Methode nicht explizit in der `Car` Klasse aufgerufen sondern implizit vom Publisher (Flux). Die folgende Abbildung zeigt die Print Ausgabe des JIT-Compilers innerhalb der Warmup Methode:

Abbildung 8.1: Ausschnitt aus der JIT Ausgabe

```
$ grep warmup.txt -e "onNext "
2850 1093 ! 3 reactor.core.publisher.FluxMap$MapSubscriber::onNext (97 bytes)
2873 1182 ! 3 reactor.core.publisher.LambdaSubscriber::onNext (40 bytes)
2878 1172 ! 3 reactor.core.publisher.StrictSubscriber::onNext (77 bytes)
2904 1269 ! 3 reactor.core.publisher.FluxPeekFuseable$PeekFuseableSubscriber::onNext (116 bytes)
2908 1270 ! 3 io.rsocket.transport.netty.SendPublisher$InnerSubscriber::onNext (9 bytes)
2908 1271 ! 3 io.rsocket.transport.netty.SendPublisher$InnerSubscriber::onNext (51 bytes)
2967 1430 ! 3 reactor.core.publisher.FluxPeek$PeekSubscriber::onNext (95 bytes)
2969 1421 ! 3 reactor.core.publisher.FluxGroupBy$GroupByMain::onNext (193 bytes)
2979 1457 ! 3 reactor.core.publisher.FluxGroupBy$UnicastGroupedFlux::onNext (68 bytes)
2980 1459 ! 3 reactor.core.publisher.MonoFlatMapMany$FlatMapManyInner::onNext (11 bytes)
2998 1501 ! 3 reactor.core.publisher.FluxRepeat$RepeatSubscriber::onNext (21 bytes)
3035 1571 ! 3 reactor.core.publisher.FluxConcatMap$ConcatMapImmediate::onNext (65 bytes)
3047 1627 ! 4 reactor.core.publisher.FluxMap$MapSubscriber::onNext (97 bytes)
3049 1093 ! 3 reactor.core.publisher.FluxMap$MapSubscriber::onNext (97 bytes) made not entrant
3094 1735 ! 3 reactor.core.publisher.UnicastProcessor::onNext (100 bytes)
3095 1736 ! 3 reactor.core.publisher.FluxDoFinally$DoFinallySubscriber::onNext (11 bytes)
3095 1737 ! 3 reactor.core.publisher.FluxConcatArray$ConcatArraySubscriber::onNext (21 bytes)
3109 1777 ! 4 reactor.core.publisher.LambdaSubscriber::onNext (40 bytes)
3110 1182 ! 3 reactor.core.publisher.LambdaSubscriber::onNext (40 bytes) made not entrant
```

Die erste Zeile in Abbildung 8.1 ist das `FluxMap` Objekt, das innerhalb der `map` Methode des Flux in Listing 8.5 Zeile 7 aufgerufen wird. Da der Flux lazy evaluiert (Siehe Kapitel 4.2.5) wird, beginnt der Flux erst zu senden, wenn eine Subscription erstellt wurde. Die zweite Zeile in Abbildung 8.1 ist die `onNext` Methode, die der Publisher beim Subscriber aufruft, wenn ein Element emittiert wird. Da der Subscriber implizit ein Lambda-Ausdruck ist, wird die `onNext` Methode des `LambdaSubscriber` aufgerufen. Hiermit wird sichergestellt, dass die Methode `onNext` optimiert ist. Dieses Verfahren ist analog für die synchrone `Car` Klasse. In der synchronen `Car` Klasse muss die `sendData` Methode geprüft werden da dort die Input- und Outputstreams aufgerufen werden, die die Messungen übertragen. (Siehe Listing 8.10).

²vgl. Oracle, Java HotSpot VM Options [Ora]

8.3.3. Simulation in Scala

Die Simulationsklassen `AsyncSimulation` und `SyncSimulation` implementieren jeweils die Methoden `run`, `warmUp` und `benchmark`.

In der `Run`-Methode wird die `Warmup`-Methode aufgerufen, die Methoden vom JIT-Compiler optimiert. Dann werden Futures gestartet, wobei jedes Future ein Benchmark ausführt. Die Messzeit eines Benchmark steht in der `durationList` Liste. Anschließend wird auf das Ende aller Futures gewartet und das Resultat abgespeichert. Das folgende Listing zeigt die Implementierung der `Run`-Methode aus der `AsyncSimulation` Klasse:

Listing 8.12: Run Methode aus der `AsyncSimulation` Klasse

```

1  def run(config: CarConfiguration = new CarConfiguration()): Unit = {
2      logger.info("START TEST")
3      warmUp()
4      Future.sequence(durationList.map(duration => Future((duration,
5          benchmark(duration, config))(executors)))).onComplete(
6          result => {
7              printResult(result.get)
8              saveResult("ReactiveBenchmarkResult.txt", result.get)
9              logger.info("END TEST")
10         }
11     )
12 }
```

Die `Warmup`-Methode:

Listing 8.13: Warmup Methode aus der `AsyncSimulation` Klasse

```

1  def warmUp(): Unit={
2      logger.info("WARMUP")
3      val iterations = 1500
4      val car = new Car(hostInfo)
5      car.connect()
6      car.send()
7      while(car.getFlowrate < iterations){
8          Thread.sleep(1)
9      }
10     car.close()
11 }
12 }
```

Wie in Kapitel 8.3.2 beschrieben, werden 1500 Koordinaten als Request gesendet und empfangen um die Methoden der Bibliothek zu optimieren.

Die Benchmark-Methode:

Listing 8.14: Benchmark Methode aus der AsyncSimulation Klasse

```
1  def benchmark(measuretime: Int, config: CarConfiguration): Int = {
2      logger.info("START BENCHMARK")
3      val car = new Car(hostInfo)
4      car.connect()
5      car.send()
6      Thread.sleep(measuretime)
7      logger.info("Throughput: {} processed requests", car.getFlowrate)
8      logger.info("END BENCHMARK")
9      car.close()
10     car.getFlowrate
11 }
```

In der Methode wird eine Client Instanz erzeugt und gestartet. In der asynchronen Variante ist die `send()` Methode nicht-blockierend. In Zeile 6 schläft der Thread die notwendige `Measuretime`-Zeit während das Car die Messungen sendet und den Durchsatz Zähler (`flowcounter`) inkrementiert. Anschließend die Instanz mit der Methode `close` beendet und die Variable `flowcounter` zurückgeliefert.

Die Simulation wird aus der folgenden Main Methode gestartet:

Listing 8.15: Main Methode aus der AsyncSimulation Klasse

```
1  def main(args: Array[String]): Unit = {
2      val simulation = new AsyncSimulation()
3      simulation.run()
4  }
5 }
```

Die `SyncSimulation` Klasse unterscheidet sich im wesentlichen nur darin, dass das Senden des Car in einem `Future` ausgelagert wird. Der restliche Code ist analog und im Anhang [A.1.6](#) zu finden.

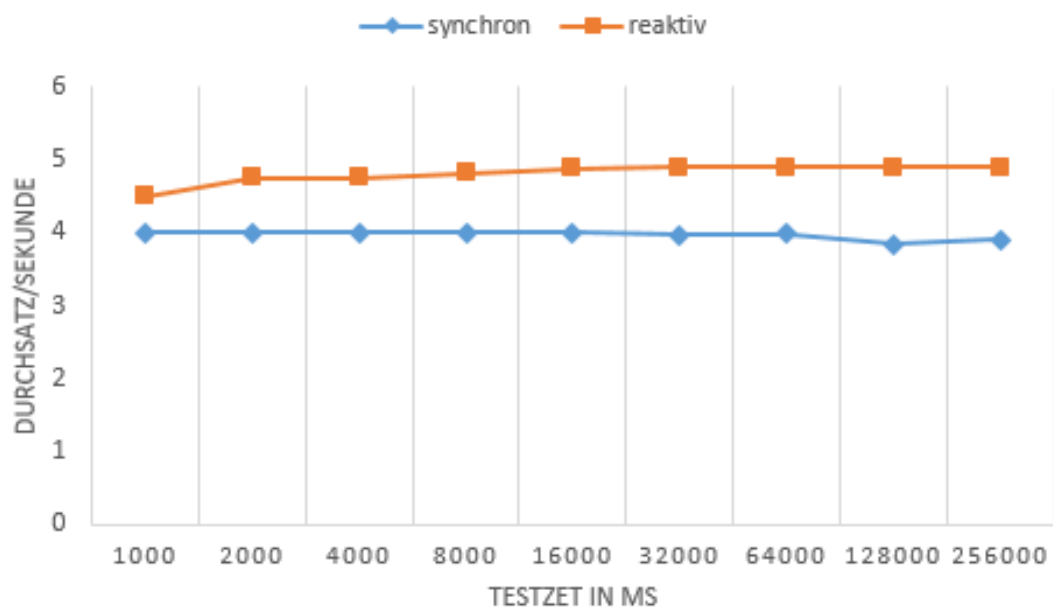
8.3.4. Auswertung der Testergebnisse

Messung mit Verzögerung

Bei der ersten Messung wurde eine Sendeverzögerung im CarConfiguration Objekt von 200 ms gesetzt. Somit folgt, dass ein durchschnittlicher Durchsatz von 5 Koordinaten pro Sekunde als optimal gilt.

Die folgende Abbildung zeigt die Messung der reaktiven und synchronen Anwendung mit Sendeverzögerung.

Abbildung 8.2: Messung mit Zeitverzögerung



Die folgende Tabelle zeigt den Durchsatz beider Varianten:

Tabelle 8.3: Messung mit Zeitverzögerung (tabellarisch)

	Reaktiv	Synchron
Messzeit in ms	Durchsatz / Sekunde	
1000	4	0
2000	4,5	4
4000	4,75	4
8000	4,75	4
16000	4,8125	4
32000	4,875	4
64000	4,890625	3,96875
128000	4,890625	3,976563
256000	4,890625	3,835938
512000	4,894531	3,90625

Die reaktive Variante strebt einen Wert von 5 an und ist somit näher am optimalen Durchsatz als die synchrone Variante. Der Grund weshalb die synchrone Variante einen geringeren Durchsatz hat ist, dass die zusätzlich zu der Verzögerung des Car (200 ms) noch auf die Antwort des Servers warten muss. Da der Server 10 ms braucht um eine Messung zu verarbeiten, muss ein synchrones Car mindestens 210 ms warten. Die folgende Abbildung demonstriert die Kommunikation zwischen Server und Car in der reaktiven und synchronen Variante:

Abbildung 8.3: Reaktive Car und Server Kommunikation mit Sendeverzögerung

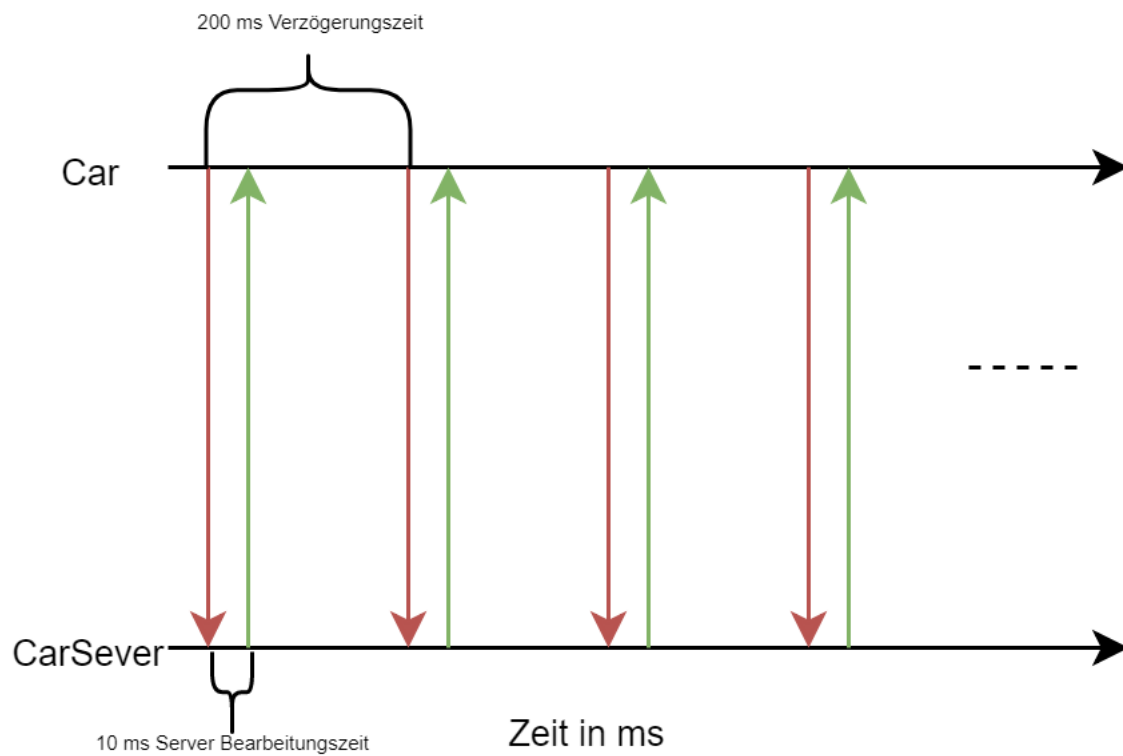
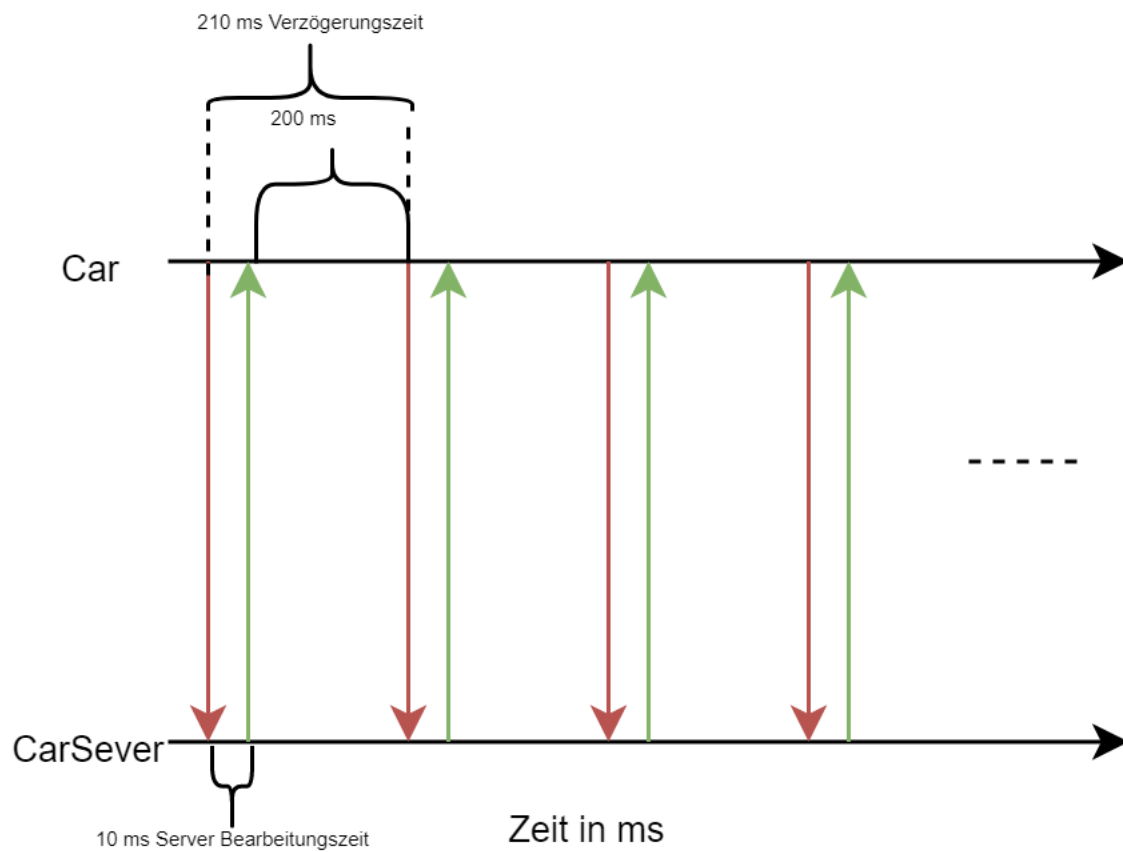


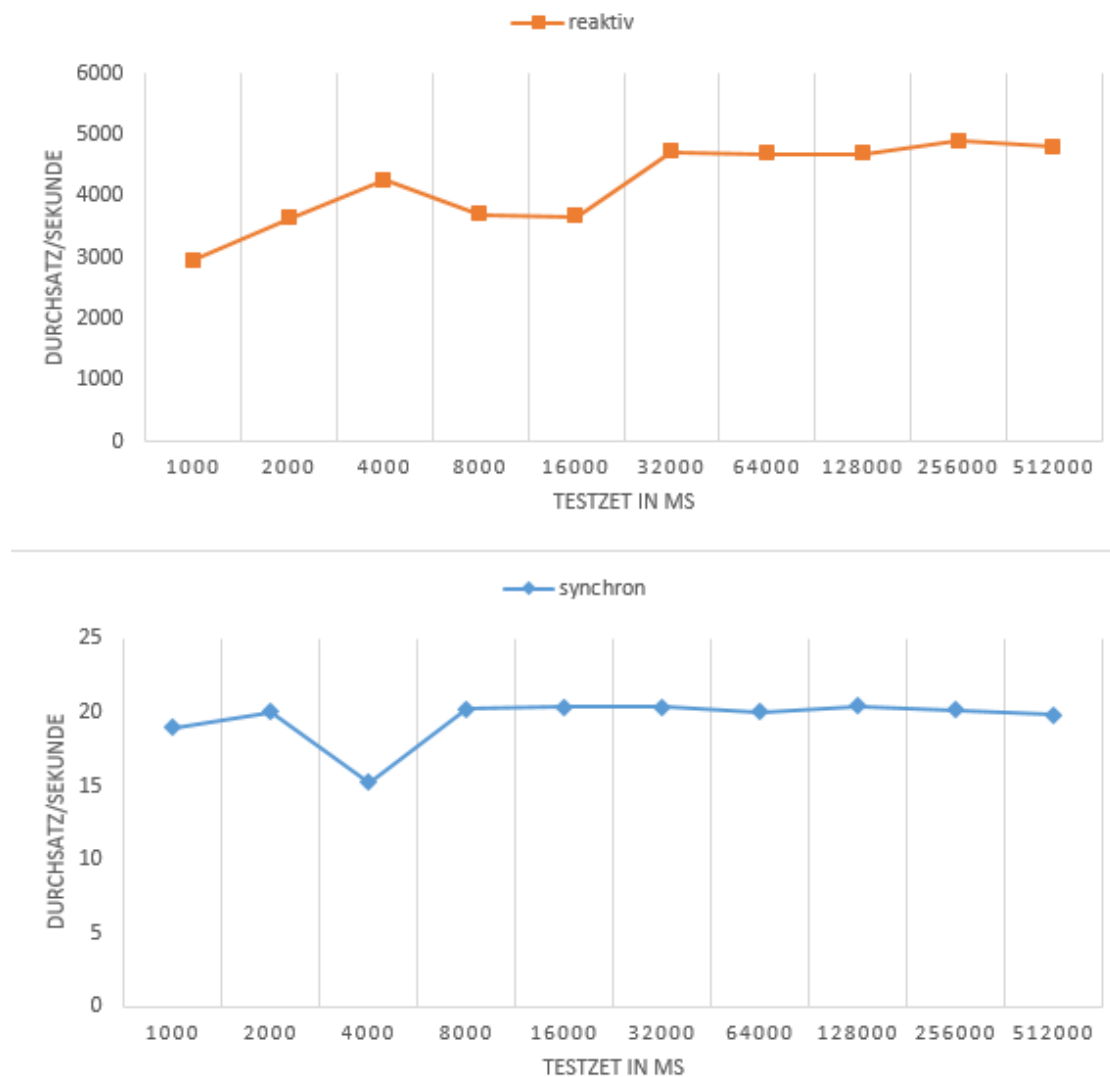
Abbildung 8.4: Synchrone Car und Server Kommunikation mit Sendeverzögerung



Messung ohne Verzögerung

Die Messung ohne Verzögerung soll die Bedingungen eines Servers unter Last simulieren, indem alle Clients so viele Messungen senden wie möglich. Die folgende Abbildung zeigt die Messung ohne Verzögerung, wobei der Maßstab unterschiedlich ist, da die reaktive Variante einen stark erhöhten Durchsatz hat:

Abbildung 8.5: Messung ohne Zeitverzögerung



Die reaktive Anwendung strebt einen konstanten Durchsatz von 5000 an während die synchrone Anwendung einen Durchsatz von 20 erreicht. Die folgende Tabelle zeigt

den Durchsatz pro Sekunde für die synchrone und reaktive Variante:

Tabelle 8.4: Messung ohne Zeitverzögerung (tabellarisch)

	Reaktiv	Synchron
Messzeit in ms	Durchsatz / Sekunde	
1000	2948	19
2000	3636,5	20
4000	4262,5	15,25
8000	3701,75	20,25
16000	3675,375	20,375
32000	4727,15625	20,375
64000	4696,484375	20,015625
128000	4683,921875	20,4375
256000	4906,941406	20,148438
512000	4797,521484	19,818359

Die hohe Differenz des Durchsatzes zwischen der reaktiven und der synchronen Variante entsteht dadurch, dass ein reaktives Car nicht-blockierend auf die Antwort des Servers wartet. Dadurch sendet das Car so viele Messungen wie möglich. Die reaktive Implementierung sendet bereits während der Bearbeitungszeit des Servers weitere Anfragen, sodass ein höherer Durchsatz möglich ist. Die folgende, Abbildung demonstriert die Kommunikation zwischen Car und Server in reaktiver als auch synchroner Variante:

Abbildung 8.6: Reaktive Car und Server Kommunikation ohne Sendeverzögerung

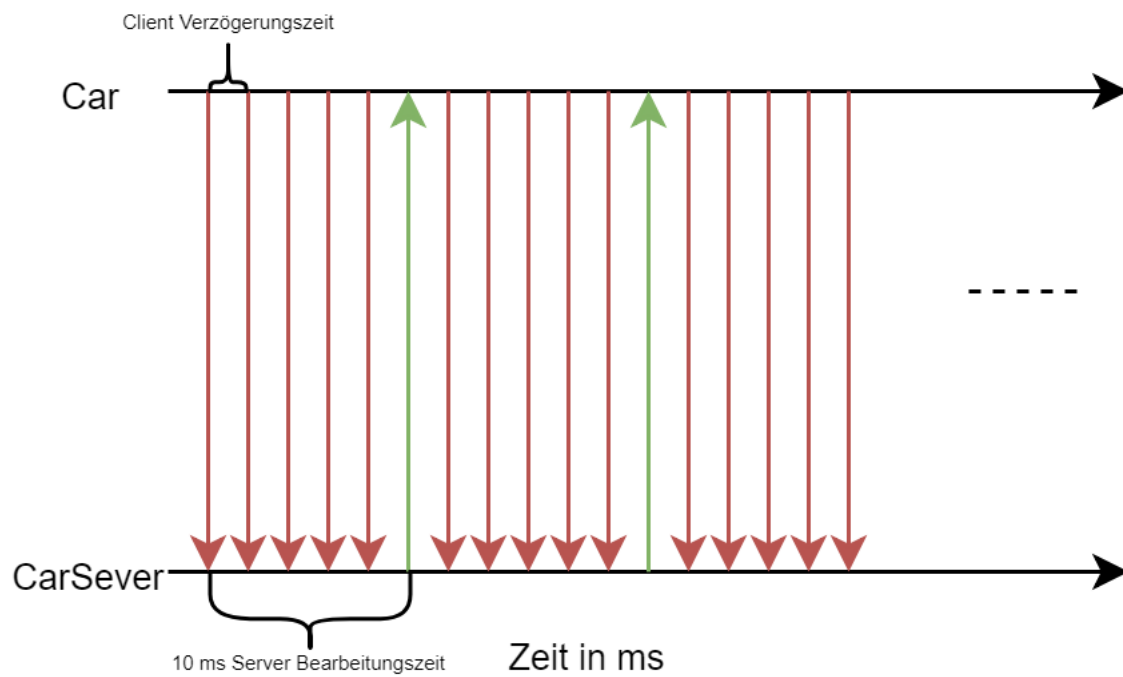
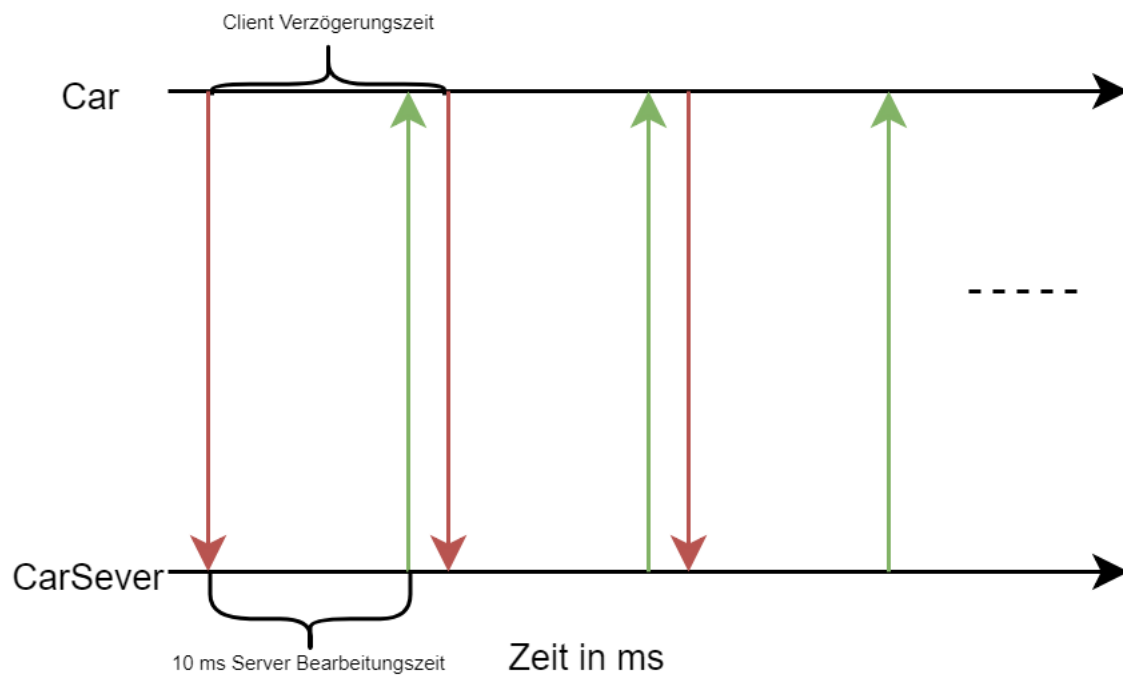


Abbildung 8.7: Synchrone Car und Server Kommunikation ohne Sendeverzögerung



Ergebnis

Die reaktive Implementierung ist deutlich performanter wie die synchrone Variante. Ohne Sendeverzögerung kann ein Client mehr als 4500 Messungen mehr versenden als ein synchroner Client. Die Geschwindigkeit ergibt sich maßgeblich aus dem blockierenden Verhalten der Anwendung. Selbst unter gedrosselten Bedingungen (200 ms Sendeverzögerung) schafft der reaktive Client einen Durchsatz nahe am Optimum.

9

Fazit

Reaktivität ist ein Begriff der im wesentlichen in zwei Bereichen der Softwareentwicklung stark an Popularität gewonnen hat. Zum einen in reaktiven Systemen und zum anderen in reaktiver Programmierung. Während ein reaktives System eine Architektur beschreibt, die eine hochgradig antwortbereite Systeme ermöglicht, meint die reaktive Programmierung ein Programmierparadigma, mit dem asynchrone und nicht-blockierende Anwendungen entwickelt werden können.

Die Streams sind als de facto Standard in der reaktiven Programmierung anzusehen, denn sie haben wie das reaktive Manifest ebenfalls eine Standardisierung, die Reactive Streams Specification. Anders wie Streams aus der Collection API von Java, können reactive Streams nicht-blockierend Daten mit Back Pressure verarbeiten. Das ermöglicht eine Skalierung in vertikaler Ebene, die mit den Java Streams nicht ohne weiteres nicht möglich ist. So gibt es zwar Techniken wie Futures, die sogar Teilaspekte der Reactive Streams anbieten. Allerdings können Futures keine Sequenzen von Daten übertragen und müssen den Rückgabewert über Callback-Funktionen verarbeiten. Diese Art der Verarbeitung führt häufig zur so genannten Callback-Hell. Reactive Streams lösen diese Probleme, jedoch bietet sich deren Verwendung nicht in jedem Fall an. Sind einzelne Anfragen und blockierender Code nützlicher, sollte überlegt werden, ob sich der erhöhte Lernaufwand für den Anwendungsfall lohnt. In Anwendungen, die parallel und nicht-blockierend laufen, lassen sich mit Reactive Streams verständliche und übersichtliche Algorithmen entwickeln.

Vorreiter auf dem Gebiet der Reactive Streams in Java ist Netflix durch die Entwicklung der RxJava Bibliothek. Diese implementiert die Reactive Streams Specification und vereinfacht die nebenläufige Programmierung durch eine deklarative API maßgeblich.

In der Android-Welt erfreut sich RxJava großer Beliebtheit und gilt dort als Standardbibliothek für reaktive Programmierung. Unternehmen wie Pivotal und Lightbend bieten mit ähnlichen Bibliotheken wie etwa Reactor3 und Akka-Streams (als Teil von Akka) Alternativen.

Von den drei betrachteten Bibliotheken erwies sich Reactor als einsteigerfreundlichste und am besten bedien- und benutzbare Bibliothek. Dabei zeichnet sich Reactor vor allem durch eine schlanke API und eine verständliche Dokumentation aus. Allerdings ist diese Bibliothek nicht so weit verbreitet und findet vorzugsweise in Spring-Projekten Anwendung. Zu den reaktiven Bibliotheken gibt es allerdings auch Frameworks, die mehrere Technologien in sich vereinen und es sogar ermöglichen, ganze reaktive Systeme im Sinne des reaktiven Manifests zu bauen. Das beliebteste Framework unter Entwicklern ist hierbei Spring, das im Webflux Modul die Reactor-Bibliothek nutzt und in Konkurrenz zu Akka und VertX steht. Besonders Akka hat in den letzten Jahren stark an Popularität gewonnen, da es als Toolkit eine große Auswahl an Modulen aufweist und mit Scala eine Alternative zu Java bietet.

Die Entwicklung des reaktiven Prototyps zeigt im Wesentlichen zwei Dinge auf: Zum einen sind reaktive Programme komfortabler zu programmieren, zum anderen sind sie performanter als blockierende Programme.

Zusammengefasst lässt sich sagen, dass reaktive Programmierung eine moderne und sichere Art ist, nebenläufig zu programmieren. Reaktive Bibliotheken sind bereits Teil der größten Frameworks auf dem Markt und ermöglichen es, stabile reaktive Systeme zu bauen. Es ist zu erwarten, dass der Trend zur reaktiven Programmierung weiter anhält und den Standard für nebenläufige, nicht-blockierende Anwendungen setzt.

Nicht zuletzt ist auch die Tatsache, dass die Java 9 Flow-API als fester Bestandteil in das JDK aufgenommen wurde, ein deutlicher Indikator für die momentane und künftige Bedeutung der reaktiven Programmierung.

Anhang

A.1. Quellcode

A.1.1. Reactive Server Implementierung

Listing A.1: Reactive Server Implementation

```
1 package prototype.async.server;
2
3 import io.rsocket.AbstractRSocket;
4 import io.rsocket.Payload;
5 import io.rsocket.RSocketFactory;
6 import io.rsocket.transport.netty.server.TcpServerTransport;
7 import org.reactivestreams.Publisher;
8 import reactor.core.Disposable;
9 import reactor.core.publisher.Flux;
10 import reactor.core.publisher.Mono;
11
12 import java.net.InetSocketAddress;
13 import java.time.Duration;
14 import java.util.Scanner;
15
16 public class CarServer {
17     private InetSocketAddress socketAddress;
18     private Disposable channel;
19
20     /**
21      * Initialize Host Information like IP and Port.
22      * @param socketAddress Contains Information about IP and Port.
23      */
24     private CarServer(InetSocketAddress socketAddress){
25         this.socketAddress = socketAddress;
26     }
27
28     /**
29      * Instantiate Server Socket in channel Variable. The receive Method
30      * returns a ServerRSocketFactory whcih is configured down below.
31      * The acceptor takes a Lambda for a SocketAcceptor Class which returns
```

```

    an AbstractSocket Instance. The AbstractSocket Instance override's
    one of the four Interaction Models for RSocket: channel.
31  * The requestChannel takes a Publisher, transforms it to a Flux,
    configure a delay for each Measurement and return it to the Caller
    respective Car Instance.
32  * The transport Methode configures the Protocol TCP, because it is a
    more stable than UDP.
33  * The start Method returns a Mono <Disposable> of a successful
    established hosted Server in a blocking manner because we only have
    one Server.
34  */
35  public void receive() {
36      channel = RSocketFactory.receive()
37          .acceptor((setupPayload, reactiveSocket) ->
38              Mono.just(new AbstractRSocket() {
39                  @Override
40                  public Flux<Payload> requestChannel(Publisher<
41                      Payload> payloads) {
42                      return Flux.from(payloads)
43                          .flatMapSequential(
44                              payload -> Flux.just(payload)
45                                  .delaySequence(Duration.
46                                      ofMillis(10)));
47                  }
48              })))
49      .transport(TcpServerTransport.create(socketAddress.
50          getHostName(), socketAddress.getPort()))
51      .start()
52      .block();
53  }
54
55  /**
56   * Dispose from incoming Connections.
57   */
58  public void close() {
59      channel.dispose();
60  }
61
62  public static void main(final String... args){
63      Scanner sc = new Scanner(System.in);
64      CarServer carServer = new CarServer(new InetSocketAddress("127.0.0.1"
65          , 1337));
66      carServer.receive();
67
68      System.out.println("Type 'close' to terminate the CarServer:");
69      while(true){
70          String input = sc.nextLine();

```

```

66         switch(input){
67             case "close": carServer.close();
68             return;
69             default:
70                 System.err.println("Try again ...");
71         }
72     }
73
74 }
75 }

```

A.1.2. Reactive Client Implementierung

Listing A.2: Reactive Client Implementation

```

1  package prototype.async.client;
2
3  import io.rsocket.RSocket;
4  import io.rsocket.RSocketFactory;
5  import io.rsocket.transport.netty.client.TcpClientTransport;
6  import io.rsocket.util.DefaultPayload;
7  import prototype.utility.Serializer;
8  import reactor.core.Disposable;
9  import reactor.core.publisher.Flux;
10 import prototype.routing.RoutingFactory;
11 import java.net.InetSocketAddress;
12 import java.time.Duration;
13
14 public class Car {
15     private final InetSocketAddress socketAddress;
16     private final RoutingFactory routingFactory = new RoutingFactory();
17     private RSocket client;
18     private Disposable serverEndpoint;
19     private final CarConfiguration carConfiguration;
20
21     /**
22      * The Flowrate define how many Requests are processed successfully.
23      */
24     private int flowrate = 0;
25
26     /**
27      * Initialize Car Instance with default CarConfiguration
28      * @param socketAddress Contains the IP and Port for corresponding Server
29      */
30     public Car(InetSocketAddress socketAddress) {
31         this.carConfiguration = new CarConfiguration();
32         this.socketAddress = socketAddress;

```

```
33     }
34
35     /**
36      *
37      * @param socketAddress Contains the IP and Port for corresponding Server
38      * @param configuration The Custom Configuration for a Car Instance
39      */
40     public Car(InetSocketAddress socketAddress, CarConfiguration
        configuration) {
41         this.socketAddress = socketAddress;
42         this.carConfiguration = configuration;
43     }
44
45     /**
46      * This Method connect to the reactive Server.
47      * The keep-alive is set to 30 Minutes in case the Connection stays
        longer (Benchmark).
48      * The transport protocol is TCP, the connection safety is guaranteed.
49      * Because each connection is only once, the return value of start(...)
        can be get in a blocking manner.
50      */
51     public void connect() {
52         this.client = RSocketFactory
53             .connect()
54             .keepAliveAckTimeout(Duration.ofMinutes(30))
55             .transport(TcpClientTransport.create(socketAddress.
                getHostName(), socketAddress.getPort()))
56             .start()
57             .block();
58     }
59
60     /**
61      *
62      * @return Integer of current Flowrate
63      */
64     public int getFlowrate() {
65         return flowrate;
66     }
67
68     /**
69      * This Method request a Channel between Server and Client by giving a
        Stream as an Argument. The Stream is the datasource.
70      * The Flux (Stream) emit elements from Iterable of a List of
        Measurements (Coordinate and Signal Strength). It will be repeated
        about 100.000 times
71      * to simulate an endless emission. Each Measure Signal is delayed by
        particular delay in ms. Each emitted Measure is mapped to a Payload
```

```

    Type from RSocket.
72  * Because this datasource is able to be subscribed multiple times the
    Stream is hot by calling the share() function.
73  */
74  public void send() {
75      serverEndpoint = client.requestChannel(
76          Flux.fromIterable(routingFactory.getRoutingType(
77              carConfiguration.ROUTETYPE).getRouteAsList())
78              .repeat(100_000)
79              .delayElements(carConfiguration.DELAY)
80              .doOnNext(measurements -> measurements
81                  .setSignalStrength(((int) (Math.random() * 10))))
82              .map(measurements -> DefaultPayload.create(
83                  Serializer.serialize(measurements)))
84              .share()
85          ).subscribe(payload -> ++flowrate);
86  }
87  /**
88  * This Method disposes from the Response Stream from the Server.
89  */
90  public void close() {
91      serverEndpoint.dispose();
92  }
93  }

```

A.1.3. Socket Server Implementierung

Listing A.3: Socket Server Implementation

```

1  package prototype.sync.server;
2  import prototype.model.Measurement;
3  import java.io.*;
4  import java.net.*;
5  import java.util.*;
6  import java.util.concurrent.CompletableFuture;
7  import java.util.concurrent.ExecutorService;
8  import java.util.concurrent.Executors;
9
10 public class CarServer {
11     private final InetAddress socketAddress;
12     private ServerSocket serverSocket;
13     private ExecutorService executorService = Executors.newFixedThreadPool(8)
14         ;
15
16     /**
17     * Initialize Host Information like IP and Port.

```

```

17     * @param socketAddress Contains Information about IP and Port.
18     */
19     private CarServer(InetSocketAddress socketAddress){
20         this.socketAddress = socketAddress;
21     }
22
23     /**
24     * This (asynchronous) Method handles incoming Connections and incoming
25     * Measurements from Car Instances.
26     * Each Car Socket is swapped to a Thread (Future) where incoming
27     * Measurements are handled in a blocking, synchronous manner.
28     *
29     * @throws IOException Occur when accepting incoming Connection fail.
30     */
31     public void receive() throws IOException {
32         this.serverSocket = new ServerSocket();
33         serverSocket.bind(socketAddress);
34         CompletableFuture.runAsync(() -> {
35             while(!serverSocket.isClosed()){
36                 Socket clientSocket;
37                 try {
38                     clientSocket = serverSocket.accept();
39                 } catch (IOException e) {
40                     System.err.println("[CarServer] " + e.getMessage());
41                     return;
42                 }
43
44                 Socket finalClientSocket = clientSocket;
45                 CompletableFuture.runAsync(() -> {
46                     ObjectInputStream ois;
47                     ObjectOutputStream oos;
48                     while(true) {
49                         try {
50                             assert finalClientSocket != null;
51                             ois = new ObjectInputStream(new
52                                 BufferedInputStream(finalClientSocket.
53                                     getInputStream()));
54                             Measurement measurement = (Measurement) ois.
55                                 readObject();
56                             // do expensive work
57                             Thread.sleep(10);
58                             oos = new ObjectOutputStream(new
59                                 BufferedOutputStream(finalClientSocket.
60                                     getOutputStream()));
61                             oos.writeObject(measurement);
62                             oos.flush();
63                         } catch (IOException | ClassNotFoundException |

```

```

        InterruptedException e) {
57         System.err.println("[CarServer] Connection to
            Client was dropped " + e.getMessage());
58         break;
59     }
60
61     }
62     }, executorService);
63
64     }
65     }, executorService);
66
67 }
68
69 /**
70  * Close the Server Socket and shutdown the Executor Threads.
71  */
72 public void close() {
73     try {
74         serverSocket.close();
75     } catch (IOException e) {
76         e.printStackTrace();
77     }
78     executorService.shutdown();
79 }
80
81 public static void main(final String... args) throws IOException {
82     Scanner sc = new Scanner(System.in);
83     CarServer carServer = new CarServer(new InetSocketAddress("127.0.0.1"
84         , 1337));
85     carServer.receive();
86
87     System.out.println("Type 'close' to terminate the CarServer:");
88     while(true){
89         String input = sc.nextLine();
90         switch(input){
91             case "close": carServer.close();
92             return;
93             default:
94                 System.err.println("Try again...");
95         }
96     }
97 }

```

A.1.4. Socket Client Implementierung

Listing A.4: Socket Client Implementation

```

1  package prototype.sync.client;
2
3  import prototype.async.client.CarConfiguration;
4  import prototype.model.Measurement;
5  import prototype.routing.RoutingFactory;
6
7  import java.io.*;
8  import java.net.InetSocketAddress;
9  import java.net.Socket;
10 import java.util.List;
11
12 public class Car {
13     private final InetSocketAddress socketAddress;
14     private Socket clientSocket;
15     private CarConfiguration carConfiguration;
16     private RoutingFactory routingFactory = new RoutingFactory();
17     private List<Measurement> route;
18     /**
19      * The Flowrate define how many Requests are processed successfully.
20      */
21     private int flowrate = 0;
22     private boolean done = false;
23
24
25     /**
26      * Initialize Car with Adress, default Car Configuration and default
27      * Route.
28      * @param socketAddress Contains the IP and Port for corresponding Server
29      */
30     public Car(InetSocketAddress socketAddress) {
31         this.socketAddress = socketAddress;
32         carConfiguration = new CarConfiguration();
33         this.route = routingFactory.getRoutingType(carConfiguration.ROUTETYPE)
34             .getRouteAsList();
35     }
36
37     /**
38      * Initialize Car with Adress, custom Car Configuration and custom Route
39      * .
40      * @param socketAddress Contains the IP and Port for corresponding Server
41      * @param configuration The Custom Configuration for a Car Instance
42      */
43     public Car(InetSocketAddress socketAddress, CarConfiguration
44         configuration) {
45         this.socketAddress = socketAddress;
46         this.carConfiguration = configuration;

```

```
43         this.route = routingFactory.getRoutingType(carConfiguration.ROUTETYPE
44             ).getRouteAsList();
45     }
46     /**
47     * Establish Connection between Car Socket and Server Socket.
48     * The socketaddress as the IP and the Port for corresponding Server.
49     */
50     public void connect() {
51         clientSocket = new Socket();
52         try {
53             clientSocket.connect(socketAddress);
54         } catch (IOException e) {
55             System.err.println("[Car] cannot connect Host " + e.getMessage())
56                 ;
57         }
58     }
59
60     /**
61     *
62     * @return Integer of current Flowrate
63     */
64     public int getFlowrate() {
65         return flowrate;
66     }
67
68     /**
69     * This Method closes the socket to the Server.
70     */
71     public void close() {
72         if(clientSocket != null){
73             done = true;
74             try {
75                 clientSocket.close();
76             } catch (IOException e) {
77                 e.printStackTrace();
78             }
79         }
80     }
81
82     /**
83     * Send 100.000.000 Measurements as Simulation of infinite Emission.
84     * Each Emission is delayed by a function delay() and increment a counter
85     * which controls when the Emission is done.
86     * Each Measurement gets a random Singal Strength
87     */
```

```

87     public void send() {
88         int deliveredElements = 0;
89         int MAXELEMENTS = 100_000_000;
90         do{
91             for (Measurement measurement : route) {
92                 measurement.setSignalStrength((int)(Math.random()*10));
93                 sendData(measurement);
94                 delay();
95                 deliveredElements++;
96             }
97         } while(deliveredElements < MAXELEMENTS && !done);
98     }
99
100     /**
101     * Like send() Method but with a lower defined border of Emissions.
102     * @param elements number of Measurements which are send.
103     */
104     public void send(int elements){
105         int deliveredElements = 0;
106         do{
107             for (Measurement measurement : route) {
108                 measurement.setSignalStrength((int)(Math.random()*10));
109                 sendData(measurement);
110                 deliveredElements++;
111             }
112         } while(deliveredElements < elements);
113     }
114
115     /**
116     * Send Measurement via OutputStream to CarServer.
117     * Receive Measurement vis InputStream.
118     * Increment flowrate for analysis purpose.
119     * @param measurement which is send
120     */
121     private void sendData(Measurement measurement){
122         try {
123             // send
124             ObjectOutputStream oos = new ObjectOutputStream(new
125                 BufferedOutputStream(clientSocket.getOutputStream()));
126             oos.writeObject(measurement);
127             oos.flush();
128
129             // receive
130             ObjectInputStream ois = new ObjectInputStream(new
131                 BufferedInputStream(clientSocket.getInputStream()));
132             Measurement c = (Measurement) ois.readObject();

```

```

132         // increment flowrate
133         ++flowrate;
134     } catch (IOException | ClassNotFoundException e) {
135         e.printStackTrace();
136     }
137 }
138
139 /**
140  * Thread sleep delay from CarConfiguration
141  */
142 private void delay() {
143     try {
144         Thread.sleep(carConfiguration.DELAY.toMillis());
145     } catch (InterruptedException e) {
146         e.printStackTrace();
147     }
148 }
149 }

```

A.1.5. Test der reaktiven Variante

Listing A.5: Scalatest der reaktiven Variante

```

1 package benchmark
2
3 import java.net.InetSocketAddress
4 import java.util.concurrent.Executors
5
6 import com.typesafe.scalalogging.Logger
7 import prototype.async.client.{Car, CarConfiguration}
8
9 import scala.concurrent.{ExecutionContext, ExecutionContextExecutor, Future}
10 import ExecutionContext.Implicits.global
11 /**
12  * Implementation of Simulation Trait
13  *
14  * @param hostInfo Contains Information about IP and Port from Server with
15  *   default
16  */
17 class AsyncSimulation(hostInfo: InetSocketAddress = new InetSocketAddress("
18     127.0.0.1", 1337)) extends Simulation{
19     val logger: Logger = Logger[AsyncSimulation]
20     val executors: ExecutionContextExecutor = ExecutionContext.fromExecutor(
21         Executors.newFixedThreadPool(8))
22
23     /**
24      * This List stores Measuretimes for each Benchmark
25      */
26 }

```

```

22     val durationList: List[Int] = List range(0,10) map(n => Math.pow(2, n
        toDouble).toInt * 1000)
23
24     /**
25      * This Method runs a Benchmark
26      * @param config Configuration for the Car (See Javadocs)
27      */
28     def run(config: CarConfiguration = new CarConfiguration()): Unit = {
29         logger info "START TEST"
30         warmUp()
31         Future sequence(durationList map(duration => Future((duration,
            benchmark(duration, config))(executors))) onComplete(
32             result => {
33                 printResult(result get)
34                 saveResult("ReactiveBenchmarkResult.txt", result get)
35                 logger info "END TEST"
36             }
37         )
38     }
39
40     /**
41      * This method runs a benchmark by instantiating a Car Instance. Then it
        connects to the Server and starts sending Measurements asynchronously
        .
42      * After a particular time: measuretime the benchmark stops and return the
        current Value of the flowrate in the Car.
43      * @param measuretime The time the benchmark claims
44      * @param config Configuration for the Car (See Javadocs)
45      * @return
46      */
47     def benchmark(measuretime: Int, config: CarConfiguration): Int ={
48         logger.info("START BENCHMARK")
49         val car = new Car(hostInfo)
50         car connect()
51         car send()
52         Thread sleep measuretime
53         logger info ("Throughput: {} processed requests", car getFlowrate)
54         logger info "END BENCHMARK"
55         car close()
56         car getFlowrate
57     }
58
59     /**
60      * This Method works like benchmark() and guarantees a JIT-optimized
        compilation of Methods which are called during the Benchmark.
61      * The only difference is, that it send a specific number of Measurements.
        In this Case 1500 because of the JVM Flag -client.

```

```

62     * Be aware that your results may differ when you let the Flags away
        because some methods are not JIT-compiled during the benchmark.
63     * The default Threshold for Compilation is 10.000 https://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html
64     */
65     def warmUp(): Unit={
66         logger.info("WARMUP")
67         val iterations = 1500
68         val car = new Car(hostInfo)
69         car connect()
70         car send()
71         while(car.getFlowrate < iterations){
72             Thread sleep 1
73         }
74         car close()
75     }
76 }
77
78 object AsyncSimulation{
79     // Be sure you running this benchmark on -client JVM Argument otherwise no
        optimazation is guaranteed
80     def main(args: Array[String]): Unit = {
81         val simulation = new AsyncSimulation()
82         simulation.run()
83     }
84 }

```

A.1.6. Test der synchronen Variante

Listing A.6: Scalatest der synchronen Variante

```

1  package benchmark
2
3  import java.net.InetSocketAddress
4  import java.util.concurrent.Executors
5  import com.typesafe.scalalogging.Logger
6  import prototype.async.client.CarConfiguration
7  import prototype.sync.client.Car
8
9  import scala.concurrent.{ExecutionContext, ExecutionContextExecutor, Future}
10 import ExecutionContext.Implicits.global
11
12 class SyncSimulation(hostInfo: InetSocketAddress = new InetSocketAddress("
    127.0.0.1", 1337)) extends Simulation {
13     val logger: Logger = Logger[SyncSimulation]
14     val executors: ExecutionContextExecutor = ExecutionContext.fromExecutor(
        Executors.newFixedThreadPool(8))

```

```
15  val durationList: List[Int] = List range(0,10) map(n => Math.pow(2, n
    toDouble).toInt * 1000)
16
17  def run(config: CarConfiguration = new CarConfiguration()): Unit = {
18    logger.info("START TEST")
19    warmUp()
20
21    Future sequence(durationList map(duration => Future((duration, benchmark(
    duration, config)))))) onComplete(
22      result => {
23        printResult(result.get)
24        saveResult("SyncBenchmarkResult.txt", result.get)
25        logger.info("END TEST")
26      })
27  }
28
29  override def benchmark(runtime: Int, config: CarConfiguration): Int = {
30    logger.info("START BENCHMARK")
31    val car = new Car(hostInfo, config)
32
33    Future {
34      car.connect()
35      car.send()
36    }(executors)
37    Thread.sleep(runtime)
38    logger.info("Throughput: {} processed requests", car.getFlowrate)
39    logger.info("END BENCHMARK")
40    car.close()
41    car.getFlowrate
42  }
43
44  def warmUp(): Unit = {
45    logger.info("START WARMUP")
46    val car = new Car(hostInfo)
47    car.connect()
48    car.send(1500)
49    car.close()
50    logger.info("END WARMUP")
51  }
52 }
53
54 object SyncSimulation {
55   def main(args: Array[String]): Unit = {
56     val simulation = new SyncSimulation()
57     simulation.run()
58   }
59 }
```

```
60 }
```


Quellenverzeichnis

- [CH13] Ben Christensen und Jafar Husain. *Reactive Programming in the Netflix API with RxJava*. Website. [Online; Stand 18. Dezember 2018]. Feb. 2013. URL: <https://medium.com/netflix-techblog/reactive-programming-in-the-netflix-api-with-rxjava-7811c3a1496a>.
- [al14] Jonas Bonér et al. *Das Reaktive Manifest*. Website. [Online; Stand 22. November 2018]. 2014. URL: <https://www.reactivemanifesto.org/de>.
- [Dr 15] Dr. Kristine Schaal Dr. Rüdiger Grammes. "Stau an der richtigen Stelle mit Backpressure". ngerman. In: *Javaspektrum*. Mai 2015. URL: https://www.sigs-datacom.de/uploads/tx_dmjournals/grammes_schaal_JS_05_15_vYon.pdf.
- [BK17] Jonas Bonér und Viktor Klang. *Reactive Programming versus Reactive Systems*. Techn. Ber. Lightbend Inc., Jan. 2017.
- [Git17] Github. *Reactive Streams*. Website. [Online; Stand 9. Oktober 2018]. Dez. 2017. URL: <https://github.com/reactive-streams/reactive-streams-jvm/blob/v1.0.2/README.md>.
- [Jog17] Tejaswini Mandar Jog. "Reactive Programming With Java 9". In: Packt Publishing, 2017. Kap. 1. introduction to Reactive Programming.
- [Mal17] Konrad Malawski. "Why Reactive? Foundational Principles for Enterprise Adoption". In: O'Reilly Media, Inc., 2017. Kap. 1. Introduction.
- [Red17] K. Siva Prasad Reddy. "Beginning Spring Boot 2". In: Apress, 2017. Kap. 12. Reactive Programming Using Spring WebFlux.
- [Ste17] Benedikt Stemmildt. *Going Reactive - Eine Einführung in die reaktive Programmierung*. code.talks. 2017. URL: <https://www.youtube.com/watch?v=YnladsQPqYk>.
- [Tom17a] Ben Christensen Tomasz Nurkiewicz. "Reactive Programming with RxJava". In: O'Reilly Media, Inc., 2017. Kap. 1. Reactive Programming with RxJava.
- [Tom17b] Ben Christensen Tomasz Nurkiewicz. "Reactive Programming with RxJava". In: O'Reilly Media, Inc., 2017. Kap. Foreword.
- [Lig18a] Inc. Lightbend. *Akka Modules*. Website. Version 2.5.18. [Online; Stand 21. November 2018]. 2018. URL: <https://doc.akka.io/docs>.

- [Lig18b] Inc. Lightbend. *Core Concepts*. Website. Version 2.5.3. [Online; Stand 07. Dezember 2018]. 2018. URL: <https://doc.akka.io/docs/akka/2.5.3/scala/stream/stream-flows-and-basics.html>.
- [Lig18c] Inc. Lightbend. *Relationship with Reactive streams*. Website. [Online; Stand 2. Oktober 2018]. 2018. URL: <https://doc.akka.io/docs/akka/current/stream/stream-introduction.html#relationship-with-reactive-streams>.
- [Lig18d] Inc. Lightbend. *Streams*. Website. Version 2.5.21. [Online; Stand 25. Februar 2019]. 2018. URL: <https://doc.akka.io/docs/akka/current/stream/index.html?language=java>.
- [PSa18] Julien Ponge, Thomas Segismont und Julien Viet et al. "A gentle guide to asynchronous programming with Eclipse Vert.x for Java developers". In: Vert.X, 2018. Kap. Chapter 1. Introduction.
- [Rao18a] Alan Mycroft Raoul-Gabriel Urma Mario Fusco. "Modern Java in Action". In: Manning, 2018. Kap. 17. Reactive Programming.
- [Rao18b] Alan Mycroft Raoul-Gabriel Urma Mario Fusco. "Modern Java in Action". In: Manning, 2018. Kap. 16. CompletableFuture: composable asynchronous programming.
- [Wik18a] Wikipedia. *Reaktive Programmierung* — *Wikipedia, Die freie Enzyklopädie*. ngerman. [Online; Stand 14. September 2018]. 2018. URL: https://de.wikipedia.org/w/index.php?title=Reaktive_Programmierung&oldid=177533954.
- [Wik18b] Wikipedia contributors. *GitHub* — *Wikipedia, The Free Encyclopedia*. Website. [Online; Stand 7. Dezember 2018]. 2018. URL: <https://en.wikipedia.org/w/index.php?title=GitHub&oldid=871677636>.
- [Wik18c] Wikipedia contributors. *Reactive Streams* — *Wikipedia, The Free Encyclopedia*. Website. [Online; Stand 18. September 2018]. 2018. URL: https://en.wikipedia.org/w/index.php?title=Reactive_Streams&oldid=860080694.
- [Dav19] Adam L. Davis. "Reactive Streams in Java Concurrency with RxJava, Reactor, and Akka Streams". In: Apress, 2019. Kap. Chapter 3. Common Concepts.
- [Gita] Github. Website. [Online; Stand 16. Februar 2019]. URL: <https://github.com/search?l=Java&q=rxjava&type=Repositories>.
- [Gitb] Github. Website. [Online; Stand 16. Februar 2019]. URL: <https://github.com/search?l=Java&q=reactor&type=Repositories>.

- [Gitc] Github. Website. [Online; Stand 16. Februar 2019]. URL: <https://github.com/search?l=Scala&q=Akka+Stream&type=Repositories>.
- [Gitd] Github. *About Stars*. Website. [Online; Stand 16. Februar 2019]. URL: <https://help.github.com/articles/about-stars/>.
- [gita] gitstar-ranking.com. Website. [Online; Stand 16. Februar 2019], [Letzte Aktualisierung; 8. Dezember 2018, 07:28 Uhr]. URL: <https://gitstar-ranking.com/ReactiveX/RxJava>.
- [gitb] gitstar-ranking.com. Website. [Online; Stand 20. Dezember 2018], [Letzte Aktualisierung; 4. April 2018, 16:50 Uhr]. URL: <https://gitstar-ranking.com/reactor/reactor>.
- [gitc] gitstar-ranking.com. Website. [Online; Stand 16. Februar 2019], [Letzte Aktualisierung; 5. September 2017, 17:04 Uhr]. URL: <https://gitstar-ranking.com/akka/akka>.
- [Kar] David Karnok. online. [Online; Stand 16. Februar 2019], [Erstellt am 10. September 2016 um 5:49 Uhr]. URL: <https://twitter.com/akarnokd/status/774590596740685824>.
- [Liga] Lightbend. *America's Largest Telecommunications Provider Goes Reactive*. Website. [Online; Stand 4. Januar 2019]. URL: <https://www.lightbend.com/case-studies/americas-largest-telecommunications-provider-goes-reactive>.
- [Ligb] Lightbend. *PayPal Blows Past 1 Billion Transactions Per Day Using Just 8 VMs With Akka, Scala, Kafka and Akka Streams*. Website. [Online; Stand 4. Januar 2019]. URL: <https://www.lightbend.com/case-studies/paypal-blows-past-1-billion-transactions-per-day-using-just-8-vms-and-akka-scala-kafka-and-akka-streams>.
- [Ligc] Inc. Lightbend. *Packages*. Website. [Online; Stand 25. Februar 2019]. URL: https://doc.akka.io/japi/akka/current/index.html?akka/stream/package-summary.html&_ga=2.134194124.394401883.1544449904-1435486093.1536821504.
- [Ligd] Inc. Lightbend. *Streams Quickstart Guide*. Website. Version 2.5. [Online; Stand 21. Februar 2019]. URL: <https://doc.akka.io/docs/akka/2.5/stream/stream-quickstart.html>.
- [Maia] ReactiveX Maintainer. *Home*. Website. [Online; Stand 20. Dezember 2018]. URL: github.com/ReactiveX/RxJava/wiki.
- [Maib] ReactiveX Maintainer. *Observable*. Website. [Online; Stand 20. Dezember 2018]. URL: <https://github.com/ReactiveX/RxJava/wiki/Observable>.

- [Maic] ReactiveX Maintainer. *ReactiveX*. Website. [Online; Stand 20. Dezember 2018]. URL: <https://www.reactivex.io>.
- [Maid] ReactiveX Maintainer. *rxjava 2.2.4 API*. Website. [Online; Stand 20. Dezember 2018]. URL: <http://reactivex.io/RxJava/javadoc/>.
- [Maie] Reactor Maintainer. *Reactor 3 Reference Guide*. Website. [Online; Stand 16. Februar 2019]. URL: <https://projectreactor.io/docs/core/release/reference/>.
- [Maif] Reactor Maintainer. *Reactor Core 3.2.3.RELEASE*. Website. [Online; Stand 20. Dezember 2018]. URL: <https://projectreactor.io/docs/core/release/api/>.
- [Ora] Oracle. *Java HotSpot VM Options*. Website. [Online; Stand 14. Februar 2019]. URL: <https://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html>.
- [] *Reactive Streams*. Website. URL: <http://www.reactive-streams.org/>.
- [SR] Hartmut Schlosser und Marcel Richters. *JAXenter-Survey: Application Frameworks - 2018, 2017, 2016*. Website. [Online; Stand 17. Februar 2019]. URL: <https://jaxenter.de/wp-content/uploads/2018/05/JAXenter-Survey-2018-Application-Framework-Trends.png>.
- [Staa] StackOverflow. *Questions tagged [akka-stream]*. Website. [Online; Stand 3. Dezember 2018]. URL: <https://stackoverflow.com/questions/tagged/akka-stream?sort=active&pageSize=15>.
- [Stab] StackOverflow. *Questions tagged [rx-java2]*. Website. [Online; Stand 3. Dezember 2018]. URL: <https://stackoverflow.com/questions/tagged/rx-java2?sort=active&pageSize=15>.
- [tse] tsegismont. Website. [Online; Stand 21. Februar 2019] [Erstellt am 12. September 2018 um 10:23 Uhr]. URL: <https://stackoverflow.com/a/52292932/8676580>.
- [Ver] Eclipse Vert.X. *Who's using Vert.x?* Website. [Online; Stand 21. November 2018]. URL: <https://vertx.io/>.