Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Klaus Peter Warnkross

Message passing style concurrency anhand des Akka Aktorenframeworks

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Klaus Peter Warnkross

Message passing style concurrency anhand des Akka Aktorenframeworks

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Friedrich Esser
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 10. Oktober 2013

Klaus Peter Warnkross

Thema der Arbeit
Message passing style concurrency anhand des Akka Aktorenframeworks

Stichworte
Akka-Framework, Message Passing Concurrency, Aktorensysteme, Scala, Software Design, Simulationssoftware, Autonome Roboter

Kurzzusammenfassung
Diese Arbeit befasst sich mit dem Akka-Framework, das als Implementierung der Message Passing Concurrency eine alternative Methode für die Herstellung von Nebenläufigkeit in Software anbietet, als sie etwa die Shared State Concurrency darstellt.

Sie erläutert die wesentliche Funktionalität des Frameworks im Detail und weist auf Best Practices und Pattern bei der Benutzung hin.

Die dargestellten Techniken werden anschließend durch die Entwicklung einer Softwarelösung für das Problem der automatischen Warenkommissionierung durch autonome Roboter konkretisiert.

Klaus Peter Warnkross

Title of the paper
Message passing style concurrency exemplified by the Akka actor framework

Keywords
Akka framework, message passing concurrency, actor systems, Scala, software design, simulation software, autonomous robots

Abstract
This paper is concerned with the Akka framework, which as an implementation of message passing concurrency offers a method for the creation of concurrency in software that provides an alternative to that of the shared state concurrency.

It illustrates the essential functions of the framework in detail and points out best practices and patterns in its application.

Subsequently, the techniques outlined will be specified by the development of a software solution to the problem of the automatic commissioning of goods by autonomous robots.

Inhaltsverzeichnis

Abbildungsverzeichnis vi

1. Einleitung 1
- 1.1. Zielsetzung . . . 1
- 1.2. Abgrenzungen . . . 2
- 1.3. Gliederung . . . 2

2. Grundlagen 3
- 2.1. Methoden um Nebenläufigkeit zu erzeugen . . . 3
  - 2.1.1. Shared State Concurrency . . . 3
  - 2.1.2. Software Transactional Memory . . . 5
  - 2.1.3. Message Passing Concurrency . . . 5

3. Das Akka-Framework 8
- 3.1. Unterschiede zum ursprünglichen Actor Model . . . 8
- 3.2. Der Aktor . . . 8
  - 3.2.1. Definition . . . 9
  - 3.2.2. Erzeugung und Start . . . 12
  - 3.2.3. Senden von Nachrichten . . . 13
  - 3.2.4. Aktoren beenden . . . 14
  - 3.2.5. Hierarchische Struktur . . . 15
  - 3.2.6. Supervision . . . 16
  - 3.2.7. DeathWatch . . . 20
- 3.3. Futures und Dataflow . . . 21
- 3.4. Spezielle Aktoren . . . 28
  - 3.4.1. Finite State Machines . . . 28
  - 3.4.2. Router . . . 30
  - 3.4.3. Agents . . . 32
- 3.5. Das Aktorensystem . . . 33
  - 3.5.1. Event Stream . . . 33
  - 3.5.2. Dead Letters-Aktor . . . 34
  - 3.5.3. Scheduler . . . 34
  - 3.5.4. Dispatcher . . . 35
  - 3.5.5. Mailboxen . . . 37
  - 3.5.6. Remoting . . . 38
- 3.6. Weitere Komponenten . . . 40

4. Leitlinien für die Programmierung mit Akka und Design Pattern **41**
4.1. Leitlinien . . . 41
4.2. Design Pattern . . . 47
4.2.1. Aktoren-Initialisierung im laufenden Betrieb fortsetzen . . . 47
4.2.2. Request/Response mit Aktoren . . . 48
4.2.3. Algorithmen aufteilen . . . 49

5. Simulationssoftware einer Warenkommissionierung durch autonome Roboter **51**
5.1. Einleitung . . . 51
5.2. Funktionsweise der Simulation . . . 51
5.3. Abbildung der Realität und Designentscheidungen . . . 53
5.4. Implementierungsdetails . . . 54
5.4.1. Die Koordinaten / Artikel . . . 55
5.4.2. Komponente `Robots` . . . 56
5.4.3. Generieren der Bestellungen . . . 57
5.4.4. Komponente `Perception` . . . 58
5.4.5. Die Roboter . . . 59
5.5. Möglichkeiten für die Weiterentwicklung . . . 64

A. Anhang **66**
A.1. CD . . . 66

Abbildungsverzeichnis

3.1. Lebenszyklus von Aktoren 19

5.1. Warenlager mit Robotern bei Simulationsstart 51
5.2. Struktur der Softwarekomponenten 55

1. Einleitung

In den vergangenen Jahren ist der Bedarf an Nebenläufigkeit in Software stark gestiegen. Nachdem der Trend hin zu mehr Prozessorkernen anstatt GHz schon länger im Desktop und Notebook Segment bestand, ist er in der letzten Zeit auch vermehrt auf den Bereich der Mobilgeräte übergegangen und damit über alle praktisch relevanten Computersysteme hinweg dominant.

Sogar mobile Geräte wie Tablets und Smartphones haben heutzutage häufig CPUs, die über vier Rechenkerne verfügen, und der erste Octa Core-Prozessor für den Mobilbereich wurde unlängst vorgestellt[1].

Des Weiteren besteht durch den Trend, Cloud Computing-Dienste in Anspruch zu nehmen, ein erhöhter Bedarf an internetbasierten Systemen, die ein hohes Volumen an Nebenläufigkeit zu bewerkstelligen haben (sog. *Big Data* Computing).

Das Akka-Framework stellt eine Lösung für diese Probleme dar. Es ist eine Implementierung der aktorenbasierten Message Passing Concurrency und ermöglicht die Konstruktion von Programmen, die auf einer interaktiven Form der Concurrency basieren. Es legt dabei den Fokus auf Performanz, Skalierbarkeit und Fehlertoleranz.

Das Paradigma der Message Passing Concurrency ermöglicht es, das Problem auf eine andere Art zu lösen: statt Daten von verschiedenen Threads aus zugänglich zu machen, sind diese in einem Akka-Programm in sog. *Aktoren* gekapselt, auf die nur immer ein Thread zur selben Zeit Zugriff hat. Die Probleme, welche die Shared State Concurrency mit sich bringt, werden so vermieden und nicht nur durch Synchronisierung ausgebessert.

1.1. Zielsetzung

Diese Arbeit ist eine umfassende Einführung in das Akka-Framework und erklärt dessen Hauptbestandteile und wie diese effektiv eingesetzt werden, um nebenläufige Software zu realisieren. Um einen Überblick auf das Thema Nebenläufigkeit zu schaffen, werden die gängi-

[1] Siehe http://www.mediatek.com/_en/Event/201307_TrueOctaCore/tureOcta_.php

gen Concurrency-Paradigmen zunächst kurz vorgestellt, wobei insb. auf das der Shared State Concurrency eingegangen wird.

Am Ende der Lektüre sollte der Leser eine gute Vorstellung davon haben, wie die Struktur von Programmen aussieht, die im Message Passing Concurrency-Paradigma geschrieben sind und, aufgrund des Detailwissens in Akka, auch konkret in der Lage sein, solche zu schreiben.

1.2. Abgrenzungen

Während diese Arbeit entstand, befand sich das sog. *Cluster*-Feature des Akka-Frameworks noch in einem experimentellen Stadium. Deswegen ist es kein Gegenstand der Arbeit geworden.

Mittlerweile hat das Feature Einzug in eine finale Version von Akka gehalten und ist sehr beachtenswert, wenn man Software schreibt, die über mehrere Rechner hinweg arbeiten soll, da es eine Reihe von Verbesserungen für verteilte Systeme mit sich bringt.

1.3. Gliederung

In Kapitel 2 werden die derzeit populärsten Methoden, um Nebenläufigkeit in Software herzustellen, kurz vorgestellt.

In Kapitel 3 geht die Arbeit auf die Besonderheiten der Message Passing Concurrency in Form des Akka-Frameworks ein und beschreibt dessen Hauptbestandteile im Detail.

In Kapitel 4 wird auf Design Pattern und Best Practices bei der Verwendung von Akka hingewiesen.

In Kapitel 5 wird eine Simulationssoftware vorgestellt, die auf dem Akka-Framework basiert.

2. Grundlagen

2.1. Methoden um Nebenläufigkeit zu erzeugen

Es gibt eine Reihe von Techniken, mit denen sich eine Kommunikation von mehreren Threads untereinander herstellen lässt. Im Folgenden werden vier prominente Methoden mit ihren wesentlichen Eigenschaften vorgestellt.

2.1.1. Shared State Concurrency

Die derzeit dominierende Methode, um Nebenläufigkeit zu erzeugen, ist die Shared State Concurrency. [Lee (2006)] Sie ist u.a. in den Programmiersprachen Java und C# direkt durch die eingebauten Synchronisierungsprimitive verfügbar.

Die grundlegende Semantik besteht darin, dass von mehreren Threads aus auf Variablen zugegriffen werden kann und eine Kommunikation der Threads durch eine Veränderung dieser stattfindet. Der neue Datensatz wird vom System an alle Threads, die eine Sicht auf die Variable haben, weitergegeben, so dass alle per se über dieselbe Version des gemeinsamen Datensatzes verfügen.

Der Nichtdeterminismus von Threads

Threads unterliegen in ihrem Ablauf einem Nichtdeterminismus, da dieser von äußeren, nicht durch den Programmierer beeinflussbaren Faktoren abhängt. Einer dieser Faktoren ist bspw. der Scheduler des Betriebssystems, der über die chronologische Abfolge von Threads und über die Zeit entscheidet, die diese aktiv auf der CPU laufen dürfen. Ein weiterer Faktor sind Ein-/Ausgabe-Latenzen, bspw. bei Zugriffen auf eine Datenbank – blockiert ein Thread so lange, bis eine Antwort vorliegt, so hängt der Zeitpunkt seiner frühestmöglichen Reaktivierung von deren Eintreffen ab.

Das nichtdeterministische Ablaufverhalten von Threads führt dazu, dass die entsprechende Sequentialisierung der Operationen aus denen sich der Ablauf der Threads zusammensetzt, bei jeder Ausführung eines Programms eine andere ist. [Moseley und Marks (2006)] Deswegen

ist auch insb. das Testen von Programmen, die in diesem Paradigma geschrieben sind, sehr schwierig. [Moseley und Marks (2006)]

Die Notwendigkeit einer Synchronisierung

In manchen Fällen führt der parallele Zugriff von mehreren Threads auf Daten zu Fehlern. So wird in Java bspw. beim Ändern eines Datensatzes einer Collection ein Laufzeitfehler geworfen, so auf diese ein paralleler Zugriff erfolgt.

Des Weiteren besteht die Problematik der sog. *Race Conditions*. Sie liegt vor, wenn sich die Semantik des Programms entsprechend der Sequentialisierungsabfolge ändert. Dies ist bspw. der Fall, wenn zwei Threads eine Variable setzen möchten und dafür den aktuellen Wert jener Variablen verändern. Falls die beiden Vorgänge parallel ablaufen, sind Szenarien denkbar, in denen die letztendlichen Werte unterschiedlich sind und davon abhängen, in welcher Reihenfolge die Lese- und Schreibvorgänge stattfanden.

Die beschriebenen Problematiken führen dazu, dass der Bedarf für Sektionen im Code besteht, die nur von einem Thread ausgeführt werden dürfen (sog. *kritische Abschnitte*). Sie dienen dazu, dass innerhalb dieser Codeabschnitte ein Determinismus der Sequentialisierungsabfolge hergestellt und damit die Semantik des Programms fixiert wird. Diese erzwungenen Sequentialisierungen werden durch Synchronisierungsprimitive wie Semaphoren, Mutexe und Monitore erreicht (im Folgenden *Locks* genannt).

Schwierigkeiten bei der Synchronisierung

Das Konzept der kritischen Abschnitte bringt jedoch neue Herausforderungen mit sich: wie groß dimensioniert muss ein Abschnitt sein? Ist er zu groß, warten möglicherweise mehr Threads als notwendig, so dass das Programm an Performanz einbüßt. Ist er zu klein, könnte die Semantik leiden.

Außerdem können durch das Locking sog. *Deadlocks* auftreten, welche dadurch charakterisiert sind, dass eine zyklische Wartebeziehung innerhalb einer Gruppe von Threads besteht. Die Threads warten auf das Freiwerden von Locks, die von anderen Threads aus der Gruppe gehalten werden. Diese werden jedoch nicht freigegeben, da die Threads die den Lock halten selbst blockiert sind und wiederum darauf warten, dass andere Threads in der Gruppe einen Lock freigeben. Das Ergebnis ist ein Stillstand der betroffenen Threads, was einen negativen Einfluss auf das Programm als solches haben kann.

Um eine Deadlock-Problematik zu vermeiden, muss eine der vier notwendigen Bedingungen für deren Auftreten beseitigt werden. So können die Locks bspw. von jedem Thread in einer

definierten Reihenfolge angefordert werden, so dass die Bedingung des zirkulären Wartens aufgelöst wird. [Tucker (2004)] Die Sicherstellung der richtigen Reihenfolge ist in den meisten Programmiersprachen jedoch die alleinige Aufgabe des Programmierers und wird nicht durch Tools unterstützt – was die Fehleranfälligkeit erhöht. [Lee (2006)]

2.1.2. Software Transactional Memory

STM ist ein Konzept, das Transaktionen für den Zugriff auf gemeinsamen Speicher anbietet, welche die atomaren Lese- und Schreiboperationen kapseln und nicht-blockierend ausgeführt werden. Es ist Grundlage für Implementierungen, die auf Standard-Computerhardware basieren und daran keine besonderen Anforderungen stellen.

Die Funktionsweise stellt sich wie folgt dar [nach Shavit und Touitou (1997)]: die Unteraktionen der Transaktionen fordern, wie bei der Shared State Concurrency, einen Lock für jeden Speicherbereich an, auf den sie zugreifen möchten und können diesen anschließend verändern. Falls allen Unteraktionen das Anfordern der Locks gelungen ist, wurde die Transaktion erfolgreich abgeschlossen. Ansonsten wird sie abgebrochen und später erneut ausgeführt. Dies geschieht so lange, bis sie erfolgreich abgeschlossen wurde.

STM bietet den Vorteil, dass der Programmierer die Synchronisierung des Programmablaufs nicht explizit bedenken muss und Deadlocks durch die feste Reihenfolge, in der die Locks intern angefordert werden, vermieden werden.

Gegen STM spricht vor allem, dass die Performance schlechter ist als jene, die etwa mit der Shared State Concurrency erreicht werden kann.

2.1.3. Message Passing Concurrency

Process Calculi

Process Calculi sind eine Familie von Ansätzen, die es ermöglichen, Nebenläufigkeit von Systemen formal zu beschreiben und auf algebraische Weise zu analysieren. [Baeten (2005)]

Es ist möglich, bestimmte Eigenschaften von Programmen die nach einem solchen Modell geschrieben wurden zu beweisen – bspw. dass diese frei von Deadlocks sind. [Buth u. a. (1997)]

Obwohl es eine Reihe von Programmiersprachen gibt, die an Process Calculi angelehnt sind, hat keine davon eine signifikante praktische Relevanz. [Wing (2002)] Deswegen werden die Semantiken dieser Sprachfamilie im Rahmen der vorliegenden Arbeit nicht weiter verfolgt.

Actor Model

Das Aktoren-Modell wurde 1973 von Carl Hewitt, Peter Bishop und Richard Steiger in der Abhandlung *A Universal Modular Actor Formalism for Artificial Intelligence* erstmals vorgestellt und 1977 durch Hewitt und Henry Baker um die fundamentalen Prinzipien ergänzt[1].

Es beschreibt Nebenläufigkeit nicht durch einen Baum aller möglichen Sequentialisierungen des Ablaufes von parallelen Prozessen, die auf einem globalen Zustand operieren, sondern auf der partiellen Ordnung von *Events*, wobei ein Event das Eintreffen einer Nachricht bei einer Instanz ist, welche diese verarbeitet. [Baker und Hewitt (1977)]

Die Motivation hinter der Formulierung des Modells war die zunehmende Verbreitung von vernetzten Systemen und der damit aufkommende Bedarf nach distributierten Anwendungen. Die Autoren waren der Ansicht, dass die bisherigen Programmiermodelle nicht optimal für diese Architektur geeignet gewesen seien, insb. da das Konzept des Shared Memory sich oftmals als der größte Flaschenhals in einem System herausgestellt habe. [Baker und Hewitt (1977)]

Nebenläufigkeit durch Message Passing

In Aktorensystemen tauschen Prozesse Daten nicht direkt über einen Shared Memory aus, sondern die Nebenläufigkeit wird durch das Senden von Nachrichten und die Verarbeitung dieser durch sog. *Aktoren* erzeugt. [Baker und Hewitt (1977)]

Aktoren konsultieren bei Eintreffen einer neuen Nachricht ihr *Script* (Programmcode) und verarbeiten sie isoliert vom restlichen Programm. Das bedeutet, dass sie nur ihren lokalen Zustand benutzen und keine Daten verwenden sollen, auf die andere Aktoren Zugriff haben. Während der Bearbeitung der Nachricht, kann ein Aktor seinen Zustand ändern, Nachrichten an andere Aktoren schicken und neue Aktoren erzeugen. Dies kann in einer beliebigen Reihenfolge und auch parallel geschehen. [Baker und Hewitt (1977)]

Nachrichten sind die einzige Möglichkeit für einen Aktor mit anderen Aktoren zu kommunizieren und seinen lokalen Zustand mitzuteilen, da außerhalb des Aktors kein Zugriff auf diesen möglich ist. Da der Sender und die Nachricht nicht miteinander gekoppelt sind, kann der Aktor nach dem Absenden sofort mit seinem Script fortfahren. [Hewitt (1977)]

Aufgrund der physikalischen Grundlage und angesichts der Nebenläufigkeit, ist die Reihenfolge in der Nachrichten bei einem Aktor ankommen nicht vorhersagbar, jedoch wird nur eine Nachricht auf einmal zugestellt und die Abarbeitung der Nachrichten erfolgt sequentiell. [Baker und Hewitt (1977)]

[1] In der Abhandlung *Laws for Communicating Parallel Processes*

Referenzen auf Aktoren

Das Senden von Nachrichten erfolgt an Referenzen, die eine Verbindung zu einem Aktor repräsentieren. Eine weitere Aufgabe der Referenzen ist, dass festgestellt werden kann, ob mehrere auf denselben Aktor verweisen. [Baker und Hewitt (1977)]

Ein Aktor kann seine Referenzen zu anderen Aktoren aus drei Quellen beziehen: er kennt jene, die er bei der Erzeugung mitbekommen hat, jene von den Aktoren, die er selbst angelegt hat, und jene, die ihm in Nachrichten zugeschickt wurden. [Baker und Hewitt (1977)]

3. Das Akka-Framework

Das Akka-Framework ist eine Implementierung eines Aktorensystems, das Bytecode für die Java Virtual Machine erzeugt und in Bibliotheken für die Programmiersprachen Java und Scala verfügbar ist.

Hinweis: die in diesem Kapitel ab Abschnitt 3.2 dargestellten Fakten wurden bereits in mehreren Werken publiziert. Darunter bsw. Typesafe Inc. (2013) und Wyatt (2013).

3.1. Unterschiede zum ursprünglichen Actor Model

Da Akka die JVM als Laufzeitumgebung benutzt sowie ein Framework darstellt und keine eigene Sprache ist, werden einige der Regeln des Aktorenmodells, wie sie von Hewitt et al. definiert wurden, nicht strikt eingehalten. So ist es bspw. möglich, in einem Aktor auf Daten zuzugreifen, die außerhalb seines Kontextes liegen und es können Daten auch prinzipiell beliebig nach außen referenziert werden, so dass ein direkter Zugriff möglich ist und nicht zwingend der Umweg über eine Nachricht gegangen werden muss. Es liegt am Benutzer, diese Leitlinien einzuhalten – was vom Akka-Team auch ausdrücklich empfohlen wird.

Des Weiteren relativiert Akka den Gedanken der nichtdeterministischen Reihenfolge der Nachrichtenzustellung und gibt eine Garantie, dass die Reihenfolge der Nachrichten die vom selben Thread oder Aktor gesendet wurden erhalten bleibt.

Ein Aktor hat in Akka auch mehr Möglichkeiten, Referenzen auf andere Aktoren zu erhalten. So kann er diese auch über URLs vom Aktorensystem abrufen, er kann Referenzen auf Gruppen von Aktoren anfordern und auf die Eltern und Nachkommen von Aktoren, die er bereits kennt, zurückgreifen.

3.2. Der Aktor

Der Aktor ist in Akka das grundlegende Element aus dem eine Applikation zusammengesetzt ist. Er ist implementiert als Java-Objekt, das von dem Aktorensystem angelegt und verwaltet wird.

Im Wesentlichen erhält er Nachrichten, wird durch diese aktiv und führt daraufhin Code aus.

3.2.1. Definition

Um in Akka einen Aktor zu erzeugen, muss eine neue Klasse definiert werden, die den Trait `akka.actor.Actor` erweitert. Im `Actor`-Trait gibt es nur einen abstrakten Member, der bei der Definition der Klasse konkretisiert werden muss. Es handelt sich um die Methode `receive`. Diese Methode muss eine Funktion vom Typ `PartialFunction[Any, Unit]` zurückgeben (im Folgenden *`receive`-Funktion* genannt), der später die Nachrichten, die an den Aktor geschickt wurden, übergeben werden und die somit die eigentliche Business Logic des Aktors darstellt.

Der Typparameter `Unit` der Funktion macht bereits deutlich, dass deren Ablauf und damit die Behandlung von Nachrichten auf Seiteneffekten basiert und keinen verwertbaren Rückgabewert liefert.

Ein einfacher Aktor könnte z.B. die Nachricht `("Gib diesen Text aus", "Das Wetter ist schön!")` verstehen und sie dahingehend interpretieren, dass der erste Teil des `Tuple2` ihn anweist, den zweiten Teil auf der Konsole auszugeben.

Eine weitere Nachricht könnte ihn anweisen, zwei Ziffern zu addieren und das Ergebnis auszugeben: `("Addiere diese Zahlen", 1, 2)`.

Eine komplette Definition eines Aktors, der die beiden Nachrichten versteht, könnte so aussehen:

```
1  import akka.actor._
2
3  class PrintOutActor extends Actor {
4    def receive = {
5      case ("Gib␣diesen␣Text␣aus", text)
6      => println(text)
7      case ("Addiere␣diese␣Zahlen", zahl1: Integer, zahl2: Integer)
8      => println(zahl1 + zahl2)
9    }
10 }
```

In der Regel besteht die `receive`-Funktion aus einem *Pattern Matching* auf die Nachricht. Der Pattern Match stellt dabei eine Zuordnung des Nachrichtenobjektes zu einem Laufzeittyp her und extrahiert seine Bestandteile, wobei meist mehrere Zuordnungsmöglichkeiten definiert werden und jedem ein Codeabschnitt zugeordnet ist.

In diesem Beispiel sind zwei Fälle definiert, auf die hin eingehende Nachrichtenobjekte überprüft werden. Der erste Fall trifft zu, wenn die Nachricht vom Typ `Tuple2` und das erste Datum, das in diesem enthalten ist, ein `String` mit dem definierten Inhalt ist. Das zweite in dem `Tuple2` enthaltene Datum wird in diesem Fall an die Variable `text` gebunden und im Codeabschnitt an die `println`-Methode übergeben.

Innerhalb der `receive`-Funktion, und falls ein Pattern Matching verwendet wird auch in den zu den Fällen gehörenden Codeabschnitten, kann beliebiger Code ausgeführt und bspw. auch Hilfsaktoren angelegt und Nachrichten verschickt werden.

Nicht behandelte Nachrichten

Nachrichten, die nicht in der `receive`-Funktion behandelt werden, auf die also kein Fall des Pattern Matches zutrifft, werden an die `unhandled`-Methode übergeben. Die Standardimplementierung der Methode bewirkt, dass die Nichtverarbeitung der Nachricht in dem `EventStream` des Aktorensystems geloggt und die Nachricht danach verworfen wird. Für den Fall, dass es sich um eine spezielle Nachricht vom Typ `akka.actor.Terminated` handelt, wirft sie eine `akka.actor.DeathPactException`.[1]

Diese Methode kann mit einer eigenen Implementierung überschrieben werden.

Kontext von Aktoren

Aktoren besitzen innerhalb des Aktorensystems einen Kontext, auf den sie zugreifen und den sie konfigurieren können. Dieser beinhaltet verschiedene Attribute des Aktors und Zugriffe auf zu ihm in Relation stehende Aktoren und das Aktorensystem, in dem sich der Aktor befindet.

Der Kontext ist in Aktoren über den Wert `context` verfügbar und vom Typ `akka.actor.ActorContext`.

Ändern der `receive`-Funktion

Der Message Handler eines Aktors kann während dieser arbeitet geändert werden. Hierzu bietet der `ActorContext` die `become`- und `unbecome`-Methoden an. Die `receive`-Funktionen sind in Akka in einem Stack organisiert, auf den man beliebig Message Handler ablegen und herunternehmen kann. Die jeweils oberste `receive`-Funktion wird als aktueller Message Handler verwendet.

[1] `Terminated`-Nachrichten erfolgen in der Regel nur, falls ein Aktor eine sog. *DeathWatch* auf einen anderen Aktor hält und dieser terminiert wird. Die DeathWatch-Funktionalität ermöglicht es Aktoren, auf diese Weise über die Beendigung von anderen Aktoren informiert zu werden.

Die Methode `become` des Kontextes eines Aktors legt die als Argument angegebene `receive`-Funktion auf den Stack; die Methode `unbecome` führt auf den Stack ein Pop durch. Zu beachten gilt es dass wenn bei einem `become` lediglich die Funktion als Argument übergeben wird, vor dem Push ein Pop durchgeführt wird, so dass vorher der alte Message Handler verworfen wird – das Pop lässt sich verhindern, indem der `become`-Methode zusätzlich zu der Funktion für den Parameter `discardOld` ein `false` übergeben wird.

Änderungen am `receive`-Stack werden aktiv, wenn die nächste Nachricht verarbeitet wird.

Timeout der `receive`-Funktion

Es existiert keine zeitliche Begrenzung zwischen dem Erhalt von Nachrichten. Man kann jedoch einen Timeout festlegen, der den betreffenden Aktor darüber benachrichtigt, dass die festgelegte Zeitspanne verstrichen ist, ohne dass eine Nachricht einging.

Der Timeout kann über die Methode `setReceiveTimeout` des `ActorContext` gesetzt werden. Die Methode erwartet ein Argument vom Typ `scala.concurrent.duration.Duration`, das die Zeitspanne definiert.

Wenn der Timeout erreicht wurde, wird dem Aktor eine Nachricht vom Typ `akka.actor.ReceiveTimeout` zugestellt.

Der Timeout kann deaktiviert werden, indem der `setReceiveTimeout`-Methode der Wert `Duration.Undefined` übergeben wird.

Es folgt die Definition eines Aktors, der einen `receive`-Timeout von 100 Millisekunden festlegt und sich nach Erhalt des `ReceiveTimeout` selbst terminiert:

```
1  import akka.actor._
2  import scala.concurrent.duration._
3
4  class TimeoutingActor extends Actor {
5    context.setReceiveTimeout(100 millis)
6
7    def receive = {
8      case ReceiveTimeout
9      => context.stop(self)
10   }
11 }
```

3.2.2. Erzeugung und Start

Wenn man versuchen würde, den obigen Aktor außerhalb des Akka-Systems mittels `new` zu erzeugen, würde Akka eine `akka.actor.ActorInitializationException` werfen. Die Anlage eines Aktors ergibt nur Sinn, so diese innerhalb des Aktorensystems geschieht, da der Aktor die dadurch bereitgestellten Ressourcen benötigt, um aktiv werden zu können.

In Akka werden Aktoren ausschließlich über die Methode `actorOf` erzeugt. Diese Methode erwartet mindestens ein Objekt vom Typ `akka.actor.Props` als Argument. Das `Props`-Objekt enthält Konfigurationsparameter, die Akka für die Instanziierung benötigt. Über das Objekt lässt sich bspw. angeben, welchen Dispatcher der Aktor verwenden soll und ob es sich um einen Router handelt.

Die minimalistischste Konfiguration des `Props`-Objektes ist die Angabe, um welchen Aktorentyp es sich handelt. Sie erfolgt als Typparameter der Aktorenklasse und kann durch einen Namen ergänzt werden, den der Aktor tragen soll[2]. Falls kein Name angegeben wird, erzeugt Akka einen synthetischen.

Sollen dem Konstruktor einer `Actor`-Klasse Argumente übergeben werden, muss das `Props`-Objekt auf eine andere Weise erzeugt werden: dem Companion Object der `Props`-Klasse muss in diesem Fall ein By-Name-Parameter mit einer Funktion übergeben werden, welche die `Actor`-Instanz mittels `new` direkt erzeugt und in diesem Zuge die Argumente übergibt.

Konkret würde die Anlage des `PrintOutActor`s wie folgt aussehen:

```
1 val system: ActorSystem = ActorSystem("System")
2 val actor: ActorRef =
3   system.actorOf(Props[PrintOutActor], "PrintOutActor")
```

Der Wert `system` stellt die Instanz eines Akka-Aktorensystems dar, in dem der zu erzeugende Aktor angelegt wird. Auf ihr erfolgt ein Aufruf der `actorOf`-Methode mit einer Instanz der `Props`-Klasse samt Typparameter mit der Klasse des Beispielaktors und dem Namen, den der Aktor tragen wird. Die Rückgabe der `actorOf`-Methode ist ein Objekt vom Typ `akka.actor.ActorRef` und dient als Referenz auf den Aktor.

Die zweite Zeile des Listings bewirkt außerdem, dass der Aktor gestartet wird, so dass er nach der Initialisierung direkt Nachrichten empfangen und aktiv werden kann.

[2] Der Name eines Aktors ist u.a. auch Teil seiner URL. Falls der angegebene Name bereits vorhanden ist, wirft Akka eine `akka.actor.InvalidActorNameException`, da dieser eindeutig sein muss.

ActorRefs

`ActorRef`s werden bei der Anlage von Aktoren durch die Methode `actorOf` oder durch die Umwandlung von URIs durch die Methode `actorFor` erzeugt. Sie werden primär dafür verwendet, um Nachrichten an den Aktor zu schicken, den sie repräsentieren. Sie können jedoch auch verglichen werden, um herauszufinden, ob mehrere Instanzen auf denselben Aktor verweisen und aus ihnen lässt sich über die Methode `path` die URI des Aktors extrahieren. Des Weiteren bieten sie über die Methode `isTerminated` die Möglichkeit herauszufinden, ob der referenzierte Aktor beendet wurde oder noch aktiv ist.

Eine `ActorRef` ist auf Instanz-Basis für Aktoren gültig, d.h., dass die Referenz nach der Terminierung eines Aktors ungültig wird und Nachrichten, die danach gesendet wurden, an den Dead Letters-Aktor zugestellt werden. Das ist selbst dann der Fall, wenn ein Aktor gestoppt und durch einen anderen mit derselben URI ersetzt wird – auch dann muss die neue `ActorRef` verwendet, bzw. eine neue angefordert werden.

`ActorRef`s sind immutable und lassen sich an entfernte Aktorensysteme schicken.

3.2.3. Senden von Nachrichten

Nachrichten sind in Akka vom Typ `Any`,[3] jedoch fügt das System implizit jeder Nachricht deren Sender hinzu, der in einem Aktor über die Methode `sender` zugänglich ist. Die Methode liefert die `ActorRef` des Senders der Nachricht, die zu diesem Zeitpunkt in der `receive`-Funktion behandelt wird.[4]

Das Senden von Nachrichten erfolgt in Akka asynchron, so dass der sendende Thread, bzw. Aktor, sofort nachdem die Nachricht abgesetzt wurde weiterarbeitet. Ist eine Nachricht abgesetzt, liegt es in der Verantwortung des Akka-Systems, diese zuzustellen – der Sender ist in dem Prozess nicht weiter involviert.

Akka gibt hierbei keine Garantie, dass Nachrichten, die über ein Netzwerk an andere Aktorensysteme geschickt wurden, auch in der Tat ankommen.[5] Es kommt jedoch zu keiner

[3] In Akka gibt es auch die Möglichkeit *Typed Actors* zu erzeugen, die ein wohl definiertes Interface haben. Man schickt einem solchen Aktor im Prinzip aber keine Nachrichten, sondern ruft die im Interface definierten Methoden auf. Deswegen ähnelt die Semantik der von RPC Implementierungen. Bei der Benutzung dieses Modells gibt es starke Einschränkungen was die Anzahl der Möglichkeiten betrifft, weswegen darauf nicht näher eingegangen wird.

[4] Falls die Nachricht von einen Thread außerhalb eines Aktors gesendet wurde, so liefert der Aufruf eine spezielle Aktoreninstanz zurück, an die diejenigen Nachrichten zugestellt werden, für die kein Empfänger existiert – es handelt sich um den *Dead Letters*-Aktor.

[5] Es existiert ein Design Pattern, das eine garantierte Zustellung von Nachrichten als Erweiterung des Programms nutzbar macht. Es beschreibt die Schaltung von zwei Aktoren zwischen Sender und Empfänger, die eine zuverlässige Zustellung sicherstellen. Das Pattern nennt sich *Reliable Proxy*.

Duplikation von Nachrichten, so dass der Programmierer davon befreit ist, diese herausfiltern zu müssen.

Akka garantiert, dass die Reihenfolge von Nachrichten, die vom selben Thread oder Aktor gesendet wurden, erhalten bleibt (wobei dazwischen auch Nachrichten von anderen Aktoren ankommen können).

Die folgende Zeile bewirkt das Senden einer Nachricht an den `PrintOutActor` (im Folgenden wird die `!`-Methode auch als `Tell` bezeichnet):

```
1  actor ! ("Addiere␣diese␣Zahlen", 1, 2)
```

Lässt man den in den Beispielen angegebenen Code laufen, erfolgt die Ausgabe ”3” auf der Konsole.

Weiterleiten von Nachrichten

Wenn ein Aktor eine Nachricht zu einem anderen Aktor weiterleiten möchte, kann es wünschenswert sein, dass der ursprüngliche Sender der Nachricht auch in der weitergeleiteten Nachricht eingesetzt wird. Würde die Nachricht per Tell weitergeleitet, wäre dies nicht der Fall, da der weiterleitende Aktor als Sender erscheinen würde.

Die Erhaltung des ursprünglichen Senders kann durch die Verwendung der Methode `forward` anstelle von `!` erreicht werden. Es ist außerdem möglich, der `!`-Methode explizit die `ActorRef`, die als Sender der Nachricht verwendet werden soll, zu übergeben. Auf diese Weise lässt sich jeder beliebige Aktor als Sender einsetzen.

Einschränkungen

Nachrichten, die an andere Aktorensysteme gesendet werden, müssen sich serialisieren lassen. Dies kann dadurch erreicht werden, dass das Interface `java.io.Serializable` von der als Nachrichtenobjekt zu verwendenden Klasse implementiert wird oder indem man den Trait `akka.serialization.Serializer` auf die zu serialisierende Klasse anpasst und in der Konfiguration als den für die Nachricht zu verwendenden Serialisierer angibt.

3.2.4. Aktoren beenden

Ein Aktor lässt sich auf verschiedene Weisen terminieren. Die gewöhnliche Vorgehensweise ist ein Aufruf der `stop`-Methode des `ActorContext`s oder `ActorSystem`s, der die entsprechende `ActorRef` des zu terminierenden Aktors übergeben wird. Dieser Vorgang geschieht asynchron, so dass sich nicht vorhersagen lässt, zum welchem Zeitpunkt genau der Aktor tatsächlich

terminiert wird, nachdem die Methode ausgeführt wurde.[6] Die Terminierung des Aktors erfolgt, nachdem dieser die aktuelle Nachricht verarbeitet hat. Alle Nachrichten, die sich zu dem Zeitpunkt noch in der Warteschlange befunden haben, werden normalerweise[7] an den Dead Letters-Aktor weitergeleitet.

Eine weitere Möglichkeit ist, dem Aktor eine Nachricht vom Typ `akka.actor.PoisonPill` zu schicken. Der Aktor wird gestoppt, sobald er die Nachricht verarbeitet.

Um den Aktor mit einer Exception zu beenden, bzw. dies den Supervisor entscheiden zu lassen, kann einem Aktor eine Nachricht vom Typ `akka.actor.Kill` gesendet werden.

Eine Möglichkeit, den Vorgang zu koordinieren, ohne eine DeathWatch auf den zu terminierenden Aktor zu halten, bietet die Methode `gracefulStop` des `akka.pattern` Package Objects. Sie veranlasst die Terminierung eines Aktors und bietet die Möglichkeit an, den Thread so lange zu blockieren, bis die `postStop`-Methode desselben verlassen wurde (wobei dieselbe Problematik hinsichtlich der Wiederverwendbarkeit des Namens besteht wie bei einer Ausführung der `stop`-Methode).

3.2.5. Hierarchische Struktur

Die Aktoren, aus denen ein Programm in Akka besteht, sind in einer Baumstruktur organisiert. Das bedeutet konkret, dass jeder Aktor einen Vateraktor haben *muss* und Nachkommen haben kann. Die Wurzel des Baumes stellt der sog. *User Guardian Actor* dar, dessen Nachkommen die vom User angelegten Aktoren sind.

URIs

Aufgrund der hierarchischen Struktur und der Tatsache, dass Aktoren eindeutige Namen haben müssen, lässt sich jedem Aktor eine URI zuordnen. Über diese kann eine den Aktor referenzierende `ActorRef` vom Aktorensystem angefordert werden, über die sich Nachrichten senden lassen.

Das erste Element der Aktoren-URIs ist der Name des Aktorensystems, gefolgt von dem User Guardian Actor und dem Pfad innerhalb der selbst angelegten Aktorenstruktur.[8]

Im Falle des `PrintOutActor`s aus den Beispielen lautet die vollständige URL wie folgt:

[6] Dies hat insb. Auswirkungen auf eine Wiederverwendung des Aktorennamens. Da diese eindeutig sein müssen und der alte Name möglicherweise noch belegt ist, kann eine `InvalidActorNameException` geworfen werden. Dies lässt sich verhindern, indem eine DeathWatch auf den zu terminierenden Aktor gelegt wird und der Name erst dann wiederverwendet wird, nachdem die `Terminated`-Nachricht angekommen ist.

[7] Dies hängt von der Art der verwendeten Mailbox ab.

[8] Wenn auf Aktoren zugegriffen wird, die sich auf anderen Computern befinden als auf demjenigen, von dem die `ActorRef` angefordert wird, muss außerdem ein Hostname und Port angegeben werden.

`akka://System[@{host}:{port}]/user/PrintOutActor`

Die `ActorRef` des Aktors kann mittels der `actorFor`-Methode des `ActorSystem`s, bzw. des `context`-Wertes innerhalb von Aktoren, generiert werden:

```
1 val actor: ActorRef =
2   system.actorFor("akka://System/user/PrintOutActor")
```

Falls der Aufruf von einem Aktor aus erfolgt, können auch relative Pfade verwendet werden. Hierbei ist zu beachten, dass dann die `actorFor`-Methode des `context`-Wertes, der innerhalb von Aktoren verfügbar ist, benutzt werden muss.

Über die `actorSelection`-Methode lassen sich Referenzen auf Gruppen von Aktoren generieren. Die Verwendung ist identisch mit jener der `actorFor`-Methode, es lassen sich jedoch auch Wildcards verwenden, über die ein Teilbaum als Ziel der Referenz definiert werden kann. Mit dem String `akka://System/user/PrintOutActor/*` lässt sich bspw. eine `ActorSelection` erzeugen, über die allen Nachkommen des `PrintOutActor`s Nachrichten gesendet werden können.

Zu beachten gilt es, dass wenn die `actorFor`-Methode mit einer lokalen URI aufgerufen wird, d.h. einer eines zum selben `ActorSystem` gehörigen Aktors, muss dieser bereits angelegt worden sein, da die erzeugte `ActorRef` sonst ins Leere zeigt. Wird stattdessen eine URL von einem entfernten Aktor oder die `actorSelection`-Methode verwendet, wird der Pfad nicht direkt verifiziert, sondern erst bei dem Senden von Nachrichten.

3.2.6. Supervision

In Akka hat jeder Aktor einen Elternaktor und dieser ist gleichzeitig auch sein Supervisor. Die Supervision ist eine Komponente eines jeden Aktors, die sich um die Fehlerbehandlung von Exceptions kümmert, die von den Nachkommen geworfen werden. Sie wird immer dann aktiv, wenn eine Exception in einem Aktor nicht innerhalb der `receive`-Funktion abgefangen wird. In so einem Fall tut dies Akka selbst und schickt dem Supervisor des fehlerbehafteten Aktors eine Nachricht, damit dieser auf das Ereignis reagieren und Maßnahmen treffen kann, die einen konsistenten Zustand des Programms wiederherstellen.

Für die Fehlerbehandlung stehen eine Reihe von Möglichkeiten zur Verfügung: der betreffende Aktor kann entweder terminiert oder neu instantiiert, der Fehler ignoriert oder zum nächsten Supervisor hochgereicht werden. Außerdem kann festgelegt werden, ob diese Entscheidung nur auf den fehlerbehafteten Aktor, oder auf alle Nachkommen des Supervisors angewendet werden soll.

Ähnlich der `receive`-Funktion, ist für diesen Entscheidungsvorgang eine Funktion vom Typ `PartialFunction[Throwable, Directive]` zuständig, die ein Pattern Matching auf die

Instanz des geworfenen Fehlers macht und ihr in Form von Instanzen vom Trait `Directive` eine der genannten Entscheidungen zuordnet. Die verfügbaren Instanzen heißen `Stop`, `Restart`, `Resume` und `Escalate`.

Die Standardimplementierung der Supervision kann vom User geändert werden, indem die Methode `supervisorStrategy` innerhalb des Aktors so überschrieben wird, dass sie eine andere Entscheidungsfunktion zurückgibt.

Das Standardverhalten bewirkt, dass der entsprechende Aktor terminiert wird, falls die geworfene Exception vom Typ `ActorInitializationException` oder `ActorKilledException` ist – d.h. ein Fehler bei der Anlage des Unteraktors auftrat oder dieser eine Nachricht vom Typ `akka.actor.Kill` empfangen hat; beim Typ `Exception` erfolgt eine Neuanlage des Aktors und alle übrigen Fehler werden an den Supervisor hochgereicht.

Direktive `Stop`

Die `Stop`-Direktive bewirkt, dass Akka den, bzw. die, betreffenden Aktoren terminiert. Da jeder Aktor einen Elternaktor haben muss, erfolgt die Terminierung rekursiv – es werden also neben dem fehlerbehafteten Aktor auch alle seine Nachkommen beendet.

Wenn Aktoren terminiert wurden, so sind alle `ActorRef`s die davor auf sie gezeigt haben, ungültig und Nachrichten, die trotzdem gesendet werden, werden an den Dead Letters-Aktor zugestellt. Dieser loggt den Erhalt der Nachricht im EventLog des Aktorensystems.

Bei der Terminierung von Aktoren wird ferner die DeathWatch-Funktionalität ausgelöst, d.h., alle Aktoren, die zuvor beim Aktorensystem Interesse an diesem Ereignis angemeldet hatten, werden durch eine Nachricht vom Typ `Terminated` über die Beendigung des Aktors informiert.

Direktive `Resume`

Bei dieser Direktive wird der geworfene Fehler ignoriert und der entsprechende Aktor fährt mit der nächsten Nachricht fort.

Direktive `Escalate`

Wenn `Escalate` als Entscheidung definiert wurde, reicht der Supervisor des Aktors, der den Fehler geworfen hat, die Exception an seinen Supervisor hoch. Der Supervisor behandelt den Fehler dann so, als hätte ihn sein Unteraktor selbst produziert und wendet seinerseits eine Direktive an.

Direktive `Restart`

Bei einem `Restart` wird eine neue Instanz des fehlerwerfenden Aktors angelegt. Dies hat den Vorteil, dass der Aktor seinen Zustand verliert – was wünschenswert sein kann, falls dieser durch den Fehler korrumpiert wurde.

Obwohl eine neue Instanz geschaffen wird, ist dieser Vorgang für das restliche Programm transparent. Die DeathWatch wird nicht getriggert und alle `ActorRef`s, die für den betroffenen Aktor bestehen, verlieren nicht ihre Gültigkeit. Auch bleibt bei einem Neustart die Mailbox des Aktors erhalten, so dass dieser nach dem Vorgang mit der Bearbeitung der nächsten Nachricht weitermacht.

Das Standardverhalten dieser Direktive sieht vor, dass der fehlerbehaftete Aktor neu gestartet wird, seine Nachkommen jedoch terminiert werden – so, als wenn auf diese die `Stop`-Direktive angewendet würde. Es liegt am Konstruktor des neu gestarteten Aktors und seiner Nachkommen, die gewünschte Aktorenstruktur neu zu erzeugen. Dieses Verhalten kann geändert werden, indem die entsprechenden Methoden, die bei Änderungen des Lebenszyklus eines Aktors aufgerufen werden, überschrieben werden.

Lebenszyklus von Aktoren

Aufgrund der unterschiedlichen Direktiven, die ein Supervisor auf Unteraktoren anwenden kann, die Fehler geworfen haben, unterliegen Aktoren einer Art Lebenszyklus. Sie wechseln bei der ersten Initialisierung von der Nicht-Existenz in die Existenz, kehren zu einem bestimmten Zeitpunkt in den ersten Zustand zurück und können dazwischen neu initialisiert werden (siehe Abbildung 3.1).

Um steuernd in diese Vorgänge eingreifen zu können, existieren in dem `Actor`-Trait vier Methoden, die während dieser aufgerufen werden und vom User überschrieben werden können.

Nachdem die `actorOf`-Methode aufgerufen wurde, sorgt Akka für die erste Instanziierung des Aktorenobjektes. Falls ein `Props`-Objekt mit einem By-Name-Argument verwendet wurde, führt Akka diese Funktion bei der Anlage aus, ansonsten verwendet es den Default-Konstruktor der Klasse.

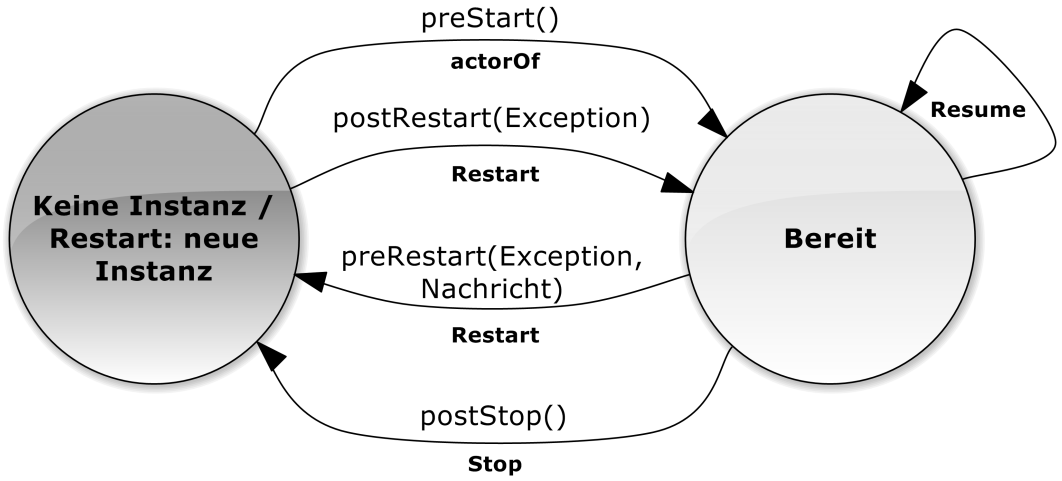

Abbildung 3.1.: Lebenszyklus von Aktoren

`preStart`

Nach der Anlage, ruft Akka die `preStart`-Methode des Aktorenobjektes auf. Sie ist standardmäßig leer und kann vom User mit zusätzlichem Initialisierungscode überschrieben werden. Eine Besonderheit dieser Art der Initialisierung ist, dass die Methode ebenfalls nach einem `Restart` des Aktors von dem `postRestart`-Lifecycle Hook aufgerufen wird. Wenn dieser entsprechend überschrieben wird, kann vermieden werden, dass die Initialisierungen in der `preStart`-Methode bei einer Neuinstantiierung des Aktors nach einem Restart ausgeführt werden, was sich in einigen Fällen als wünschenswert erweisen kann.

Der Aufruf der `preStart`-Methode erfolgt synchron, so dass der Aktor erst mit der Bearbeitung von Nachrichten beginnt, sobald diese verlassen wurde. Aktoren können auch während der Initialisierung Nachrichten erhalten – diese werden solange zurückgestellt, bis der Aktor bereit zu deren Verarbeitung ist.

`postStop`

Bevor ein Aktor terminiert wird, erfolgt ein Aufruf der `postStop`-Methode. Die Standardimplementierung ist ebenfalls leer. Hier kann der User bspw. Ressourcen freigeben, die der Aktor verwendet hat. Diese Methode wird in der Standardimplementierung außerdem

vom `preRestart`-Hook aus aufgerufen, der ausgeführt wird, bevor der Aktor im Zuge eines `Restarts` neu angelegt wird.

`pre-` und `postRestart`

Die beiden Methoden `preRestart` und `postRestart` werden im Zuge der Anwendung der `Restart`-Direktive eines Supervisors ausgeführt, und zwar die erstgenannte Methode auf der alten Instanz, die durch eine neue ersetzt werden soll, und die zweitgenannte auf der neu erzeugten Instanz nach dem `Restart`.

Der `preRestart`-Hook wird mit zwei Argumenten aufgerufen: der Instanz der Exception, die der Grund des Neustarts war und, falls verfügbar, der zu dem Zeitpunkt verarbeiteten Nachricht (gekapselt in einem `Option`-Objekt). Die Standardimplementierung terminiert alle Nachkommen und ruft anschließend den `postStop`-Hook auf.

Aufrufe der `postRestart`-Methode erfolgen mit der Instanz der Exception als Argument und rufen normalerweise lediglich den `preStart`-Hook auf.

3.2.7. DeathWatch

Aktoren können sich beim Aktorensystem registrieren, um eine Benachrichtigung zu erhalten, falls andere Aktoren terminiert werden. Sie bekommen in diesem Fall eine Nachricht vom Typ `Terminated` zugestellt, die sie innerhalb der `receive`-Funktion behandeln sollten, da die Standardimplementierung der `unhandled`-Methode ansonsten eine Exception wirft.

Ein Bestandteil der `Terminated`-Nachricht ist die `ActorRef` des terminierten Aktors.

Die Funktion wird aktiviert, indem die Methode `watch` des `ActorContext` mit der `ActorRef` des zu observierenden Aktors als Argument aufgerufen wird.

Beendet werden kann eine DeathWatch, indem, analog zu der `watch`-Methode, die Methode `unwatch` aufgerufen wird.

Die Definition eines Aktors, der eine DeathWatch auf einen anderen Aktor registriert und darauf reagieren kann, lautet bspw. so:

```
1 import akka.actor._
2
3 class DeathWatchingActor extends Actor {
4   val unterAktor: ActorRef = context.actorOf(Props(new Actor {
5     def receive = Actor.emptyBehavior
6   }))
7   context.watch(unterAktor)
8   unterAktor ! Kill
```

```
9
10    def receive = {
11      case Terminated(‘unterAktor‘)
12      => println(s"Unteraktor␣$unterAktor␣terminiert.")
13    }
14  }
```

Wenn man eine Instanz des `DeathWatchingActor`s anlegt, erzeugt dieser einen Unteraktor und registriert eine DeathWatch auf ihn. Der Unteraktor wirft durch die gesendete `Kill`-Nachricht eine `ActorKilledException`, auf die hin sein Supervisor eine Terminierung desselben veranlasst. Die Terminierung löst die DeathWatch des `DeathWatchingActor`s aus und dieser erhält eine `Terminated`-Nachricht. Daraufhin gibt er einen Text auf der Konsole aus, der die Terminierung des Unteraktors kundtut.

3.3. Futures und Dataflow

Futures sind in Akka ein wichtiges Werkzeug, um Concurrency zu erzeugen. In Akka können Funktionen asynchron ausgeführt werden, d.h., dass sie von einen Thread abgesetzt werden und parallel zu diesem, von einem anderen Thread berechnet werden. Ein Future ist ein Handle auf diese asynchrone Berechnung, über das sich das Ergebnis abrufen lässt und sich Berechnungen kombinieren lassen. Wenn die Funktion bspw. einen Rückgabewert vom Typ `Int` haben wird, so lautet der Typ des Futures `Future[Int]`.

Der zukünftige Rückgabewert lässt sich mit Hilfe des Futures jederzeit abfragen bzw. es kann überprüft werden ob dieser bereits vorliegt oder der Thread solange blockiert werden, bis dies der Fall ist. In der Regel wird dies jedoch nicht praktiziert, sondern Aktoren, die Futures verwenden, veranlassen, dass das Ergebnis stattdessen an sie selbst oder einen anderen Aktor weitergeleitet wird, sobald es berechnet wurde. In diesem Fall wird das Future nicht verwendet, um den Rückgabewert der asynchronen Funktion abzufragen, sondern lediglich, um diese anhand ihrer Rückgaben miteinander zu kombinieren.

Aufgrund dieser Eigenschaften eignen sich Futures in den Fällen, in denen parallele Vorgänge auf einen Aktor verzichten können, insb. wenn ein Handle auf das Ergebnis einer Anfrage wünschenswert ist – etwa um die Reihenfolge bei mehreren Anfragen zu erhalten. Futures haben bei einzelnen Berechnungen einen geringeren Overhead als Aktoren.

Voraussetzungen und Verwendung

Da Futures Handle auf asynchrone Berechnungen sind, brauchen diese einen neuen Thread, auf denen sie ablaufen können. In Scala benötigen sie dafür eine Instanz eines `scala.concurrent.ExecutionContext`. In Kombination mit Akka kann dafür eine Dispatcher-Instanz verwendet werden, denn diese ist gleichzeitig ein `ExecutionContext`. Innerhalb von Aktoren kann dafür der verwendete Dispatcher durch einen Import in den Scope gebracht werden. Er wird dadurch aufgrund der Tatsache, dass er als implicit definiert ist, automatisch als zu verwendender `ExecutionContext` eingesetzt. Außerhalb von Aktoren kann bspw. der Default-Dispatcher des Aktorensystems verwendet werden.

Ein Future lässt sich generieren, indem eine asynchron ausgeführte Funktion mit dem `scala.concurrent.Future`-Objekt abgesetzt wird.

Im folgenden Beispiel erzeugt ein Aktor ein Future auf eine mathematische Berechnung und lässt sich das Ergebnis in einer Nachricht zuschicken, sobald es berechnet wurde. Anschließend gibt er es auf der Konsole aus:

```
1  // Import des Futures und der Funktionalität zur Weiterleitung
2  // des Ergebnisses
3  import scala.concurrent.Future
4  import akka.pattern.pipe
5
6  class FutureActor extends Actor {
7      import context.dispatcher
8
9      Future {
10       100 * 5 // wird asynchron berechnet
11     } pipeTo self // Weiterleitung des Ergebnisses
12
13     def receive = {
14       case x: Int => println(x)
15     }
16   }
```

Man beachte, dass die Funktion, die dem `Future`-Objekt übergeben wird, sofort auf einem anderen Thread berechnet wird – in diesem Fall auf einem, aus dem Pool des eingestellten Dispatchers. Wenn der Thread die Berechnungen abgeschlossen hat, ist das Ergebnis über das Future verfügbar und kann extrahiert, oder, wie in diesem Fall, an einen Aktor zugestellt werden.

Den Wert eines Futures auslesen

Wenn man den Wert eines Futures direkt auslesen will, besteht die Problematik, dass unbekannt ist, wann dieser vorliegt. Man kann durch die Methode `isCompleted` zwar in Erfahrung bringen, ob dies bereits der Fall ist, in vielen Situationen ergibt es allerdings nur Sinn, den Thread, der den Wert benötigt, so lange zu blockieren, bis das Ergebnis vorliegt – wie bereits erwähnt, ist das keine Vorgehensweise, die empfohlen wird [Typesafe Inc. (2013)].

Soll es dennoch geschehen, bietet das Objekt `scala.concurrent.Await` zwei Funktionen an, die dies ermöglichen: mit `ready` lässt sich der Thread so lange blockieren, bis die Funktion eine Rückgabe (einen Wert oder eine Exception) erzeugt hat und `result` gibt in diesem Fall zusätzlich das Ergebnis zurück. Zu beachten gilt es, dass die Methode `result` eine Exception wirft, falls die zu dem Future gehörige Funktion eine solche geworfen hat. Beiden Methoden muss ein Timeout übergeben werden, der jedoch unendlich lang sein kann. Wird die Zeitspanne überschritten, werfen die Methoden eine `TimeoutException`.

Futures als Anfragen an Aktoren

In Akka ist eine starke Koppelung von Aktoren an Futures möglich. Es lassen sich Futures an Anfragen an Aktoren zwischenschalten, so dass die Antwort nicht direkt an den anfragenden Aktor geschickt, sondern dazu benutzt wird, den Wert des Futures bereitzustellen.

Die Syntax für eine solche Anfrage ähnelt dabei jener einer normalen Nachrichtenversendung mittels `Tell`, die für gewöhnlich mit dem `!`-Operator geschieht. Soll die Antwort an ein Future weitergeleitet werden, benutzt man die Methode `ask` und schreibt statt des `!` ein `?`. Diese Methode hat ein Future vom Typ `Future[Any]` als Rückgabe, da aufgrund der Tatsache, dass eingehende Nachrichten den Typ `Any` haben, keine statische Ermittlung des Rückgabetyps möglich ist.

Mit Hilfe der Methode `mapTo` lässt sich der Typ des Futures ändern – bei einem Future vom Typ `Future[Any]` gibt `mapTo[Int]` bspw. einen Future vom Typ `Future[Int]` zurück. Falls der Cast jedoch nicht erfolgreich war, der Rückgabewert also einen nicht kompatiblen Laufzeittyp hat, wird das Future einen Wert vom Typ `ClassCastException` enthalten.

Die `ask`-Methode benötigt außerdem eine Instanz des Typs `akka.util.Timeout`, welche die Zeitspanne angibt, nach der das Future mit einem Wert vom Typ `akka.Pattern.AskTimeoutException` erfüllt wird – dieser Parameter ist als implicit definiert.

Verkettung von Futures

Eine wichtige Eigenschaft von Futures ist, dass sie miteinander kombiniert werden können. Man kann bspw. ein Future erzeugen, dass einen Wert errechnet und es mit einem anderen verknüpfen, das den erzeugten Wert manipuliert. Das zweite Future wird erst dann berechnet, wenn die Berechnung des ersten abgeschlossen ist. Anschließend könnte man eine Funktion an das zweite Future hängen, die das manipulierte Ergebnis einem Aktor in einer Nachricht übermittelt.

Die beschriebene Vorgehensweise wird im folgenden Code demonstriert. Das erste Future stellt eine mathematische Berechnung an, das zweite arbeitet mit deren Ergebnis und erzeugt einen neuen Wert. Das Endergebnis wird an einen Aktor weitergeleitet, der es anschließend auf der Konsole ausgibt:

```
Future {
  100 * 5 // Erstes Future
} flatMap { x =>
  Future {
    x * 2 // Zweites Future - "x" ist das Ergebnis des ersten
  }
} pipeTo actor: ActorRef
```

Die Methoden, Futures miteinander zu kombinieren, sind `map` und `flatMap`. Sie hängen eine neue Funktion an die zu einem Future gehörige, haben das Ergebnis dieser als Eingabe und liefern ein neues Future als Rückgabe. Die angehängten Funktionen werden erst dann berechnet, wenn der Wert, den sie als Eingabe haben, vorliegt – es handelt sich um eine Kettenreaktion.

Obwohl eine auf diese Weise an die Berechnung eines Futures angehängte Funktion nur einen Wert als Eingabe hat, lässt sich in ihr auf beliebig viele Futures zugreifen, um so die Ergebnisse von mehr als nur einer Funktion zu verknüpfen. Dies geschieht, indem die Methoden ineinander geschachtelt werden.

Im folgenden Beispiel werden zwei Futures erzeugt und deren Ergebnisse in einem dritten Future kombiniert:

```
val future1 = Future { 100 * 5 }
val future2 = Future { 20 + 80 }

val future3 = future1 flatMap { f1: Int =>
  future2 map { f2: Int =>
    f1 + f2
  }
```

```
}
```

In diesem Fall, wird beim ersten Future die Methode `flatMap` und nicht `map` verwendet, da das dritte Future sonst den Typ `Future[Future[Int]]` und nicht `Future[Int]` hätte.

For-Comprehensions

Da Futures die für eine For-Schleife nötigen Methoden implementieren, lassen sie sich auch innerhalb einer solchen verwenden.

Das obige Beispiel kann auf folgende Weise in einer For-Schleife ausgedrückt werden:

```
val future1 = Future { 100 * 5 }
val future2 = Future { 20 + 80 }

val future3 = for {
  f1 <- future1
  f2 <- future2
} yield f1 + f2
```

Zu beachten gilt es, dass die Verknüpfung von Futures innerhalb einer For-Schleife einer durch eine `map`-Methode erfolgten Kombination entspricht. Die Futures werden zwar asynchron, aber nacheinander berechnet. Im obigen Beispiel geschieht die Berechnung parallel, da die Futures außerhalb der For-Schleife definiert wurden.

Listen in Futures umwandeln

Das Objekt `Future`, über das in den vorherigen Beispielen auch die Futures erzeugt wurden, definiert einige Hilfsmethoden rund um das Zusammenspiel von Collections und Futures. Eine besonders bemerkenswerte dieser Methoden hat eine Collection von Futures als Eingabe und ein Future mit einer Collection als Ausgabe – sie wandelt mehrere Futures in ein einzelnes um. Die Methode heißt `sequence`.

Diese Funktionalität ist insb. dann wünschenswert, wenn eine Aufgabe auf mehrere Aktoren aufgeteilt werden und die Reihenfolge der Zwischenergebnisse wichtig ist und erhalten bleiben soll. Man könnte in diesem Fall eine Liste mit Futures, die die Zwischenergebnisse repräsentieren, anlegen, diese Liste mit Hilfe der `sequence`-Methode in ein einzelnes Future einer Liste von zukünftigen Zwischenergebnissen umwandeln und sich die Ergebnisse mittels der `pipeTo`-Methode zuschicken lassen, sobald das Future alle Zwischenergebnisse aggregiert hat.

Funktionen an Futures anhängen

Es ist möglich, Funktionen an Futures anzuhängen, ohne dass in diesem Zuge ein neues Future erzeugt wird. Innerhalb dieser Callbacks hat man, je nach Variante, Zugriff auf den erfolgreich generierten Wert des Futures, die geworfene Exception oder auf ein Ergebnis beider möglichen Ausgänge. Wenn Futures mit den Methoden `map` oder `flatMap` verknüpft werden, hat man diese Möglichkeit nicht, sondern arbeitet stets mit dem erfolgreich erzeugten Wert als Eingabe. Wenn ein Future eine Exception wirft, propagiert diese in der Kette der verknüpften Futures als Ergebnis und die zu berechnenden Funktionen werden übersprungen.

onSuccess

Dieser Callback wird nur dann ausgeführt, wenn das Ergebnis des Futures vorliegt und die Funktion keine Exception geworfen hat. Es hat eine `PartialFunction` als Parameter, die den Typ des Rückgabewertes als Eingabe hat.

onFailure

Diese Funktion wird nur im Fehlerfall ausgeführt, wenn die zu dem Future gehörige Funktion eine Exception geworfen hat. Sie akzeptiert eine `PartialFunction` mit einem `Throwable` als Eingabe.

onComplete

Dieser Callback wird in jedem Fall ausgeführt, wenn das Future berechnet wurde. Die zugehörige `PartialFunction` akzeptiert eine `scala.util.Try`-Instanz als Eingabe. Falls ein Wert erzeugt wurde, so passiert er die Funktion in einer Instanz der `Success`-Klasse, ansonsten befindet sich die Exception in einer `Failure`-Instanz.

An ein Future können beliebig viele Callbacks angehängt werden – auch mehrere desselben Typs. Die Ausführungsreihenfolge ist nichtdeterministisch.[9]

Exception Handling

Die Methode `recover` hängt eine Funktion an ein Future, die einer geworfenen Exception einen Wert des ursprünglichen Rückgabetyps zuordnet und damit das neu erzeugte Future

[9] Die Reihenfolge lässt sich erzwingen, indem statt der Methode `onComplete` die in Futures definierte Methode `andThen` verwendet wird. Um diese Funktionalität zu erreichen, wird bei jedem Aufruf eine neue Future-Instanz erzeugt, die mit dem selben Wert vervollständigt wird, wie das original Future.

erfüllt. Soll ein anderes Future für die Substituierung verwendet werden und nicht ein expliziter Wert, steht als Alternative die Methode `recoverWith` zur Verfügung.

Im folgenden Code wird ein Future erzeugt, das bei der Berechnung eine Exception wirft. Da es durch die `recover`-Methode um eine Fehlerbehandlungskomponente ergänzt wurde, wird das verknüpfte Future `recoveredFuture` mit dem Wert "100" anstelle der Exception erfüllt.

```
1 val recoveredFuture: Future[Int] = Future { 1 / 0 } recover {
2   case _: ArithmeticException => 100
3 }
```

`fallbackTo`

Eine weitere Möglichkeit, um in Futures geworfene Exceptions zu verarbeiten besteht darin, dass man eine Gruppe von Futures dazu benutzt, den Wert eines darauf basierenden Futures bereitzustellen. Sobald mindestens eines der Futures mit der Berechnung Erfolg hatte, wird das darauf basierende Future mit einem Wert statt einer Exception erfüllt.

Diese Möglichkeit besteht mit der Methode `fallbackTo`. Sie akzeptiert ein Future als Argument und erzeugt ein neues Future, das den erfolgreich generierten Wert eines der beiden Futures annehmen wird. Durch eine Aneinanderreihung von Aufrufen dieser Methode lassen sich beliebig viele Futures miteinander kombinieren – am Ende wird ein Future zurückgegeben, das den Wert eines der Eingabefutures annehmen wird.

Dataflow

Die Dataflow-Komponente in Akka bietet eine alternative Möglichkeit, Futures zu komponieren, die oft kürzeren und leichter verständlichen Code produziert. [Typesafe Inc. (2013)]

Sie funktioniert, indem die Ergebnisse von Futures innerhalb eines Codeblocks ausgelesen und verwendet werden. Dabei ist besonders hervorzuheben, dass die Ausführung des Flow-Blocks asynchron geschieht und der Thread nicht blockiert wird, bis die Ergebnisse der Futures vorliegen, so wie es z.B. bei der Verwendung von `Await.result` der Fall ist.

Es lassen sich in einem Flow-Block auch Variablen angeben, die zu einer späteren Zeit mit einem Wert versehen werden und die bei einem Auslesen innerhalb des Flow-Blocks, so der Wert noch nicht zugewiesen wurde, ebenfalls kein Blockieren des Threads zur Folge haben. Diese Funktionalität wird durch sog. *Promises* erreicht, die das Gegenstück eines jeden Futures darstellen.

Die Rückgabe eines Flow-Blocks ist ein Future, das den Wert enthält, zu dem der Codeblock evaluiert wurde.

Verwendung von Dataflow

Die Voraussetzungen, um die Dataflow-Komponente zu verwenden, sind, dass das Delimited Continuations-Plugin des Scala-Compilers aktiviert sein und die Funktionalität über den Import `import akka.dataflow._` in den Scope gebracht werden muss. Dataflow-Blöcke brauchen außerdem, so wie Futures, einen `ExecutionContext` auf dem sie ablaufen können.

Um einen Flow-Block zu definieren, wird der Methode `flow` eine By-Name-Funktion übergeben. Innerhalb des Blockes greift man auf die Ergebnisse von Futures zu, indem man die `apply`-Methode der Instanz verwendet. Dasselbe gilt für Promises. Promises werden mit einem Wert erfüllt, indem ihnen dieser mit der «-Methode übergeben wird.

Im folgenden Beispiel werden ein Promise und ein Future angelegt und in einem `flow`-Block miteinander kombiniert. In diesem ist definiert, dass der `onSuccess`-Callback auf der durch den Block erzeugten Future-Instanz ausgeführt wird, sobald der Wert des extern definierten Futures erzeugt wurde und auch die Dataflowvariable in Form des Promises mit einem Wert versehen worden ist. Der zweite `flow`-Block erfüllt das Promise mit einem Wert.

```
1 val promise = Promise[String]()
2 val future = Future { "Data" }
3
4 flow {
5   future() + promise()
6 } onSuccess { case x: String => println(x) }
7
8 flow { promise << "flow" }
```

Sobald der zweite Dataflow-Block berechnet wurde, liegt neben dem Wert des Futures, der mit hoher Wahrscheinlichkeit bereits vorher erzeugt wurde, auch das Datum des Promises und damit die beiden für die Erfüllung des durch den ersten Block erzeugten Futures benötigten Werte vor und der Callback gibt den Text ”Dataflow” auf der Konsole aus.

3.4. Spezielle Aktoren

3.4.1. Finite State Machines

Hinweis: da dieses Feature eine größere Anzahl an Facetten hat, wird es nur konzeptionell beschrieben. Für eine genaue Definition siehe Typesafe Inc. (2013).

Der `akka.actor.FSM`-Trait erweitert Aktoren um die Funktionalität von internen Zustandsübergängen, wie sie in der Automatentheorie üblich sind. Hierbei wird eine Reihe von Zuständen definiert, in denen sich ein Aktor befinden kann und Datentypen, mit denen der Aktor in den Zuständen arbeitet.

Jedem Zustand ist hierbei eine Funktion zugeordnet, die bei an den Aktor gesendeten Nachrichten aktiv wird und die Programmlogik des Aktors enthält. Sie hat die Nachricht und den aktuellen Datensatz als Eingabe und den neuen Zustand als Rückgabe und ändert sich üblicherweise bei einem Übergang in einen anderen Zustand, so dass der Aktor in unterschiedlichen Zuständen unterschiedlich auf Nachrichten reagiert.

In jedem Zustand hat man Zugriff auf einen Datensatz in Form eines Objektes einer der definierten Datentypen. Bei einem Übergang in einen anderen Zustand, wird diesem ein zu verwendender Datensatz übergeben – dies kann ein neuer Datensatz oder die alte Instanz sein.

Es wird empfohlen, die Zustände als `case objects` zu definieren. [Typesafe Inc. (2013)] Die Datensätze, die innerhalb der Zustände verfügbar sind, können als `case classes` definiert werden. Sie dienen als Behältnis für die Daten, die für die Aktorenlogik im jeweiligen Zustand interessant sind.[10]

Es folgt ein Beispiel, in dem mehrere Zustände und Datentypen definiert wurden:

```
1 sealed trait State
2 case object Idle extends State
3 case object Active extends State
4
5 sealed trait Data
6 case object Uninitialized extends Data
7 case class Todo(target: ActorRef, queue: Seq[Any]) extends Data
```

Code entnommen aus Typesafe Inc. (2013).

Zustandsübergänge

Es kann Code angegeben werden, der bei Übergängen von einem Zustand in einen anderen ausgeführt wird. Innerhalb dieses Codes hat man Zugriff auf den aktuellen und den Datensatz, der im neuen Zustand benutzt werden wird.

[10] Es empfiehlt sich, für die Zustände und Datentypen jeweils einen Obertyp zu definieren, da der FSM-Trait einen Typparameter für die Zustände und Daten erwartet.

Zustandstimeout

Ähnlich wie bei `receive`-Funktionen kann für die einzelnen Zustände ein Timeout angegeben werden. Wenn dieser verstrichen ist, ohne dass eine Nachricht empfangen wurde, wird dem `FSM`-Aktor eine Nachricht vom Typ `FSM.StateTimeout` zugestellt.

Der Timeout lässt sich bei der Definition eines Zustandes, oder bei Zustandsübergängen angeben, wobei letzteres eine überschreibende Wirkung hat. Der in der Definition gesetzte Zeitraum lässt sich zur Laufzeit ändern und der Timeout deaktivieren.

Nicht verarbeitete Nachrichten

Es lässt sich eine Funktion definieren, die Nachrichten verarbeitet, die nicht vom jeweiligen Zustand verarbeitet worden sind. In ihr lassen sich z.B. Message Handler für Nachrichten definieren, die nicht an einen Zustand gebunden sind und generelle Gültigkeit haben.

Benachrichtigungen über Zustandsübergänge

Andere Aktoren können sich bei einem `FSM`-Aktor über dessen aktuellen Zustand und bei Zustandsübergängen informieren lassen. Für diese Zwecke registrieren sie sich bei dem entsprechenden Aktor durch das Senden einer Nachricht vom Typ `akka.actor.FSM.SubscribeTransitionCallBack`.

3.4.2. Router

Router stellen in Akka eine einfache Möglichkeit dar, mit Teilnehmern einer Gruppe von Aktoren zu kommunizieren. Sie nehmen stellvertretend für ihre Routees Nachrichten entgegen[11] und leiten diese, je nach Routervariante, an einen oder mehrere ihrer zugeordneten Aktoren weiter.

Es gibt in Akka zwei Varianten, um Routern Aktoren zuzuweisen: die Router können ihre Routees automatisch erzeugen oder man kann ihnen vorher angelegte Aktoren zuweisen. Bei der ersten Variante ist der Router gleichzeitig der Supervisor seiner Routees und erzeugt deren Namen synthetisch. Die zu verwendende Supervisionstrategie kann bei der Anlage übergeben werden.

Routees können von Routern automatisch auf verschiedenen Aktorensystemen erzeugt werden. Dazu muss dem Router bei der Anlage eine Liste von Systemen zur Verfügung gestellt werden, auf die er seine Aktoren verteilen soll. Alternativ lassen sich die Adressen

[11] In der tatsächlichen Implementierung, nehmen Router die Nachrichten ihrer Routees aus Performanzgründen nicht selbst entgegen, sondern die Nachricht wird direkt an den entsprechenden Zielaktor gesendet.

der Aktorensysteme auch in der Konfiguration angeben. Die Verteilung der Aktoren auf die unterschiedlichen Systeme erfolgt in einer Round-Robin-Weise.

Wenn Router ihre Routees automatisch erzeugen, kann die Anlage der eingestellten Anzahl an Aktoren sofort erfolgen, oder der Router kann so konfiguriert werden, dass er seinen Aktorenpool nach Bedarf erzeugt. Es lässt sich u.a. eine Ober- und eine Untergrenze der anzulegenden Aktoren festlegen und der Router wird die Größe des Pools innerhalb dieser Grenzen je nach Auslastung der Routees korrigieren, indem er Aktoren erzeugt und terminiert.

In Akka sind folgende Routervarianten vorhanden:

`RoundRobinRouter`

Dieser Router stellt seinen Routees Nachrichten in einer Round-Robin-Weise zu. Betrachtet man die Gruppe der Routees eines Routers als Liste, übermittelt dieser Router Nachrichten, indem er über diese Liste iteriert, dem jeweiligen Aktor die aktuelle Nachricht zustellt und nach dem letzten Routee wieder beim ersten beginnt.

`RandomRouter`

Dieser Router leitet eine ankommende Nachricht an einen zufälligen Aktor aus seiner Gruppe an Routees weiter.

`SmallestMailboxRouter`

Dieser Router stellt eine ankommende Nachricht an den Routee zu, der aktuell über die wenigsten Nachrichten in seiner Mailbox verfügt. Falls Routees im Pool sind, die sich in einem anderen Aktorensystem befinden, bekommen diese die niedrigste Priorität, da deren Mailboxinhalt unbekannt ist.

`BroadcastRouter`

Dieser Router stellt eine ankommende Nachricht an alle seine Routees zu.

`ScatterGatherFirstCompletedRouter`

Dieser Router leitet so wie der `BroadcastRouter` ankommende Nachrichten an alle Routees weiter, jedoch wird nur die als erste erzeugte Antwort an den Sender der Anfrage weitergeleitet. Eventuelle Antworten von anderen Routees, die danach erzeugt wurden, werden vom Router

verworfen. Akka realisiert dies, indem jeder Weiterleitung einer Nachricht an einen Routee eine Instanz in Form eines `Future`s zwischengeschaltet wird, die die Antwort entgegennimmt.

`ConsistentHashingRouter`

Aus Nachrichten, die an diesen Router geschickt werden, muss sich ein Hashkey generieren lassen. Dieser wird intern einem Routee zugeordnet, so dass Nachrichten mit demselben Hashkey an denselben Aktor weitergeleitet werden. Der Hashkey kann auf verschiedene Arten definiert werden: die Nachricht selbst kann den Key enthalten, die Nachricht kann in eine spezielle Klasse gekapselt werden, in der sich der Key angeben lässt, oder das Mapping von Nachricht zu Hashkey wird bei der Anlage des Routers definiert – die Alternativen können auch parallel verwendet werden.

Eigene Router lassen sich implementieren, indem Erweiterungen des Traits `akka.routing.RouterConfig` geschrieben werden.

Unabhängig von der Art des verwendeten Routers, können allen Routees durch Nachrichten der Klasse `akka.routing.Broadcast` Broadcasts gesendet werden. Die Klasse akzeptiert ein Argument vom Typ `Any`, das die Nachricht darstellt, die an die Aktoren übermittelt wird.

3.4.3. Agents

Agents sind Instanzen, die eine Datenstruktur kapseln und einen Zugriff auf diese von mehreren Parteien aus in einer threadsicheren Weise ermöglichen. Der Wert, den ein Agent kapselt, ist eine Instanz einer beliebigen Klasse.

Die Voraussetzung für die Verwendung von Agenten ist, dass sich ein `ActorSystem` als impliziter Wert im Scope befindet bzw. dem Agenten explizit bei der Anlage übergeben wird, da Agenten intern auf Aktoren basieren.

Setzen und verändern des Wertes von Agenten

Der Wert eines Agenten wird verändert, indem dem Agenten neue Werte oder Funktionen übergeben werden, die den aktuellen Wert ändern.

Das Übermitteln der Wertänderungen und deren Anwendung erfolgt asynchron[12] und geschieht über die Methode `send`. Sie akzeptiert sowohl neue Werte als auch Funktionen, die

[12] Wie bei Nachrichten, die an Aktoren gesendet werden, ist es bei Agenten auch der Fall, dass Wertänderungen in keiner deterministischen Reihenfolge durchgeführt werden. Jedoch bleibt die Reihenfolge auf Basis des sendenden Threads erhalten.

den aktuellen Wert verändern, indem sie einen neuen Wert des gekapselten Datentyps als Rückgabe haben.

Die Methode `send` wird standardmäßig in einem Thread aus dem Pool des Default-Dispatchers des verwendeten Aktorensystems ausgeführt. Wird stattdessen die Methode `sendOff` verwendet, läuft die Berechnung in einem separaten Thread ab, der nicht diesem Threadpool angehört.

Auslesen des Wertes von Agenten

Der Wert eines Agenten kann mit der `apply`-Methode jederzeit ausgelesen werden – dieser Vorgang geschieht synchron. Es gilt zu beachten, dass sich aufgrund der Nebenläufigkeit der Änderungsvorgänge nicht vorhersagen lässt, ob zum Zeitpunkt der Rückgabe bereits alle Änderungsvorgänge, die vorher abgesetzt wurden, erfolgt sind. Um dies sicherzustellen, existiert die Methode `await`, die den Wert erst dann zurückgibt, wenn alle zum Zeitpunkt der Anfrage abgesetzten Änderungsvorgänge durchgeführt worden sind. Die Methode `future` tut dasselbe, blockiert jedoch den aktuellen Thread nicht, sondern gibt ein Future auf den zukünftigen Wert zurück.

3.5. Das Aktorensystem

Jede Akka-Anwendung benötigt mindestens ein Aktorensystem, in dem die Aktoren, aus denen sich das Programm zusammensetzt, verwaltet werden, und das die für die Ausführung benötigten Ressourcen bereitstellt.

Das System ist modular aufgebaut und die Bestandteile lassen sich über eine Konfiguration austauschen und modifizieren.

3.5.1. Event Stream

Akka bietet mit dem Event Bus-Feature die Möglichkeit an, zentrale Stellen zu schaffen, die ein Publish-Subscribe-Verfahren für Aktoren bereitstellen. Es funktioniert auf die Weise, dass sich Aktoren bei einem Event Bus für den Erhalt einer bestimmten Art von Nachricht registrieren. Wenn Nachrichten dieses Typs beim Event Bus eingehen, werden sie an die registrierten Aktoren in Form von normalen Nachrichten weitergeleitet.

Der Event Stream ist der zentrale Event Bus von Akka. Standardmäßig werden an ihn Logging-Nachrichten und nicht an Aktoren zustellbare Nachrichten (sog. Dead Letters) publiziert. An ihn können auch durch eigene Aktoren Nachrichten vom Typ `AnyRef` geschickt werden, die an alle für den jeweiligen Typ registrierten Aktoren weitergeleitet werden.

Die Registrierung beim Event Stream wird durch die Methoden `subscribe` und `unsubscribe` gesteuert; Nachrichten werden mit `publish` gesendet. Es stehen zwei Varianten der `unsubscribe`-Methode zur Verfügung, so dass ein Aktor sich für einen Nachrichtentyp oder für alle abmelden kann.

Im folgenden Beispielcode, registriert sich ein Aktor für den Erhalt von `String`s, die an den Event Stream publiziert wurden. Er schickt diesem anschließend selbst eine Nachricht vom Typ `String` und deregistriert sich für diesen Nachrichtentyp, sobald er die selbst publizierte Nachricht erhalten hat.

```
1  import akka.actor._
2
3  class PublishingActor extends Actor {
4    val eventStream = context.system.eventStream
5    eventStream.subscribe(self, classOf[String])
6    eventStream.publish("Nachricht␣an␣mich␣selbst.")
7
8    def receive = {
9      case "Nachricht␣an␣mich␣selbst."
10     => eventStream.unsubscribe(self, classOf[String])
11   }
12 }
```

3.5.2. Dead Letters-Aktor

Der Dead Letters-Aktor ist die Aktoreninstanz, an die Nachrichten zugestellt werden, die an eine `ActorRef` gesendet wurden, die auf keinen aktiven Aktor verweist.

Der Dead Letters-Aktor leitet die erhaltenen Nachrichten in einem `DeadLetter`-Objekt an den Event Stream weiter. Es enthält neben der Nachricht auch den Sender und den ursprünglichen Empfänger derselben.

Zu beachten gilt es, dass Akka auf die Zustellung von Nachrichten an den Dead Letters-Aktor keine Garantie gibt – es muss damit gerechnet werden, dass diese verloren gehen.

3.5.3. Scheduler

Jedes Aktorensystem beinhaltet eine Scheduler-Instanz. Sie wird dazu benutzt, eine beliebige Funktion zu einem zukünftigen Zeitpunkt bzw. periodisch auszuführen.

Der Trait `akka.actor.Scheduler` bietet zu diesem Zweck zwei Methoden an - `scheduleOnce` und `schedule`. Neben den Zeitspannen wird den Methoden eine By-Name-

Funktion übergeben, die ausgeführt werden soll. Falls diese nicht explizit übergeben wird, muss sich ein `ExecutionContext` im Scope des Methodenaufrufs befinden, den der Scheduler für die Ausführung der Funktion benutzen soll. Innerhalb von Aktoren kann der verwendete Dispatcher aus dem `ActorContext` importiert werden, da dieser gleichzeitig einen `ExecutionContext` darstellt.

Die Rückgabe der Scheduler-Methoden ist eine Instanz vom Typ `akka.actor.Cancellable`. Sie dient als Handle auf die auszuführende Funktion, mit dem sich der Vorgang über die Methode `cancel` abbrechen lässt.

Der Scheduler führt die Funktionen aus Performanzgründen nicht exakt zur angegebenen Zeit, sondern leicht versetzt aus. Standardmäßig beträgt die maximale Zeitversetzung 100 Millisekunden – sie lässt sich über den Wert `tick-duration` in der Konfiguration anpassen. Niedrigere Werte machen den Scheduler genauer, erhöhen jedoch den für die Verwaltung nötigen Aufwand.

Der Aktor im folgenden Code übergibt eine Funktion an den Scheduler, die im Abstand von fünf Sekunden ausgeführt wird und diesem eine Nachricht sendet. Sobald der Aktor die erste Nachricht empfangen hat, beendet er den Scheduling-Vorgang.

```
1  import akka.actor._
2  import scala.concurrent.duration._
3
4  class SchedulingActor extends Actor {
5    import context.dispatcher
6
7    val heartbeatSender = context.system.scheduler.schedule(
8      initialDelay = 5 seconds, interval = 5 seconds){
9        self ! "Heartbeat"
10     }
11
12   def receive = {
13     case "Heartbeat"
14     => heartbeatSender.cancel()
15   }
16 }
```

3.5.4. Dispatcher

Jeder Aktor ist in Akka einem Dispatcher zugeordnet. Dispatcher stellen in einem Aktorensystem die Threads zur Verfügung, die die Aktoren brauchen um arbeiten zu können. Wurde

einem Aktor eine Nachricht gesendet, wird er durch den Dispatcher aktiviert, indem dieser ihm die Nachricht zustellt.

Es kann in einem Aktorensystem eine beliebige Anzahl an Dispatchern geben und es gibt verschiedene Varianten, die sich über die Konfiguration einstellen und anpassen lassen.

Bei der Anlage eines Aktors, wird der zu verwendende Dispatcher über die Methode `withDispatcher` der `Props`-Instanz festgelegt. In diesem Zuge muss der Methode ein `String` übergeben werden, der auf einen in der Konfiguration definierten Dispatcher verweist. Falls über diese Methode kein Dispatcher explizit festgelegt wird, verwendet Akka den Standard-Dispatcher für den Aktor.

Es stehen in Akka vier Grundvarianten an Dispatchern zur Verfügung:

Dispatcher

Dies ist die Variante, die als Standard für den Default-Dispatcher definiert ist. Dieser Dispatcher verwendet einen Threadpool, um Aktoren aktiv werden zu lassen und tut dies immer dann, wenn Nachrichten für den jeweiligen Aktor in dessen Mailbox vorhanden und Threads frei sind.

PinnedDispatcher

Dieser Dispatcher reserviert einen eigenen Thread für jeden Aktor, der ihm zugewiesen wird.

BalancingDispatcher

Der `BalancingDispatcher` verwendet wie der `Dispatcher` einen Threadpool für seine Aktoren, jedoch teilen diese sich eine Mailbox. Das hat zur Folge, dass Nachrichten, die an einen Aktor geschickt werden, der einen solchen Dispatcher verwendet, an einen zufälligen Aktor desselben zugestellt werden und nicht zwingend an den Aktor, dessen `ActorRef` verwendet wurde.

Es gilt zu beachten, dass diesem Dispatcher keine Router-Instanzen zugeordnet werden können.

CallingThreadDispatcher

Dieser Dispatcher aktiviert den Aktor, an den die Nachricht gesendet wurde, auf dem Thread, der diese geschickt hat – die Nachricht wird also synchron verarbeitet.

Die Dispatcher, die einen Threadpool verwenden, arbeiten im Hintergrund mit einer Instanz des `java.util.concurrent.ExecutorService`. Über den Eintrag `executor` kann in der Konfiguration eines Dispatchers festgelegt werden, ob dieser eine Instanz vom Typ `fork-join-executor` oder `thread-pool-executor` verwenden soll. Es können auch eigene Implementierungen verwendet werden, die die Klasse `akka.dispatch.ExecutorServiceConfigurator` erweitern. Der Standard-Dispatcher verwendet eine `fork-join-executor`-Instanz.

Eigene Dispatcher lassen sich implementieren, indem die Klasse `akka.dispatch.MessageDispatcherConfigurator` erweitert wird.

Außerdem lässt sich, über den Wert `throughput` in der Konfiguration, für solche Dispatcher definieren, wie viele Nachrichten ein Aktor maximal verarbeitet, bis der Thread beginnt, an einen anderen Aktor Nachrichten zuzustellen. Der Standardwert des Default-Dispatchers ist 5.

3.5.5. Mailboxen

Zu jedem Aktor existiert eine Mailbox. Akka verwendet sie um Nachrichten zwischenzuspeichern, die an Aktoren gesendet wurden. So wie bei den Dispatchern gibt es auch mehrere Mailbox-Implementierungen. Die Variante der zu verwendenden Mailbox bzw. Mailboxen ist an den Dispatcher gekoppelt und pro Dispatcher-Instanz kann nur eine Mailbox-Variante festgelegt werden.

Bei allen Dispatcher-Varianten außer dem `BalancingDispatcher` besitzt jeder Aktor eine eigene Mailbox – bei `BalancingDispatchern` existiert nur eine Mailbox für alle Aktoren, die diesem zugeordnet sind.

Es lassen sich eigene Mailboxtypen implementieren, in Akka mitgeliefert werden die folgenden:

`UnboundedMailbox`

Dies ist die Standardvariante – es handelt sich um eine Mailbox, die so lange Nachrichten aufnimmt, bis der Speicher des Systems gefüllt ist.

`BoundedMailbox`

Diese Mailbox hat ein Limit für die Anzahl an Nachrichten, die sie aufnehmen kann. Wenn das Limit erreicht ist, werden die gesendeten Nachrichten nach Ablauf eines konfigurierbaren Timers an den Dead Letters-Aktor zugestellt.

Priority Mailboxen

Die beiden erstgenannten Mailboxtypen gibt es jeweils in einer Variante, die eine Priorisierung der Nachrichten vornimmt und zuerst die mit einer hohen Priorität zustellt.

Hierfür muss ein eigener Subtyp der entsprechenden Mailboxdefinition implementiert werden, der eine Instanz der Klasse `akka.dispatch.PriorityGenerator` enthält. In ihr ist definiert, welche Prioritäten den Nachrichten zugeordnet werden.

Dies geschieht über eine Funktion, die den Nachrichten einen Integer-Wert zuordnet. Je niedriger der Wert, desto höher ist die Priorität.

Im folgenden Beispiel ist die Definition einer `UnboundedPriorityMailbox` angegeben, die eine Priorisierung von Nachrichten vornimmt:

```
1  import akka.dispatch.PriorityGenerator
2  import akka.dispatch.UnboundedPriorityMailbox
3  import com.typesafe.config.Config
4
5  class MyPrioMailbox(settings: ActorSystem.Settings, config: Config)
6    extends UnboundedPriorityMailbox(
7      PriorityGenerator {
8        case "highpriority" => 0
9        case "lowpriority"  => 2
10       case otherwise      => 1
11     })
```

Code entnommen aus Typesafe Inc. (2013); gekürzt und modifiziert.

Durable Mailbox

Die Besonderheit der Durable Mailbox ist, dass die Nachrichten auf einem Permanentspeichermedium gesichert werden, so dass sie nicht verloren gehen, wenn das Aktorensystem unvorhergesehen beendet wird. Nach einem Neustart des Systems stehen den Aktoren, die eine solche Mailbox verwenden, die noch nicht zugestellten Nachrichten zur Verfügung.

3.5.6. Remoting

Das Remoting-Feature bietet die Möglichkeit, `ActorRefs` von Aktoren zu bekommen, die auf anderen Systemen platziert sind und Aktoren statt auf dem lokalen auf einem entfernten Aktorensystem zu erzeugen.

Voraussetzungen

Um Remoting nutzen zu können, muss in der Konfiguration festgelegt sein, dass statt der lokalen `ActorRef`-Implementierung jene genommen wird, die Referenzen erzeugt, die über mehrere Aktorensysteme hinweg Gültigkeit haben – dafür muss der Wert `akka.actor.provider` auf den Wert `akka.remote.RemoteActorRefProvider` eingestellt werden.

Es müssen außerdem ein Hostname und ein Port festgelegt werden, über die das jeweilige Aktorensystem im Netzwerk zu erreichen ist. Ferner kann die Art der Verbindung in der Konfiguration angegeben werden und ob bzw. wie diese verschlüsselt sein soll.

Darüber hinaus gilt es zu beachten, dass sich Objekte, die als Nachrichten von einem System zu einem anderen geschickt werden, serialisieren lassen müssen.

Außerdem müssen die Klassen der Aktoren, die auf einem entfernten System erzeugt werden, durch den Classloader desselben verfügbar sein.

Nachrichten an Remote-Aktoren

Zu Remote-Aktoren gehörige `ActorRef`s können mit der `actorFor`-Methode durch das lokale Aktorensystem erzeugt werden. Hierzu muss eine URL für den Aktor angegeben werden, die den Hostnamen, den Port und den Namen des Aktorensystems enthält, auf dem sich der entsprechende Aktor befindet.

Anschließend kann die generierte `ActorRef` auf dieselbe Weise benutzt werden, wie Referenzen, die auf lokale Aktoren verweisen.

Aktoren auf entfernten Aktorensystemen erzeugen

Aktoren lassen sich nicht nur auf dem lokalen Aktorensystem, sondern auch auf externen Systemen erzeugen. Dabei ändert sich nichts an der logischen Aktoren- und Supervisionhierarchie – Aktoren können einen Supervisor haben, der sich nicht in demselben Aktorensystem befindet.

Aktoren können automatisch oder manuell auf einem entfernten System angelegt werden. Automatisch geschieht dies, indem das entfernte System in der Konfiguration angegeben und eine URI des Aktors festgelegt wird, der in diesem erzeugt werden soll. Nun kann der über die URI definierte Aktor auf die herkömmliche Weise im lokalen Aktorensystem angelegt werden und die Instantiierung geschieht im entfernten System. Innerhalb der URI lassen sich Wildcards verwenden, so dass sich bspw. angeben lässt, dass alle Aktoren eines bestimmten Unterbaumes auf einem entfernten System erzeugt werden sollen.

Die andere Weise Aktoren entfernt zu erzeugen besteht darin, das entfernte System über die in der `Props`-Klasse definierte Methode `withDeploy` bei der Anlage des Aktors anzugeben.

Remote Events

Die Remoting-Komponente erzeugt eine Reihe von Events, die sich über den Event Stream empfangen lassen. Bspw. wird ein Event geloggt, wenn ein entferntes System sich erfolgreich verbunden hat, terminiert wurde o.Ä.

Einschränkungen bei der DeathWatch

Zu beachten gilt es, dass das DeathWatch-Feature zwar mit entfernten Aktorensystemen funktioniert, jedoch keine `Terminated`-Nachricht geschickt wird, falls die Verbindung zu dem System abreißt, auf dem sich der Remote-Aktor, auf den die DeathWatch gehalten wird, befindet.

3.6. Weitere Komponenten

Akka besitzt eine Reihe weiterer Komponenten, die in diesem Kapitel nicht oder nur am Rande behandelt wurden. Eine Auswahl davon wird in diesem Abschnitt kurz zusammengefasst:

Circuit Breaker bilden einen Wrapper um Funktionen. Schlagen diese zu oft fehl, entweder durch Exceptions oder Timeouts, so wirft der Wrapper bei nachfolgenden Aufrufen sofort eine Exception, ohne die Funktion auszuführen. Das Verhalten stellt sich nach einiger Zeit zurück, um zu testen, ob zukünftige Funktionsaufrufe wieder in der Lage sind, erfolgreich abzulaufen.

Das Feature kann dazu verwendet werden, Teile des Programms zu deaktivieren, wenn diese ausfallen, um Ressourcen zu schonen, die bei einer erfolglosen Ausführung der entsprechenden Funktion vergeudet würden.

ActorLogging ist eine einfache Möglichkeit, Informationen in einem Event Bus zu loggen und auf der Konsole ausgeben zu lassen.

Transactors sind eine Art von Aktoren, die in der Lage sind, an verteilten Transaktionen teilzunehmen.

Akka IO bietet eine Integration von Netzwerk-Sockets in Aktoren an.

TestKit stellt Werkzeuge zur Verfügung, mit denen sich Aktorensysteme leichter auf Korrektheit testen lassen.

Microkernel stellt eine Akka-Anwendung samt den benötigten Komponenten in einem Paket zur Verfügung.

4. Leitlinien für die Programmierung mit Akka und Design Pattern

Das Akka-Framework bietet eine Vielzahl an Funktionen und Möglichkeiten an, diese einzusetzen. Es ist nicht immer sofort klar, welche Vorgehensweise die erfolgversprechendste für das aktuelle Problem ist.

Dieses Kapitel stellt eine Zusammenfassung von Leitlinien für den Umgang mit den Features des Systems dar und erläutert danach einige Design Pattern, die als Lösungsansätze für häufig auftretende Probleme dienen sollen.

Hinweis: die Inhalte dieses Kapitels lehnen sich zu großen Teilen an das Buch Wyatt (2013) an.

4.1. Leitlinien

Die Einhaltung der dargestellten Leitlinien sorgt dafür, dass die Performanz, Fehlertoleranz und die Codequalität im Gesamten steigt, was sich letztendlich direkt auf die Qualität des Endproduktes auswirken kann.

Regeln des Aktorenmodells einhalten

Interne Variablen von Aktoren nur intern verwenden

Ein wichtiges Prinzip des Aktorenmodells besagt, dass der State, die Variablen, die ein Aktor intern definiert hat und verwendet, nicht in anderen Teilen des Programms referenziert sein dürfen. Das ist notwendig, da sonst ein Shared State entstehen und mit diesem die Problematiken des entsprechenden Paradigmas Einzug halten würden. Es ist zwar möglich, einen solchen Shared State wie eine unveränderliche Datenstruktur zu behandeln und ausschließlich lesend darauf zuzugreifen, jedoch kann dies in komplexen Programmen schwer sichergestellt werden und ist damit anfällig für Laufzeitfehler, die, wie bereits beschrieben, schwierig zu entdecken sein können.

Der State eines Aktors kann auf vielfältige Weise aus seinem Kontext nach außen gelangen: bspw. durch Nachrichten an andere Aktoren oder Funktionen die an den Scheduler geschickt oder als Futures verwendet werden und intern Referenzen auf Variablen haben.

Unproblematisch ist hingegen die Weitergabe von unveränderlichen Datenstrukturen. Diese können von mehreren Threads parallel verarbeitet werden, ohne dass nachteilige Effekte entstehen würden.

Immutable Nachrichten

Eine ähnliche Problematik besteht bei Nachrichten, die Datenstrukturen verwenden, die veränderbar sind: falls solche Nachrichten an mehrere Aktoren weitergeleitet werden, besteht die Gefahr, dass ein Shared State entsteht, wenn mehr als eine Instanz Referenzen auf die Daten hält. Um diese Problematik von vornherein auszuschließen, sollten Nachrichten ausschließlich Datenstrukturen enthalten, die unveränderlich sind.

Nachrichten sollten keine Literale sein

In vielen der Codebeispiele aus dem 3. Kapitel wurden Strings als Nachrichten an Aktoren verwendet. Dies geschah lediglich, um den Text möglichst kurz zu halten. In der Realität sollte man darauf verzichten, Literale als Nachrichten zu verwenden, da die Nachricht im Falle eines Schreibfehlers nicht mehr verarbeitet werden kann und der Fehler möglicherweise erst zur Laufzeit auffällt. Weil der Nachrichtentyp im Pattern Matching der `receive`-Funktion ein String ist, kann der Compiler bei Schreibfehlern keinen Fehler erzeugen und akzeptiert den Code.

Benutzt man stattdessen für Nachrichten das Typsystem und versendet statt eines Strings bspw. ein `case object`, so kann der Compiler Schreibfehler effektiv erkennen und so zu einem korrekten Programmablauf beitragen.

Message Handler von Aktoren zusammensetzen

Oftmals ist es praktikabel, den Message Handler für einen Aktor in mehrere Stücke aufzuteilen, etwa, weil man die Implementierung eines Aktors aus mehreren Traits zusammensetzt, in denen verschiedene Teil-Message Handler definiert sind.

Um dies zu tun, kombiniert man die Funktionen mit Hilfe des `orElse`-Combinators, der in `PartialFunction`s definiert ist.

Aktorenlogiken kurz halten

Wenn der Message Handler eines Aktors aktiviert wird, läuft die `receive`-Funktion in einem Stück ab, bis sie verlassen wird. Das bedeutet, dass während dieser Zeit auf dem entsprechenden CPU-Kern kein anderer Aktor oder Thread aktiv werden und Daten verarbeiten kann. Wenn die Message Handler einer großen Anzahl an Aktoren besonders lang sind, kann deswegen die Reaktionszeit des Programms reduziert sein, da andere Aktoren, die darauf warten zum Zuge zu kommen, eine längere Zeit davon abgehalten werden.

Dies sollte beachtet werden, falls die Anforderung an das Programm besteht, eine geringe Reaktionszeit auf Anfragen zu haben.

Aktorenlogik begrenzen: Single Responsibility

Es ist nicht nur aufgrund allgemeiner Überlegungen sinnvoll, den Umfang der Funktionalität eines Aktors zu begrenzen, etwa um das Programm leichter nachvollziehbar zu machen, sondern auch, wegen der eingeschränkten Möglichkeiten der Supervision-Komponente von Aktoren. Diese kennt nur Exception-Typen, denen sie eine Maßnahme zuweist.

Falls ein Aktor ein größeres Spektrum an Aufgaben abdeckt, ist es wahrscheinlich, dass er denselben Fehlertyp aufgrund von unterschiedlichen Operationen werfen kann. Wenn eine dieser Operationen einen Restart erfordert, die andere jedoch einen Resume, kann der Supervisor keine richtige Entscheidung mehr treffen, da er diesem Fehler nur eine Maßnahme zuordnen kann.

Funktionalität erweitern durch das Zwischenschalten von Aktoren

Aktoren akzeptieren jede Art von Nachricht. Wenn man bereits über eine Funktionalität verfügt, die durch den Austausch einer Nachricht von zwei Aktoren zustande kommt, kann man diese unter Umständen leichter erweitern, wenn man einen oder mehrere Aktoren dazwischenschaltet, ohne dass man die bestehenden Aktoren verändert. Dies ist immer dann leicht möglich, wenn die neue Funktionalität isoliert von der bestehenden existieren kann – etwa wenn eine Nachricht an eine Instanz repliziert oder verändert werden soll, bevor sie beim Empfänger eingeht.

Wichtige Aktoren mit einer DeathWatch belegen

Wann immer sich ein Aktor auf die Funktion eines anderen Aktors verlässt, sollte der entsprechende Aktor mit einer DeathWatch versehen werden. Im Falle eines Ausfalls können so

entsprechende Maßnahmen getroffen werden um die Funktion des Programms wiederherzustellen.

Außerdem kann auf diese Weise festgestellt werden, ob Nachrichten, die an einen Aktor gesendet werden, ins Leere, d.h. zum Dead Letters-Aktor, gehen oder tatsächlich an den entsprechenden Aktor zugestellt werden bzw. dieser überhaupt existiert, falls die Referenz mit der `actorFor`-Methode über die URI erzeugt wurde.

Futures bevorzugt verwenden

Futures und Aktoren haben eine Reihe an Gemeinsamkeiten. Man kann beide als Konstrukte verstehen, die eine Eingabe bekommen und in der Regel eine Ausgabe produzieren.

Aktoren haben hierbei einen State, der sich über ihren Lebenszyklus verändert und können durch ihre Fähigkeit, Nachrichten zu empfangen, mehr als eine Eingabe und von verschiedenen Quellen bekommen.

Das Alleinstellungsmerkmal des Futures ist, dass ein Handle auf die Berechnung existiert, das es etwa erlaubt, die Reihenfolge für mehrere (Teil-)Berechnungen festzuhalten. Ferner sind Futures wesentlich einfacher in der Handhabung, da sie als Literale definiert werden und einen geringen deklarativen Aufwand benötigen und sich einfacher verknüpfen lassen. Zudem haben sie einen geringeren Performanceoverhead als Aktoren.

Der State, den Aktoren besitzen, und ihre Möglichkeit, Nachrichten zu empfangen, sind ihre wesentlichen Vorteile. Sie erlauben es Aktoren, auf die Außenwelt zu reagieren. Immer wenn dies nicht benötigt wird, sollte statt einem Aktor, ein Future verwendet werden. Oft ist dies der Fall, wenn sich die gewünschte Funktionalität in einer isolierten Funktion ausdrücken lässt, die schon bei der Anlage alle nötigen Daten besitzt um durchlaufen zu können bzw. diese durch ein vorgeschaltetes Future erhalten wird.

Latenzen der Nachrichten beachten

Aufgrund der Tatsache, dass Rechenressourcen knapp werden können und Aktoren über eine Mailbox verfügen, die voll werden kann, kann man die Zeitspanne, die zwischen dem Senden einer Nachricht und deren Verarbeitung besteht, nie sicher voraussagen. Man sollte deshalb beachten, dass sie variabel ist und länger als gewünscht ausfallen kann. Eine Möglichkeit, die Funktion des Programmes dennoch sicherzustellen, ist bei Anfragen an andere Aktoren mit Timeouts oder der Circuit Breaker-Funktionalität zu arbeiten.

Aufgrund der Latenz sollte man Anfragen nicht als durchgeführt betrachten, nachdem sie abgesetzt wurden, sondern erst zu dem Zeitpunkt, an dem sie von dem durchführenden Aktor verarbeitet werden.

Diese Denkweise harmoniert auch gut mit dem Faktum, dass die Nachrichtenzustellung in entfernte Aktorensysteme nicht zuverlässig ist und Nachrichten verloren gehen können. Je nach Situation muss hier mit Timeouts und ACKs gearbeitet bzw. Gebrauch von dem sog. *Reliable Proxy Pattern* gemacht werden, falls ein Verlust von Nachrichten nicht akzeptabel ist.

Das Blockieren von Threads vermeiden

Das Blockieren von Threads sollte vermieden werden, da es in vielen Fällen eine Vergeudung von Ressourcen bedeutet. Je stärker ein Programm Gebrauch von Blocking macht, um so mehr Threads werden im Pool des Aktorensystems benötigt, um das Programm am Laufen zu halten und eine gute Auslastung der CPU-Kerne zu erreichen.

Die zusätzlichen Threads stellen jedoch einen Overhead dar, der möglichst klein gehalten werden sollte. Threads kosten Hauptspeicher und je mehr davon existieren, desto öfter wird das Betriebssystem diese de- bzw. aktivieren, um sicherzustellen, dass kein Thread vernachlässigt wird. Dies benötigt zusätzlich Rechenressourcen, da der Zustand des alten Threads jeweils gesichert und der des neuen wiederhergestellt werden muss.

Einen dedizierten Threadpool für I/O verwenden

In vielen Programmen muss synchrones I/O ausgeführt, müssen also Daten mit externen Ressourcen ausgetauscht werden. Während dieses Vorgangs werden ebenfalls Threads benötigt, die jedoch keine Arbeit auf der CPU verrichten.

Es ist praktikabel, den Aktoren, die hauptsächlich solche Aufgaben verrichten, einen eigenen Dispatcher und damit einen dedizierten Threadpool zuzuweisen. Das Nutzungsverhalten dieser Threads unterscheidet sich von dem anderer Aktoren und ein separater Dispatcher ermöglicht eine bessere Anpassung an die Bedürfnisse des Programms und die Schonung von Ressourcen, da das Austauschen von Threads durch das Betriebssystem minimiert wird.

Dedizierte Supervisor verwenden

Befinden sich auf einer Hierarchiestufe Aktoren, die eine unterschiedliche Supervisionstrategie erfordern, lohnt es sich oftmals, für jede benötigte Strategie einen eigenen Aktor zu definieren. Solche Aktoren haben keinen anderen Zweck, als die passende Supervision für eine Gruppe von Aktoren bereitzustellen.

Kritische Aufgaben an periphere Aktoren übertragen

Wenn ein Aktor eine Exception wirft, hat das oftmals auch Auswirkungen auf seine Nachkommen – zumindest, wenn eine andere Supervisionstrategie als Resume angewendet wird. Falls sich dieser Aktor auf einer oberen Ebene der Aktorenhierarchie befindet, kann der Aufwand, der betrieben werden muss, um insb. eine Restart-Direktive anzuwenden, enorm sein und negative Effekte auf das Programm haben, falls die Eventualität nicht beachtet wurde.

Deswegen ist es ratsam, Aufgaben, die Exceptions werfen können, an Aktoren zu delegieren, die keine Nachkommen haben. Es reduziert den Aufwand, der sonst betrieben werden müsste und ermöglicht es dem Programmierer, einen Stil anzuwenden, der Exceptions nicht unter allen Umständen zu vermeiden sucht, da die Terminierung von Aktoren einen geringen Overhead verursacht.

Mehr Dependency Injection und weniger `actorFor`

Durch die mit der `actorFor`-Methode verwendeten URI entsteht eine starke Koppelung an die bestehende Aktorenhierarchie. Werden Aktorenreferenzen hingegen über den Konstruktor eines Aktors übergeben, erfolgt die Bindung dynamisch. Das ist wünschenswert, weil der entsprechende Code angepasst werden muss, falls sich die Hierarchie ändert, im Falle der Dependency Injection jedoch nicht.

Ein weiterer Nachteil der Verwendung der `actorFor`-Methode besteht darin, dass diese in jedem Fall eine `ActorRef` zurückliefert, also auch dann, wenn ein Schreibfehler in der URI existiert.

Tunebare Parameter in die Konfiguration auslagern

Die Größe des für einen Dispatcher zu verwendenden Threadpools ist ein gutes Beispiel für einen Parameter, der gut in der Konfiguration definiert werden kann. Der Umstand, dass sich dieser Parameter in der externen Konfiguration befindet, erlaubt es Personen, diese Charakteristik des Programms zu ändern, ohne es modifizieren zu müssen.

Es ist möglich, eigene Parameter in die Konfiguration auszulagern, und dies sollte geschehen, falls der Wert des Parameters zum Zeitpunkt der Programmierung unklar ist oder sich je nach Systemkonfiguration ändern kann. Dies erlaubt es den Anwendern des Programms, selbsttätig eine Optimierung durchzuführen.

4.2. Design Pattern

4.2.1. Aktoren-Initialisierung im laufenden Betrieb fortsetzen

Aktoren lassen sich nicht immer vollständig bei der Anlage durch die `actorOf`-Methode initialisieren. Dies kann z.B. den Grund haben, dass sie Daten benötigen, die ihr Supervisor nicht besitzt und die deshalb nach der Anlage angefordert werden müssen.

Dies stellt jedoch erst ein Problem dar, wenn der Aktor während der Initialisierungsphase Anfragen bekommt, auf die er reagieren muss. Da er zu der Zeit in der diese eingehen nicht in der Lage ist, die gewünschte Funktionalität durchzuführen, muss er, bzw. sein Client, eine alternative Strategie anwenden, falls es nötig ist, dass keine Anfrage verloren geht.

Es gibt drei Alternativen, mit dieser Situation umzugehen: die Anfragen können ignoriert werden, es kann eine Absage an das Request gesendet werden oder die Anfragen können nach hinten gestellt werden.

Anfragen ignorieren

Werden die kommenden Anfragen ignoriert, muss der Client diese mit einem Timeout versehen. Wenn der Timeout abläuft, kann der Client eine alternative Aktion durchführen.

Diese Alternative besitzt einen relativ hohen Performance-Overhead, da jede Anfrage mit einem Timeout versehen werden muss, auch wenn dies die meiste Zeit über nicht nötig wäre.

Der Vorteil dieser Vorgehensweise ist, dass für alle Anfragen ein Timeout existiert, der auch dann nützlich sein kann, wenn der angefragte Aktor bereits initialisiert, aber überlastet ist.

Anfrage absagen

Der angefragte Aktor kann die Requests auch mit einer Absage beantworten, solange er noch nicht in der Lage ist, sie zu beantworten.

Der größte Nachteil dieser Alternative ist, dass die Absage im Client der Anfrage zugeordnet werden muss, damit er weiß, welche seiner Anfragen verworfen wurde – so wie es auch bei der Timeout-Lösung geschehen muss. Diese Zuordnung kann etwa durch eine Anfragen-ID geschehen, die in die Absage enthalten ist und durch den Client verstanden wird.

Vorteilig ist, dass die Anfrage relativ schnell abgebrochen wird und der angefragte Aktor wenig Aufwand treiben muss.

Anfragen zurückstellen

Akka bietet mit dem `Stash`-Trait die Möglichkeit an, Nachrichten zwischenzuspeichern. Dazu wird innerhalb der `receive`-Funktion die Methode `stash()` aufgerufen. Wenn die Initialisierung abgeschlossen ist, können die zwischengespeicherten Nachrichten mit der Methode `unstashAll()` in die Mailbox umgelagert werden, so dass sie vom Aktor erneut verarbeitet werden.

Diese Methode hat folgende Nachteile: um den Trait benutzen zu können, muss der Aktor über einen speziellen Mailboxtyp verfügen, nämlich die `UnboundedDequeBasedMailbox`. Ferner gilt zu beachten, dass die zwischengespeicherten Nachrichten keinen Restart des Aktors überleben.

Hinsichtlich der Klassen-Linearisierung von Scala gilt: da der Trait den `preRestart()`-Lifecycle Hook überschreibt, muss er an letzter Stelle der Trait-Mixin-Liste stehen.

4.2.2. Request/Response mit Aktoren

Wann immer ein Aktor eine Anfrage an einen anderen Aktor stellt, stellt sich die Frage, wie mit der Antwort umzugehen ist. Der anfragende Aktor muss beim Eintreffen der Antwort wissen, zu welcher Anfrage diese gehört, damit er sie sinnvoll verarbeiten kann – er braucht dazu einen Kontext.

Im Folgenden werden verschiedene Wege beschrieben, die Beantwortung einer Anfrage mit einem Kontext zu versehen.

Lösung mit Futures

Wenn man die Anfrage statt als normale Nachricht mit der `ask`-Methode sendet, besitzt man ein Future auf die Antwort und kann diese durch angehängte Funktionen verarbeiten und an andere Aktoren weiterleiten.

Hierbei besteht die Einschränkung, dass die Response nicht durch den anfragenden Aktor, sondern durch die auf dem Future definierten Funktionen verarbeitet wird. Dies ist oftmals nicht akzeptabel.

Positiv an dieser Lösung ist, dass man mit dem Future einen Timeout zu der Anfrage bekommt, auf dessen Auslösung man reagieren kann.

Lösung mit temporären Aktoren

Alternativ zu den Futures, kann man auch Aktoren verwenden, die für jede Anfrage eigens angelegt werden. Hierdurch ist man in der Lage ein breiteres Spektrum an Funktionalität

abzudecken, da Aktoren einen State haben und Nachrichten empfangen können, hat jedoch einen höheren deklarativen Overhead und nicht zwangsweise einen Timeout.

Kontext im State des Clients zwischenspeichern

Wenn der anfragende Aktor den Kontext in seinem State speichert, kann die Behandlung der Antwort in dem anfragenden Aktor geschehen.

Der anfragende Aktor könnte dazu etwa die Anfragen, die er an einen anderen Aktor stellt, in einer Liste zwischenspeichern. Da die Reihenfolge von Nachrichten, die derselbe Aktor gesendet hat, erhalten bleibt, entspricht die Reihenfolge, in der die Antworten auf die Anfragen eingehen, jener der zwischengespeicherten Anfragen. Dadurch kann eine Zuordnung von Antworten an Anfragen gemacht und der Kontext wiederhergestellt werden.

Der Client würde bei dieser Lösungsvariante eine Liste mit Anfragen pro Server-Aktor führen. Es gilt jedoch zu beachten, dass die Logik des Servers so arbeiten muss, dass die Anfragen der Reihe nach beantwortet werden und keine Antworten verloren gehen dürfen, da es sonst zu falschen Zuordnungen kommen würde.

Eine Erweiterung dieses Lösungsmusters, die die genannten Probleme auflöst, ist, jede Anfrage mit einer ID zu versehen und diese auch in der Nachricht zum Server zu schicken. Dieser würde die ID dann in der Antwort einschließen, so dass der Client die Zuordnung des Kontextes über die ID machen kann. Alternativ kann der Kontext komplett in die Nachricht übernommen werden – siehe dazu das nachfolgende Lösungsmuster.

Kontext in die Antwort inkludieren

Der Client braucht keinen Kontext zwischenzuspeichern, wenn der Server diesen in der Antwort mitschickt.

Dies hat den Vorteil, dass der Kontext einen Restart des Clients überlebt und der Client weniger Arbeitsspeicher benötigt, da er keine temporären Datenstrukturen zu halten braucht.

Nachteilig ist, dass der Server so strukturiert werden muss, dass er den Kontext berücksichtigt, was die Koppelung zwischen den beiden Parteien erhöht. Außerdem stellen die zusätzlichen Daten einen Overhead dar, falls diese über ein Netzwerk gesendet werden.

4.2.3. Algorithmen aufteilen

Um die Dauer, die ein Message Handler am Stück aktiv ist zu reduzieren, kann es erforderlich sein, einen Algorithmus so zu strukturieren, dass die Aufgabe in mehrere Teilaufgaben unterteilt

wird. Die benötigte Zeit pro Teilaufgabe kann so relativ zu der Gesamtlaufzeit des Algorithmus reduziert werden.

Besonders einfach ist es, solche Algorithmen zu unterteilen, die es erfordern, die gleiche Funktion x-mal mit unterschiedlichen Ausgangsdaten auszuführen und die Ergebnisse zu aggregieren. In diesem Fall, stellt ein Aufruf der Funktion eine Teilaufgabe dar und das aggregierte Ergebnis den Abschluss der Gesamtaufgabe.

Hat man eine Funktionalität in Teilaufgaben unterteilt, kann man diese entweder sequentiell ablaufen lassen, um die Kapazität des Systems für andere Aufgaben zu erhöhen, oder diese parallel laufen lassen, um die Dauer bis zum Abschluss der Aufgabe auf Kosten der Restkapazität zu reduzieren.

Sequentielle Ausführung

Der Aktor kann den Algorithmus so aufteilen, dass er die noch zu erledigenden Teilaufgaben und die Ergebnisse bereits abgeschlossener in einer Nachricht vermerkt. Zunächst enthält die Liste der Aufgaben alle zu erledigenden Teilaufgaben und die Liste der Ergebnisse ist leer.

Der Aktor schickt sich nun selbst diese Nachricht und erledigt die erste Teilaufgabe. Wenn dies erfolgt ist, streicht er sie von der Liste und fügt das Ergebnis zu der Liste der Ergebnisse hinzu und schickt sich die aktualisierte Nachricht. Dies wiederholt sich so lange, bis die Aufgabenliste leer ist – die Ergebnisliste stellt die Aggregation der Teilaufgaben und damit das Endergebnis dar.

Diese Vorgehensweise bewirkt, dass der Aktor die Aufgabe nicht am Stück durchführt. Er aktiviert sich selbst für jede Teilaufgabe erneut, indem er sich eine Nachricht schickt.

Parallele Ausführung

Wenn sich die Teilaufgaben parallelisieren lassen, können diese durch Futures parallel ausgeführt werden. Hierbei werden diese nicht durch Nachrichten sequentialisiert, sondern die Liste der Teilaufgaben wird in eine Liste von Futures umgewandelt. Mittels der `Future.sequence()`-Methode können die Ergebnisse asynchron aggregiert und anschließend an einen Aktor weitergeleitet werden.

5. Simulationssoftware einer Warenkommissionierung durch autonome Roboter

5.1. Einleitung

Im Folgenden wird eine Software beschrieben, die mithilfe des Akka-Frameworks und der Programmiersprache Scala realisiert wurde. Die Implementierung repräsentiert ein Warenlager, das von Robotern befahren wird, um Artikel für eingehende Bestellungen zu kommissionieren und diese anschließend an eine Packstation zu übergeben.

5.2. Funktionsweise der Simulation

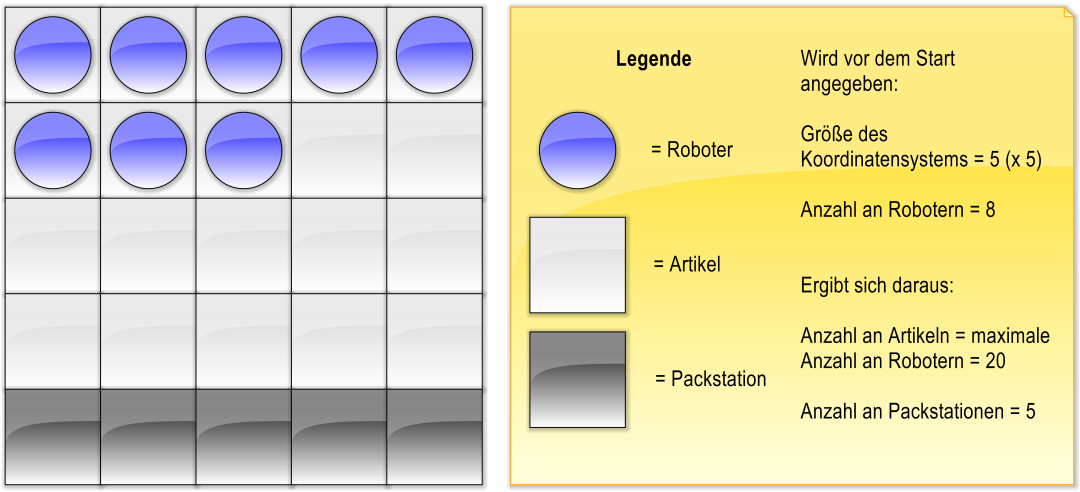

Abbildung 5.1.: Warenlager mit Robotern bei Simulationsstart

Warenlager

Das Warenlager besteht aus einem in der Größe frei definierbaren Koordinatensystem, auf dem sich die Artikel befinden. Jede Koordinate steht für einen anderen Artikel (als Stellvertreter für Artikel und Koordinaten wird in der Implementierung dieselbe Datenstruktur verwendet). Die untere Reihe des Koordinatensystems ist für die Packstationen reserviert und beinhaltet keine Artikel (siehe Abbildung 5.1).

Platzierung der Roboter

Die Anzahl der Roboter kann ebenfalls frei festgelegt werden, sie darf die maximal zulässige Menge pro Koordinatensystem jedoch nicht überschreiten. Die Roboter werden beim Start des Programms auf den Koordinaten platziert (siehe Abbildung 5.1) - die Reihe der Packstationen steht dafür nicht zur Verfügung.

Bestellungen

Bei Programmstart wird die eingestellte Anzahl an Bestellungen generiert. Die Artikel werden hierbei zufällig ausgewählt und ihre Anzahl kann pro Bestellung variieren. Wenn der Vorgang abgeschlossen ist, werden die Bestellungen an die Roboter weitergeleitet.

Bewegung der Roboter zu den Artikeln

Nachdem ein Roboter eine Bestellung erhalten hat, beginnt er sich auf den ersten Artikel derselben zuzubewegen. Er kann sich hierbei nur auf die Koordinaten bewegen, die horizontal oder vertikal an ihn angrenzen und sich innerhalb des Koordinatensystems befinden.

Nachdem der Roboter beim Artikel angekommen ist, lädt er ihn auf und bewegt sich entweder zum nächsten Artikel oder zu einer Packstation.

Hat ein Roboter alle Bestellungen kommissioniert, entfernt er sich aus dem Koordinatensystem, um die Packstation, auf der er sich gerade befindet, für andere Roboter freizugeben.

Kollisionsvermeidung

Falls die Simulation mit mehr als einem Roboter abläuft, besteht die Möglichkeit, dass ein Roboter seine Position auf eine Koordinate ändern möchte, die bereits durch einen anderen Roboter belegt ist. In solchen Fällen führen die Roboter keinen Positionswechsel durch, da sie

im Vorfeld überprüfen, ob die entsprechende Koordinate frei ist. Falls dies nicht der Fall ist, warten sie eine gewisse Zeit oder führen ein Ausweichmanöver durch[1].

Simulationsende

Das Ende der Simulation ist erreicht, wenn alle Bestellungen kommissioniert worden sind und alle Roboter das Koordinatensystem verlassen haben.

5.3. Abbildung der Realität und Designentscheidungen

Die Roboter sind der eigentlich aktive Teil der Simulation. Sie bewegen sich in einer passiven Umgebung, die als Menge von Koordinaten implementiert ist und funktionieren dezentral: sie legen eigenständig ihre Ziele fest und bestimmen die Route, die sie nehmen. Wenn sie auf Kollisionskurs mit einem anderen Roboter sind, entscheiden sie über das weitere Vorgehen.

Nachdem ein Roboter eine Bestellung erhalten hat, beginnt er sich auf den ersten Artikel derselben zuzubewegen. In der Realität geschähe dies durch eine physikalische Positionsänderung; in der Simulation ändert der Roboter den Teil seines internen Zustands, der die aktuelle Position enthält, nach einer einstellbaren Zeit auf die neue Position.

Das Aufladen von Artikeln geschieht in der Simulation, indem die Koordinate, die den aktuellen Artikel innerhalb der Bestellung repräsentiert, aus deren Datenstruktur entfernt wird.

Um eine Kollisionsvermeidung zu realisieren, würden die Roboter in der Realität über Kameras verfügen, mit denen sie feststellen könnten, ob das Feld, auf das sie sich bewegen möchten, frei ist oder nicht. Die Kameras würden ein visuelles Bild einfangen, das Abbild einer zentralen Komponente der Wirklichkeit ist – der physikalischen Welt. In der Simulation wird diese Komponente durch einen Aktor realisiert, der die Positionen aller Roboter auf dem Koordinatensystem kennt und von diesen gefragt wird, ob eine angrenzende Position frei ist. Dies entspräche in der Realität der Auswertung eines Kamerabildes.

Diese Designentscheidung impliziert, dass die realen Roboter, die durch die Simulation abgebildet werden, komplett isoliert voneinander agieren und sich nicht absprechen können. Wenn dies möglich wäre, dann ließe sich die Kollisionsvermeidung optimieren, indem Protokolle entwickelt werden könnten, die ein intelligenteres Vorgehen der beteiligten Roboter ermöglichen würden.

[1] Die Ausweichmanöver werden durchgeführt, sollte die Koordinate auch beim zweiten Versuch blockiert sein und verhindern, dass es zu unendlich langen Wartezeiten kommt, falls sich etwa zwei Roboter frontal begegnen.

Das Abladen der eingesammelten Artikel einer Bestellung auf einer Packstation wird simuliert, indem die (nunmehr leere) Liste, welche die aktuelle Bestellung repräsentiert, aus der internen Datenstruktur entfernt wird.

Realisierung durch Aktoren

Das Aktoren-Paradigma hat sich für die Realisierung der Software als sehr geeignet herausgestellt, da sich die Eigenschaften des Aktoren-Primitives hervorragend für die Implementierung der Komponenten eignen.

Dass die Roboter einen eigenen State haben, der nicht nach außen verfügbar sein muss, und auf Ereignisse reagieren, entspricht einer intuitiven Modellierung, in Anbetracht dessen, dass ihr Programmablauf aufgrund der Tatsache häufig unterbrochen werden muss, dass die Roboter Aktionen wie Positionswechseln nicht unmittelbar durchführen können sollen. Diese Unterbrechung geschieht, indem der genutzte Thread in dieser Zeit für andere Zwecke zur Verfügung steht und der Roboter-Aktor anschließend durch den Erhalt einer Nachricht erneut aktiv wird.

Die Realisierung der Roboter-Aktoren mithilfe des `FSM`-Traits war hierbei eine Abwägungsfrage – man hätte darauf auch verzichten und stattdessen mit einem dynamischen Austauschen der `receive`-Funktion arbeiten können.

Würde man die Software etwa mit Techniken der Shared State Concurrency schreiben, so würden sich die Threads der Roboter für diesen Zeitraum blockieren. Außerdem würde die Software auf eine zentrale Datenstruktur zugreifen und mit Locking arbeiten müssen, um die Zugriffe zu synchronisieren. Beide Umstände würden die in vorigen Abschnitten beschriebenen Probleme mit der Korrektheit des Programms und negativen Folgen für die Effizienz mit sich bringen.

Da ein Aktor verwendet wird, ist der Zugriff auf die zentrale Komponente in der vorliegenden Implementierung bereits perfekt synchronisiert. Außerdem ist diese Lösung wesentlich einfacher skalierbar, wie im Abschnitt 5.5 beschrieben wird.

5.4. Implementierungsdetails

Dieser Abschnitt beschreibt die Funktionsweise der einzelnen Softwarekomponenten. Dessen Strukturierung wird in der Abbildung 5.2 visuell dargestellt.

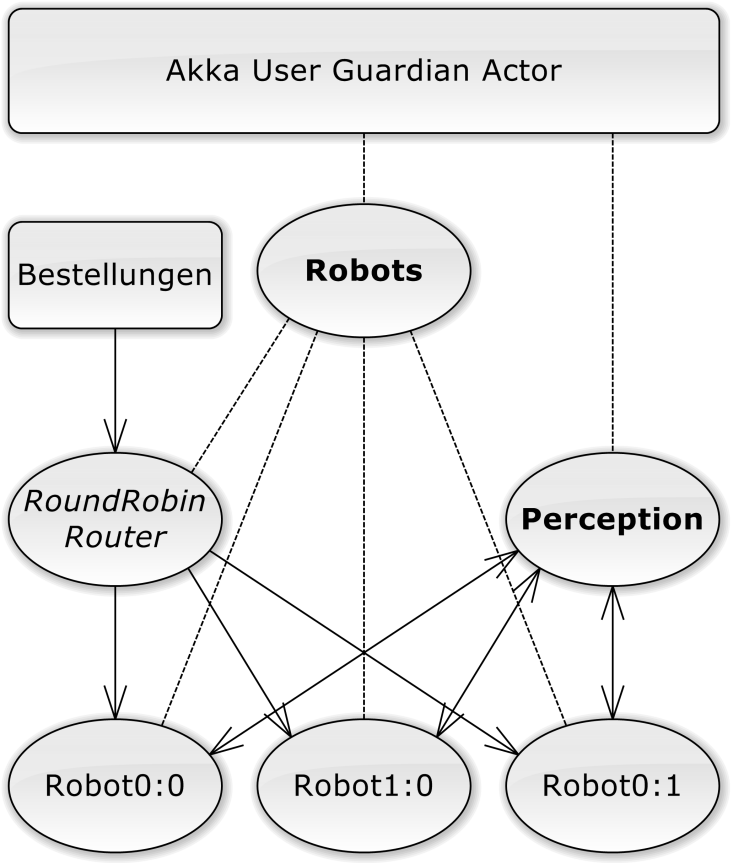

Abbildung 5.2.: Struktur der Softwarekomponenten: die Ellipsen stehen für Aktoren, die gestrichelten Linien für Abstammungsbeziehungen und die Pfeile für Kommunikationsrichtungen. Die Anzahl an Robot-Aktoren ist variabel.

5.4.1. Die Koordinaten / Artikel

Die Koordinaten werden durch dieselbe Datenstruktur wie die Artikel repräsentiert, so dass jede Koordinate, mit Ausnahme jener der letzten Zeile, auch ein Artikel ist:

```
1 object Coord {
2   /* erzeugt eine zufällige Koordinate des Bereichs auf dem die
3   Artikel liegen (ohne die letzte Zeile) */
4   def random() = {
5     Coord(Random.nextInt(<< Länge der X/Y-Achse >>),
6       Random.nextInt(<< Länge der X/Y-Achse >> - 1))
```

```
  }
}

case class Coord(val x: Int = 0, val y: Int = 0) {
  override def toString = {
    "" + x + ":" + y
  }
}
```

5.4.2. Komponente Robots

Das Programm startet, indem eine Instanz des Robots-Aktors angelegt wird. Während seiner Initialisierung erzeugt der Aktor die in der Konfiguration angegebene Anzahl an Robotern als seine Kinder:

```
import scala.collection.mutable.HashMap

def generateRobots(amount: Int) = {
  << werfe eine Exception, falls die Anzahl an Robots gleich 0
    oder nicht ausreichend Platz ist >>
  @tailrec
  def generateRobotsInt(amount: Int, lastCoord: Coord,
    robots: HashMap[ActorRef, Coord]): HashMap[ActorRef, Coord] = {

    if (amount == 0)
      return robots

    lastCoord match {
      case Coord(x, y) =>
        if (<< x kleiner als der letzte Wert der X-Achse >>) {
          val newCoord = Coord(x + 1, y)
          << lege einen neuen Robot an >>
          generateRobotsInt(amount - 1, newCoord,
            robots += (newRobot -> newCoord))
        } else {
          val newCoord = Coord(0, y + 1)
          << lege einen neuen Robot an >>
          generateRobotsInt(amount - 1, newCoord,
            robots += (newRobot -> newCoord))
        }
```

```
26      }
27    }
28    generateRobotsInt(amount, Coord(-1, 0), HashMap[ActorRef, Coord]())
29  }
30
31  val robots: HashMap[ActorRef, Coord] =
32    generateRobots(<< eingestellte Anzahl an Robotern >>)
```

Im Anschluss wird der `Perception`-Aktor mit der `actorOf`-Methode des `ActorSystems` angelegt und dabei der Wert `robots` im Konstruktor übergeben.

Danach erzeugt der `Robots`-Aktor einen `RoundRobinRouter` mit den angelegten Robotern als Routees und setzt ein Future ab, das die Bestellungen erzeugt und an den Router weiterleitet, damit diese gleichmäßig auf die einzelnen Roboter verteilt werden.

Der `Robots`-Aktor hält eine DeathWatch auf seinen internen Router. Da sich die `Robot`-Aktoren selbst terminieren, nachdem sie alle Bestellungen abgearbeitet haben, weiß der `Robots`-Aktor, dass die Simulation beendet ist, wenn er über die Terminierung des Routers unterrichtet wird. Auf dieses Ereignis reagiert der Aktor mit einer Log-Ausgabe[2].

5.4.3. Generieren der Bestellungen

Das Future, das die Bestellungen erzeugt, tut dies, indem es eine Liste mit Listen von Koordinaten anlegt. Die inneren Listen repräsentieren jeweils eine Bestellung, wobei jede Koordinate einen Artikel repräsentiert:

```
1  Future {
2    val orderAmount = << Gesamtzahl an Bestellungen >>
3
4    val orders = (1 to orderAmount) map ({ _ =>
5      val itemAmount = << Zufällig bestimmter Wert innerhalb der
6        eingestellten Schranken >>
7      (1 to itemAmount) map ({ _ =>
8        Coord.random()
9      }) toList
10   }) toList
11
12   orders.foreach(<< sende die Bestellung an den RoundRobinRouter
13     der Roboter >>)
14 }
```

[2] Er beendet das Programm deswegen nicht, weil sonst die Ausgaben der über den Trait `ActorLogging` zur Verfügung gestellten Logging-Aktoren abgeschnitten werden würden.

5.4.4. Komponente `Perception`

Der `Perception`-Aktor ist eine zentrale Komponente, die im gegenseitigen Austausch mit den einzelnen Robotern steht. Sie kennt die Position eines jeden Roboters auf dem Koordinatensystem und simuliert die Fähigkeit eines Roboters, seine visuellen Sensoren zu benutzen, um andere Roboter um sich herum zu erkennen und diesen auszuweichen.

Ein Roboter wird diese Komponente stets dazu benutzen, um festzustellen, ob eine auf der X- oder Y-Achse angrenzende Koordinate frei ist oder eine Ausweichposition zu ermitteln.

Initialisierung der Roboter

Beim Start des Aktors, sendet dieser zunächst eine Referenz von sich selbst an jeden Roboter.

Der `Perception`-Aktor kann den Robotern nicht bereits bei der Instantiierung mitgegeben werden, da er die Position eines jeden Roboters und dessen `ActorRef`s für die Identifizierung benötigt[3].

Kollisionsvermeidung

Möchte ein Roboter feststellen, ob eine Koordinate frei ist, sendet er eine Nachricht an den `Perception`-Aktor, welche die entsprechende Koordinate beinhaltet. Die Perception antwortet darauf mit einer Nachricht, die angibt, dass die Koordinate entweder frei oder blockiert ist. Falls das erstere der Fall ist, nimmt die Perception ferner an, dass der Roboter sich unverzüglich auf die freie Koordinate bewegen wird und aktualisiert seine Position mit der entsprechenden Koordinate. Ob eine Koordinate frei ist oder nicht, bestimmt die Perception, indem sie über die Positionen aller vorhandenen Roboter iteriert und diese auf Wertgleichheit mit der zu überprüfenden Koordinate testet.

Ermittlung einer Ausweichposition

Wenn ein Roboter ein Ausweichmanöver durchführen will, fragt er beim `Perception`-Aktor nach einer Koordinate, auf die er sich bewegen kann, die auf folgende Weise bestimmt wird:

```
1 def calculateBackoffCoord(): Option[Coord] = {
2   val currentPosition = << Koordinate, auf der sich der anfragende
3     Roboter zzt. befindet >>
4
```

[3] Für dieses Problem der Initialisierung bei wechselseitigen Abhängigkeiten sind jedoch auch andere Lösungen denkbar – so könnten die Roboter bspw. dem `Perception`-Aktor ihre Positionen mitteilen und die Initialisierungsreihenfolge so umgekehrt werden.

```
5   currentPosition match {
6     case Coord(x, y) =>
7       /* erzeugt eine Liste von möglichen Ausweichkoordinaten.
8       Die an die aktuelle Position angrenzenden Koordinaten werden
9       auf ihre Eignung hin überprüft. Erstes Kriterium:
10      die Koordinate muss sich im Koordinatensystem befinden.
11      Zweites Kriterium: die Position muss frei sein */
12      val candidates = List(Coord(x - 1, y), Coord(x + 1, y),
13        Coord(x, y - 1), Coord(x, y + 1)).view.filter({coord =>
14          coord.x >= 0 && coord.x <= gridDimension - 1 &&
15          coord.y >= 0 && coord.y <= gridDimension - 1
16        }).filter(isFree(_)).force
17
18      candidates match {
19        case Nil =>
20          None
21        case c =>
22        /* falls es mehrere Koordinaten gibt, wird per Zufall eine
23        ausgewählt */
24          Some(c(Random.nextInt(c.size)))
25      }
26  }
27 }
```

So wie bei der Rückmeldung über eine freie Koordinate nimmt die Perception nach der Bestimmung der Ausweichposition an, dass der Roboter sich unverzüglich auf diese bewegen wird und aktualisiert dessen Position innerhalb der internen Datenstruktur.

Entfernung von Robotern

Hat ein Roboter alle Bestellungen abgearbeitet und sich zu einer Packstation begeben, entfernt er sich aus dem Koordinatensystem. Er tut dies, indem er der Perception die entsprechende Nachricht sendet, woraufhin diese seine `ActorRef` aus der internen Datenstruktur entfernt.

5.4.5. Die Roboter

Die Roboter sind in Akka als Finite State Machines implementiert.

Datensätze

Innerhalb der einzelnen Zustände arbeiten die Roboter mit folgenden Datensätzen:

```
sealed trait Data

/* dieser Datensatz signalisiert, dass sich der Robot aktuell auf
kein Ziel zubewegt */
case object Standing extends Data

/* dieser Datensatz wird in den Zuständen MovingToCoord und
AvoidingCollision benutzt und beinhaltet die Zielkoordinate, auf die
sich der Robot zzt. zubewegt */
case class Moving(destination: Coord) extends Data
```

Zustände

Die Roboter können sich in den folgenden Zuständen befinden:

```
sealed trait State

// der Startzustand
case object Uninitialized extends State

/* in diesem Zustand ist ein Roboter, wenn die Initialisierung
abgeschlossen ist, aber noch keine Bestellung empfangen wurde */
case object Idle extends State

/* in diesem Zustand hat der Roboter ein Ziel, auf das er sich
zubewegt */
case object MovingToCoord extends State

/* in diesem Zustand ist der Roboter, wenn das Feld, auf das er
sich vorher bewegen wollte, durch einen anderen Robot blockiert
gewesen ist */
case object AvoidingCollision extends State
```

`Uninitialized` und `Idle`

Der Zustand `Uninitialized` ist der Startzustand. Er unterscheidet sich vom Zustand `Idle` nur in der Hinsicht, dass die Referenz zum `Perception`-Aktor noch nicht empfangen wurde.

Sobald dies geschehen ist, aber noch keine Bestellungen vorliegen, wechselt der Roboter nach `Idle`.

Im Zustand `Idle` ist der Roboter immer dann, wenn er den `Perception`-Aktor bereits empfangen, aber gerade kein Ziel hat.

Der verwendete Datensatz ist in beiden Zuständen das case objekt `Standing`, da sich der Roboter nicht bewegt.

Sobald eine Bestellung und der `Perception`-Aktor vorhanden ist, wechselt der Roboter in den Zustand `MovingToCoord`, verwendet eine Instanz des `Moving(destination: Coord)`-Datensatzes mit der Zielkoordinate des ersten Artikels der ersten Bestellung und beginnt sich darauf zuzubewegen.

`MovingToCoord`

In diesem Zustand ist der Roboter, wenn er sich auf das Ziel zubewegt, das in dem Datensatz `Moving(destination: Coord)` definiert ist. Innerhalb des Zustands reagiert der Roboter auf zwei Nachrichtentypen, die von dem `Perception`-Aktor als Antworten auf seine Anfragen gesendet werden:

Antworten von der Perception

- **Koordinate blockiert**
  Falls die angefragte Koordinate blockiert ist, ändert der Roboter seinen Zustand nach `AvoidingCollision` und weist den Scheduler an, die Perception nach einer gewissen Zeit zu fragen, ob die Koordinate frei geworden ist. Die zu wartende Zeit wird hierbei per Zufall bestimmt, ist jedoch durch eine obere Schranke limitiert, die sich in der Konfiguration festlegen lässt.

  Dieses Verhalten bewirkt, dass ein Roboter zunächst eine Zeit lang wartet, bevor er ein Ausweichmanöver durchführt.

- **Koordinate frei**
  Ist die Koordinate frei, ändert der Roboter seine interne Position auf diese und entscheidet, was als nächstes zu tun ist:

```
1 if (<< Ziel erreicht >>) {
2   if (<< aktuelle Bestellung erledigt >>) {
3     log.info("Station␣reached.␣Unloading␣commissioned␣items")
4
```

```
    << aktuelle Bestellung (leere Liste) löschen >>
    if (<< alle Bestellungen erledigt >>) {
      log.info("All orders completed. Moving away")

      perception ! RemoveMe
      context.stop(self)
      goToIdle
    } else /* noch Bestellungen vorhanden */ {
      log.info("Commissioning the next order")

      moveToNextItem()
    }
  } else /* aktuelle Bestellung nicht erledigt */ {
    log.info("Loading item " + << aktuelle Position >>)

    << entferne den aktuellen Artikel aus der Bestellung >>
    if (<< aktuelle Bestellung jetzt erledigt >>) {
      val station = Coord(<< aktuelle Position auf der X-Achse >>,
        << letzter Wert der Y-Achse >>)

      log.info("Current order completed, moving to station " +
        station)

      << bewege dich zur nächsten Koordinate auf dem Weg zur
        Packstation >>
      stay using Moving(station)
    } else /* aktuelle Bestellung noch nicht erledigt */ {
      moveToNextItem()
    }
  }
} else /* Ziel nicht erreicht */ {
  log.info("On position " + << aktuelle Position >>)

  << bewege dich zur nächsten Koordinate auf dem Weg zum Ziel >>
}
```

```
/* bewirkt, dass sich der Roboter auf den nächsten Artikel der
Bestellung zubewegt, indem das Ziel gesetzt und veranlasst wird,
dass der Roboter eine Nachricht bekommt */
def moveToNextItem() = {
```

```
5   if (<< nächste Koordinate auf dem Weg zum nächsten Artikel >> ==
6     << aktuelle Position >>) {
7     /* dieselbe Nachricht, welche die Perception sendet, wenn eine
8     Koordinate frei ist */
9     self ! Free(position)
10  } else {
11    /* fragt die Perception nach einiger Zeit, ob die nächste
12    Koordinate auf dem Weg zum nächsten Artikel frei ist */
13    schedule(movingDuration millis, IsFree(<< nächste Koordinate auf
14      dem Weg zum nächsten Artikel >>), perception)
15  }
16
17  if (stateName == MovingToCoord)
18    stay using Moving(<< nächster Artikel >>)
19  else
20    goto(MovingToCoord) using Moving(<< nächster Artikel >>)
21 }
```

Der Roboter bewegt sich auf ein Ziel zu, indem er beim Scheduler eine Nachricht registriert, die nach der in der Konfiguration eingestellten Zeitspanne für Positionswechsel an die Perception gesendet wird und diese fragt, ob die nächste Koordinate auf dem Weg zum Ziel frei ist. Der Erhalt der Antwort veranlasst den Roboter, die in diesem Abschnitt beschriebenen Aktionen durchzuführen.

AvoidingCollision

In diesen Zustand gerät der Roboter, wenn er im Zustand `MovingToCoord` eine Rückmeldung von der Perception erhalten hat, dass die Koordinate, auf die er seine Position wechseln wollte, durch einen anderen Roboter blockiert gewesen ist.

Innerhalb des Zustandes versteht der Roboter dieselben Nachrichten wie im Zustand `MovingToCoord`, führt jedoch andere Aktionen durch. Zusätzlich versteht er die Rückmeldung von der Perception, die ihm eine Koordinate für ein Ausweichmanöver mitteilt.

Erhält der Roboter von der Perception die Rückmeldung, dass die zuvor blockierte Position frei geworden ist, wechselt er zurück in den Zustand `MovingToCoord` und sendet sich die Rückmeldung erneut zu. Da er sich bei der Behandlung der Nachricht wieder im alten Zustand befinden wird, führt er daraufhin seinen Weg zum Ziel fort.

Erhält der Roboter stattdessen die Rückmeldung, dass die Koordinate immer noch blockiert ist, bleibt der Roboter in diesem Zustand und fordert von der Perception eine Ausweichposition an.

Wenn der Roboter die Rückmeldung zu seiner Anfrage nach einer Ausweichposition erhält und diese das `None`-Objekt enthält, weiß er, dass alle Koordinaten um ihn herum von anderen Robotern blockiert werden. Er wechselt daraufhin zurück in den Zustand `MovingToCoord` und versucht sich erneut auf sein Ziel zuzubewegen, indem er den Scheduler benutzt, um die Perception nach einer gewissen Zeit zu fragen, ob die nächste Koordinate auf dem Weg zum Ziel frei geworden ist. Die zu wartende Zeit wird hierbei per Zufall bestimmt, ist jedoch durch eine obere Schranke limitiert, die sich in der Konfiguration festlegen lässt.

Enthält die Rückmeldung von der Perception eine Ausweichposition, wechselt der Roboter seine Position entsprechend und verfährt auf dieselbe Weise, als hätte er ein `None`-Objekt erhalten, nur dass die Zeitspanne, die er wartet, über eine andere obere Schranke verfügt – sie lässt sich ebenfalls in der Konfiguration ändern.

Unbehandelte Nachrichten

Ein Roboter kann neue Bestellungen auch erhalten, während er sich bewegt: innerhalb des `whenUnhandled`-Blocks ist definiert, dass neue Bestellungen zu der internen Datenstruktur hinzugefügt werden.

Bei Zustandswechseln

Die Roboter erzeugen eine Log-Ausgabe, wenn sie in einen anderen Zustand als `Uninitialized` wechseln.

5.5. Möglichkeiten für die Weiterentwicklung

Die Bestellungen werden von der Simulation beim Start generiert und im Programmverlauf kommen keine weiteren Bestellungen hinzu. Der Grund für diese Designentscheidung ist, dass sich der Programmablauf so leichter vorhersagen lässt, als wenn die Bestellungen dynamisch erzeugt oder herausgegeben werden würden. Im realen System wäre dieser Aspekt jedoch anders.

Es findet keine Optimierung der Route für die Kommissionierung einer Bestellung statt, so dass die Artikel in der durch die Bestellung definierten Reihenfolge angefahren werden. Würde

die Route auf die Art optimiert, dass die Reihenfolge der Artikel anhand der Nähe zum Roboter bestimmt wird, ließe sich die Effizienz steigern.

Eine Bestellung wird exklusiv von einem Roboter kommissioniert. Eine noch weitergehende Möglichkeit, eine Steigerung der Effizienz zu erreichen, wäre, dass mehrere Roboter an einer Bestellung arbeiten können.

Die Verwendung von einem einzigen `Perception`-Aktor stellt ab einer gewissen Größe des Koordinatensystems einen Flaschenhals dar. Würden ggf. mehrere `Perception`-Aktoren verwendet werden, die das Koordinatensystem unter sich aufteilen, ließe sich der Flaschenhals beseitigen.

Die Kollisionsvermeidung kann optimiert werden, indem ein Roboter zunächst eine alternative Route plant, statt zu warten oder ein Ausweichmanöver durchzuführen, falls die nächste Koordinate auf dem Weg zum Ziel blockiert ist. So könnte er bspw. versuchen, sich dem aktuellen Ziel auf der entgegengesetzten Achse anzunähern.

A. Anhang

A.1. CD

Inhalt

Auf der enthaltenen CD befindet sich eine Kopie der vorliegenden Arbeit im PDF-Format, sowie eine ausführbare Version der Simulationssoftware, samt deren Sourcecode.

Informationen zur Verwendung der Software befinden sich in der Datei `README`.

Download

Die auf der CD befindliche Software, samt Quellcode, lässt sich auch über folgenden Link aus dem Internet herunterladen (Größe: ca. 8,7 MB):

http://peter.warnkross.de/bachelorarbeit/BA_Warnkross_Software.zip

Literaturverzeichnis

[Baeten 2005] Baeten, Jos C. M.: A brief history of process algebra. In: *Theoretical Computer Science* 335 (2005), Nr. 2–3, S. 131–146. – URL http://www.sciencedirect.com/science/article/pii/S0304397505000307. – ISSN 0304-3975

[Baker und Hewitt 1977] Baker, Henry ; Hewitt, Carl: Laws for Communicating Parallel Processes / MIT Artificial Intelligence Laboratory. URL http://dspace.mit.edu/handle/1721.1/41962, 1977. – Forschungsbericht

[Buth u. a. 1997] Buth, Bettina ; Kouvaras, Michel ; Peleska, Jan ; Shi, Hui: Deadlock Analysis for a Fault-Tolerant System. In: *Algebraic Methodology and Software Technology*, 1997, S. 60–74. – http://dx.doi.org/10.1007/BFb0000463

[Hewitt 1977] Hewitt, Carl: Viewing control structures as patterns of passing messages. In: *Artificial Intelligence* 8 (1977), Nr. 3, S. 323–364. – URL http://dx.doi.org/10.1016/0004-3702(77)90033-9. – ISSN 0004-3702

[Lee 2006] Lee, Edward A.: The Problem with Threads / EECS Department, University of California, Berkeley. URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html, 2006 (UCB/EECS-2006-1). – Forschungsbericht

[Moseley und Marks 2006] Moseley, Ben ; Marks, Peter: Out of the Tar Pit. In: *SOFTWARE PRACTICE ADVANCEMENT (SPA)*, 2006. – http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.93.8928

[Shavit und Touitou 1997] Shavit, Nir ; Touitou, Dan: Software transactional memory. In: *Distributed Computing* 10 (1997), Nr. 2, S. 99–116. – URL http://dx.doi.org/10.1007/s004460050028. – ISSN 0178-2770

[Tucker 2004] Tucker, Allen B. (Hrsg.): *Computer Science Handbook, Second Edition*. Chapman and Hall / CRC, 2004. – http://www.crcpress.com/product/isbn/9781584883609

[Typesafe Inc. 2013] *Akka Documentation 2.1.4*. Typesafe Inc. 2013. – http://doc.akka.io/docs/akka/2.1.4/

[Wing 2002] Wing, Jeannette M.: *FAQ on Pi-Calculus*. Carnegie Mellon University. 2002. – http://www.cs.cmu.edu/~wing/publications/Wing02a.pdf

[Wyatt 2013] Wyatt, Derek: *Akka Concurrency*. Artima Press. 2013. – http://www.artima.com/shop/akka_concurrency

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 10. Oktober 2013 Klaus Peter Warnkross