Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Sigurd Sippel

Distributive Steuerung humanoider Roboter auf Basis des Aktorenkonzepts

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Sigurd Sippel

Distributive Steuerung humanoider Roboter auf Basis des Aktorenkonzepts

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. Friedrich Esser
Zweitgutachter: Prof. Dr. Gunter Klemke

Eingereicht am: 07. Juni 2013

Sigurd Sippel

Thema der Arbeit

Distributive Steuerung humanoider Roboter auf Basis des Aktorenkonzepts

Stichworte

Aktorenmodell, Aktor, distributive Steuerungrung, humanoider Roboter, Interface Description Language, Message Passing, Nao, Metabetriebssystem, Pattern Matching, Serialisierung, Scala,Service-oriented architecture, Supervising

Kurzzusammenfassung

Diese Bachelorarbeit beschäftigt sich mit dem Entwurf einer Architektur, mit der eine distributive Steuerung eines humanoiden Roboters umgesetzt wird. Basierend auf dem Aktorenkonzepts und framebasierten Kommunikation von unveränderlichen und durch eine IDL definierte Nachrichten wird dem Nao Roboter kommuniziert. Dem werden aktuelle Metabetriebssysteme wie ROS gegenübergestellt.

Sigurd Sippel

Title of the paper

Distributed controlling of humanoid robots based on actor concept

Keywords

actor mode, actor, distributed controlling, humanoid robot, interface description language, message passing, Nao, meta operating system, pattern matching, serialization, Scala, service-oriented architecture, supervising

Abstract

This bachelor thesis is concerned with a architecture which realized controlling humanoid robots. Architecture is based on actor concept and frame based communication of immtuble and IDL controlled messages to realize a heterogeneous distributed system.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Aufbau und Struktur 3

2 Vergleich von Metabetriebssystemen **5**

2.1 Robot Operating System (ROS) 6

2.2 Middleware for Robotic Applications (MIRA) 7

2.3 Aldebaran Naoqi 8

2.4 Résumé 8

3 Funktionale Grundlagen **9**

3.1 Distributives System 9

3.1.1 Nebenläufigkeit und Parallelität 11

3.1.2 Kommunikationsmechanismen in Gegenüberstellung 11

3.1.3 Transparenz 14

3.1.4 Fehlertransparenz 15

3.1.5 Ausführungsgarantie 16

3.1.6 Shared state concurrency 17

3.2 Aktoren 18

3.2.1 Kommunikation in einem Aktorensystem 19

3.2.2 Unveränderbare Zustände 22

3.2.3 Dispatcher als Abstraktionselement 25

3.2.4 Delegation als Sicherheits- und Geschwindigkeitswerkzeug 25

3.2.5 Supervising im Aktorenkontext 26

3.2.6 Referenzierung von Aktoren 29

3.3 Kommunikation mit heterogenen Systemen 31

3.3.1 Serialisierung 31

3.3.2 Bindung an Zielsprachen 32

3.3.3 Framebasierte Kommunikation 35

4 Realisierung einer distributiven Steuerung 39
4.1 Vorstellung des Naoqi SDK 39
4.2 Anwendung: Steuerung durch Smartphone und Spielecontroller 41
4.3 Kommunikation mit dem Nao 42
4.4 Kommunikationssequenzen im Aktorensystem 49
4.5 Verteilung 60
4.6 Strukturierung auf Dateisystemebene 62

5 Fazit und Ausblick 65

Inhaltsverzeichnis der angehangenen CD 68

Codebeispielverzeichnis 69

Glossar 71

Abkürzungsverzeichnis 72

Literaturverzeichnis 73

Versicherung der Selbständigkeit 76

1 Einleitung

Der Roboter ist eine Maschine, die dazu entwickelt wurde dem Menschen bestimmte Arbeiten abzunehmen. Der Industrieroboter ist auf eine konkrete Aufgabe spezialisiert, wie zum Beispiel auf das Bearbeiten eines Werkstücks. Der Industrieroboter verwendet einen Greifarm, Sensoren und Werkzeuge um das Werkstück zu bearbeiten. Eine spezielle Steuerung, die genau für diese Aufgabe konzipiert ist, analysiert die gemessenen Daten der Sensoren und steuert den Greifarm.

Vom Industrieroboter grenzt sich der humanoide Roboter ab, da er dem Menschen nachempfunden ist. Ein humanoider Roboter hat Arme und Beine und einen beweglichen Kopf. Er ist in der Lage sich mit den Beinen wie ein Mensch im Raum zu bewegen und wie dieser mit den Armen nach Gegenständen zu greifen. Eingebaute Kameras lassen es zu, die aktuelle Position im Raum zu erfassen oder andere Gegenstände im Raum zu erkennen. Der Roboter hat keine Werkzeuge, die auf eine bestimmte Aufgabe beschränkt sind. Das Einsatzgebiet ist die Umgebung des Menschen (vgl. Chibani et al., 2012a, S. 883). Beispielsweise ist ein humanoider Roboter in der Lage dem Menschen zu Hause Aufgaben abnehmen, die der Mensch nicht mehr erledigen kann. Benötigt ein Mensch altersbedingt Medikamente, so kann der Roboter den Menschen überwachen und an die Einnahme der Medikamente erinnern.

Damit humanoide Roboter den Menschen unterstützen, müssen sie mit ihrer Umgebung interagieren (vgl. Chibani et al., 2012b, S. 807). Eine wichtige Anwendung besteht beispielsweise darin, dass der Mensch den Roboter aus der Ferne steuern kann. Dann werden neben einer einfachen Bewegungssteuerung auch Video- und Audiodaten der Umgebung benötigt. Ein Smartphone eignet sich insbesondere dann, wenn keine direkte Sicht zum Roboter möglich ist. Dann benötigt der Roboter Zugriff auf das lokale Netzwerk, um sich mit der Steuerung zu verbinden. Damit der Roboter mobil bleibt, ist eine drahtlose Anbindung an das Netzwerk sinnvoll.

Um rechenintensive Prozesse, wie Bildbearbeitung, auszulagern, sind zusätzliche Rechenkapazitäten erforderlich, die über ein lokales Netzwerk erreichbar sind (vgl. Chibani et al., 2012b, S. 808). Metabetriebssysteme für Roboter, wie das Robot Operating System (vgl. ROS, 2012), sind neben der Gerätesteuerung und Hardwareabstraktion darauf fokusiert, einen Roboter mit anderen Rechner kommunizieren zu lassen (s. 2 Vergleich von Metabetriebssystemen). Ein ein Metabetriebssystem erleichtert es Aufgaben auf andere Rechner auszulagern. Der Roboter ist in der Lage sich auf seine wesentlichen Funktionen, wie Bewegungskoordination, Aufnahme und Wiedergabe von Audio und Video, zu beschränken.

Die Lösung in dieser Bachelorarbeit basiert auf folgenden Open Source Frameworks:

- Aktorennsystem Akka (Typesafe): Ein in Scala geschriebenes Framework, dass konzeptionell an das aus Erlang bekannte Aktorenmodell angelehnt ist
- ZeroMQ (iMatix): Ein TCP-basiertes Kommunikationsframework
- Protocolbuffers (Google): Serialisiert und Deserialisiert Nachrichten in verschiedenen Programmiersprachen

Das Ziel dieser Bachelorarbeit ist es, am Beispiel eines Aldebaran Nao Roboters eine distributive Steuerung zu ermöglichen, sodass der Nao durch verschiedene Controller, wie ein Smartphone, eine Tastatur und ein Spielecontroller, vom Menschen gesteuert wird. Verwendet wird dafür das Aktorensystem Akka (s. 3.2 Aktoren). In einem Aktorensystem kommunizieren leichtgewichtige Prozesse - die Aktoren - miteinander. Kommuniziert wird ausschließlich über Nachrichten. Die Nachrichten, die direkt zum Nao versendet werden, werden durch ein TCP-basiertes Framework ZeroMQ - kurz ØMQ - transportiert (s. 3.3 Kommunikation mit heterogenen Systemen).

Auf Basis des Aktorensystems wurde eine Architektur entwickelt, die es ermöglicht verschiedene Applikationen parallel zu betreiben, die auf einen Nao zugreifen. Ein wesentliche Eigenschaft des Aktorenmodell ist die Fehlertoleranz (s. 3.1.4 Fehlertransparenz). Wenn Fehler auftreten, müssen die fehlerhaften Applikationen in der Lage sein zu reagieren, indem beispielsweise ein Neustart eingeleitet wird. Dabei dürfen andere Applikationen nicht beeinträchtigt werden.

Da auf dem Nao die hardwarenahe Programmiersprache C++ (vgl. Aldebaran, 2013a) verwendet wird und das Aktorensystem Akka in Scala (vgl. Typesafe, 2013, S. 1) geschrieben ist, müssen die Nachrichten unabhängig von der verwendeten Sprache verstanden werden. Für

die Beschreibung der Nachrichten wird Google Protocolbuffers - kurz Protobuf - verwendet (s. 3.3.2 Bindung an Zielsprachen). Dabei soll insbesondere auf etablierte Software wie Akka, ØMQ und Protobuf zurückgegriffen werden um eine stabile Software zu ermöglichen.

1.1 Aufbau und Struktur

Diese Arbeit gliedert sich in mehrere Bereiche: Zunächst werden in Kapitel 2 drei verschiedene Metabetriebssysteme auf konzeptioneller Ebene verglichen. Der Fokus liegt dabei auf der Kommunikation zwischen Roboter und anderen Rechnern.

In Kapitel 3 werden die funktionalen Grundlagen erläutert, welche die Basis für die Realisierung liefern. Zunächst werden grundlegende distributive Systeme vorgestellt und auf wichtige Eigenschaften und Kommunikationsmechanismen eingegangen. Auf den Kommunikationsmechanismen aufbauend werden Architekturen, die ein distributives System ermöglichen, vorgestellt und verglichen. Insbesondere der Umgang mit Fehlern wird vertieft, da dies einer der Kernpunkte des Aktorenmodells ist (vgl. Typesafe, 2013, S. 9).

Das Aktorenmodell wird anschließend anhand des Frameworks Akka beschrieben. Es wird gezeigt, wie Akka das Message Passing verwendet, um nebeläufig zu kommunizieren. Es wird vorgestellt, wie sich Zustände abbilden lassen. Außerdem wird auf Supervising, mit der Akka Fehlertoleranz realisiert, eingegangen. Abschließend wird die Referenzierung von Aktoren vorgestellt, die insbesondere für die Kommunikation mehrerer Aktorensysteme miteinander interessant wird.

Den Abschluss der funktionalen Grundlagen bildet die Kommunikation mit heterogenen Systemen. Für die Kommunikation mit dem Nao wird die Serialisierung des von Google entwickelten Frameworks Protobuf vorgestellt. Zum Abschluss wird der Transport der serialisierten Nachrichten mit ØMQ erläutert.

Für die Realisierung werden in Kapitel 4 das Naoqi SDK und dessen Restriktionen vorgestellt. Danach wird das Anwendungsbeispiel beschrieben. Detailliert wird auf die Erstellung der Nachrichten eingegangen und darauf, wie diese versendet und empfangen werden. Die Einbindung in die Gesamtarchitektur und deren Kommunikationsmechanismen werden ebenso

erläutert, wie die Aktoren mit dem Metabetriebssystem des Naos kommunizieren. Außerdem wird ein Überblick gegeben, wie die Realisierung auf Dateisystemebene strukturiert ist.

In Kapitel 5 werden Erfahrungen mit der verwendeten Software beschrieben und auf mögliche Verbesserungen eingegangen. Verbesserungen werden sowohl im Bezug auf die verwendeten Software-Bibliotheken, als auch in Hinblick auf die Architektur vorgestellt.

Abbildung 1.1: Aldebaran Nao (Aldebaran, 2013c)

2 Vergleich von Metabetriebssystemen

Roboter leben in einer Umgebung, in der mobile Geräte, wie Smartphones und andere Roboter, über das drahtloses Netzwerk kommunizieren, Internetdienste verwenden und selbst eigene Dienste anbieten. Insbesondere der Roboter ist dabei nicht nur Nutzer, sondern auch Dienstanbieter und muss sich dazu in diese heterogene Umgebung integrieren (vgl. Chibani et al., 2012b, S. 806f).

Auf Basis von Diensten, wie Laufen, Greifen oder Liefern von Kamerbildern kann ein Verhalten eines Roboters in einer Situation (z.B. den Inhalt eines Kühlschrankes überprüfen) abgebildet werden. Diese auf den Roboter zugeschnittenen Dienste werden durch das Metabetriebssystem zur Verfügung gestellt. Ein Metabetriebssystem wird auf einem Betriebssystem ausgeführt und verwendet dessen Gerätetreiber und Speicherverwaltung, sodass das Metabetriebssystem beispielsweise unabhängig von der verwendeten Prozessorarchitektur ist.

Insbesondere ein autonomes Handeln von Robotern ist jedoch noch ein Zukunftsszenario, in dem sich Roboter selbständig in einer unbekannten Umgebung zurechtfinden und selbst Entscheidungen treffen können. Sie müssen auch wissen, wie die Entscheidung umsetzen können (vgl. Chibani et al., 2012b, S. 808). Da Roboter dazu nicht selbständig genug handeln können, ist das Delegieren dieser Fähigkeiten zum Menschen nötig. Nur die Ausführung kann vom Roboter übernommen werden. Die Kommunikation mit dem Menschen, der z.B. durch sein Alter und seine Gesundheit unterschiedliche Fähigkeiten mit sich bringt oder auch benötigt, ist dabei ein großes ungelöstes Problemfeld (vgl. Chibani et al., 2012b, S. 808).

Ein Roboter benötigt ein Metabetriebssystem, welches grundlegende Funktionen des Roboters über eine API zur Verfügung stellt und eine Kommunikation über das Netzwerk mit anderen Rechnern ermöglicht. Drei Beispiele werden in diesem Abschnitt vorgestellt.

2.1 Robot Operating System (ROS)

Das Metabetriebssystem Robot Operating System (ROS) bietet eine Service-oriented Architecture (s. 3.1.2 Kommunikationsmechanismen in Gegenüberstellung) für Roboter (vgl. Chibani et al., 2012b, S. 807), die in ausgewählten, meist hardwarenahen Programmiersprachen implementiert ist. Auf Basis von Message Passing kommuniziert ROS mit verschiedenen Robotern, die in ROS über Adapter integriert werden. Dazu liefert ROS beispielsweise auch Adapter zu Webservices, wie RSS Feeds, sodass die Kommunikation auch mit Systemen außerhalb der Roboterwelt möglich ist. Es wird nicht nur auf einem lokalen Roboter oder zwischen mehreren Robotern des gleichen Typs zu kommuniziert. Jedem Roboter wird sein von ihm benötigtes spezielles Metabetriebssystem überlassen. Über das Netzwerk ist ROS mit den einzelnen Robotern verbunden. So können Applikationen auf einem Rechner ausgeführt werden und dabei der Roboter gesteuert werden. Sofern der Roboter grundlegende Dienste unterstützt (z.B. Laufen, Kamerabilder liefern), ist der Roboter in der Lage eine Applikation, wie die Überprüfung der Medikamente, auszuführen, die auf diesen Diensten basiert. Dazu muss ein Adapter in ROS für den Roboter implementiert sein und die Applikation die ROS-spezifischen Nachrichten verwenden. Der Roboter an sich muss dafür keine ROS Abhängigkeit besitzen (vgl. Quigley et al., 2009, S. 1ff).

ROS an sich ist zwar in einigen, meist hardwarenahen Sprachen geschrieben, jedoch bleibt die Spezifikation auf Level der Nachrichten. Die Nachrichten werden durch eine eigene IDL spezifiziert (s. 3.3.2 Bindung an Zielsprachen). ROS verwendet einen Namensdienst, der als Master und als Slave zur Verüfung gestellt wird, um im Falle eines Ausfalls des Masters noch einen Slave zur Verfügung stellen zu können. Beide basieren auf dem HTTP Protokoll und XML Nachrichten. Daher ist eine eigene Implementierung auf Basis der spezifizierten Nachrichten möglich. Die verschiedenen Dienste in ROS, wie die eingebunden Roboter, sind über eine eine URI Referenzierung (s. 3.2.6 Referenzierung von Aktoren) zugänglich. Message Passing und Referenzierung ermöglichen eine hohe Modularität und Wiederverwendbarkeit. Module können auf mehreren Nodes instanziiert werden und dadurch skalieren. Welche Nodes welche Services anbieten wird durch eine XML Konfiguration definiert (vgl. Quigley et al., 2009, S. 1ff).

2.2 Middleware for Robotic Applications (MIRA)

In Anlehnung an ROS gibt es ein weiteres, von der Universität Ilmenau entwickeltes, Metabetriebssystem MIRA. Bei MIRA liegt der Fokus ebenfalls auf der distributiven Sicht, um Roboter aus der lokalen, isolierten Welt herauszuheben und mit ihrer Umgebung kommunizieren zu lassen. Grundlegend ist ROS als Referenz für das Design von MIRA zu sehen, jedoch gibt es markante Unterschiede. MIRA ist im Gegensatz zu ROS vollständig dezentral, da kein zentraler Namensdienst benötigt wird. Damit ist ein Single Point of Failure vermeidbar. MIRA verfügt über verschiedene Serialisierungstechniken für Nachrichten (JSON, XML), verzichtet jedoch explizit auf eine IDL. Als Hintergrund wird das Verlangen nach objektorientierten Techniken wie Polymorphie gesehen, die der funktionalen IDL von ROS gegenüberstehen. Weiterhin werden Geschwindigkeitsvorteile gegenüber ROS gesehen, da Daten innerhalb eines Prozesses auch direkt (durch Synchronisierung) verteilt werden können (vgl. Einhorn et al., 2012, S. 2591ff).

Neben Anstrengungen in hoher Performanz, intuitiver Funktionen und Typsicherheit stellt MIRA die verschiedenen Einheiten im distributiven System in den Mittelpunkt. Anstatt wie ROS diese Node zu nennen, bezeichnet MIRA diese als Unit. Units kommunizieren durch Message Passung oder RPC und können in verschiedenen Prozessen ausgeführt werden und miteinander kommunizieren. Units können zur Laufzeit ihren Ort wechseln, ohne dabei in den Code eingreifen zu müssen. Jede Unit erhält eine Nachrichtenschlange namens Channel. Units und Channels ähneln in Ansätzen den in Rahmen dieser Arbeit vorgestellten Aktoren und Mailboxen (s. 3.2 Aktoren). Da der Datenzugriff auf Synchronisierung basiert, wird versucht mit einem slotbasierten Verfahren den Zugriff möglichst effizient halten. Für den Zugriff auf eine Variable werden (Zeit)Slots vergeben, in denen der Zugriff erfolgen muss (vgl. Einhorn et al., 2012, S. 2593f).

2.3 Aldebaran Naoqi

Zwar können ROS und MIRA direkt auf einem Roboter ausgeführt werden, jedoch sind diese für die Verteilung im Netzwerk gedacht. Es gibt es auch Metabetriebssysteme, die darauf konzipiert sind ausschließlich auf einem lokalen Roboter ausgeführt zu werden und nur rudimentär anderen Rechnern kommunizieren können. Dazu gehört das Metabetriebssystem Naoqi von Aldebaran. Auf Basis einer speziell angepassten Gentoo Distribution namens OpenNAO wird Naoqi ausgeführt. Naoqi bietet eine API, die grundlegende Dienste des Nao, wie Bewegung, Sprachausgabe oder Videoübertragung, bereitstellt. Diese API ist spezifisch für die eingesetzten Geräte. Es gibt es die Möglichkeit über RPC die Naoqi API von einem anderen Computer zu verwenden (s. 4.1 Vorstellung des Naoqi SDK). Es existieren weder eine IDL noch Messages. Es wird eine spezielle Entwicklungsumgebung und eine Virtualisierung angeboten, die es ermöglichen Naoqi auf einem Computer auszuführen und zu testen, ohne dabei auf einen zur Verfügung stehenden Nao angewiesen zu sein. Dabei werden jedoch Dienste, wie Videoübertragung, nicht angeboten, da kein Zugriff auf die Kamera des Naos besteht (vgl. Aldebaran, 2013b).

2.4 Résumé

Auffallend ist, dass die hier vorgestellten Metabetriebssysteme sehr hardwarenah angelegt sind. Für die Entwicklung von grundlegenden Diensten, wie Laufen oder Treppe steigen, die sehr zeitkritisch sind, werden hardwarenahe Implementierungen verwendet. Denn zeitkritische Anwendungen sind in hardwarenahen Sprachen einfacher zu implementieren (vgl. Sarabia et al., 2011, S. 670).

Die zweite Auffälligkeit ist das Verständnis eines distributiven Systems. ROS setzt auf einen zentralen Namensdienst, MIRA setzt verstärkt Synchronisierung ein. Jedoch bleibt das Thema Fehlertoleranz (s. 3.1.4 Fehlertransparenz) in den Architekturen ein Nischenthema. Dabei ist auch RPC zu nennen, welches durch das Request-reply Protokoll wiederum an Gewohnheiten der lokalen Programmierung erinnert. Interessant ist auch der starke Einsatz von XML, welches als sehr flexibles, sperriges und rechenintensives Format gilt (s. 3.3.1 Serialisierung).

3 Funktionale Grundlagen

Die funktionalen Grundlagen sind in drei Abschnitte eingeteilt: Zunächst werden wichtige Eigenschaften distributiver Systeme und verschiedene distributiver Architekturmuster vorgestellt. Insbesondere die Kommunikationsmechanismen werden erläutert, die im Hinblick auf ihre Fehlertoleranz ein distributives System beeinflussen. Darauf Aufbauend wird das Aktorenmodell als Architekturmuster dessen Implementierung Akka vorgestellt. Abschließend wird auf Serialisierung eingegangen, am Beispiel von IDLs die Bindung an Zielsprachen erläutert und eine framebasierte Kommunikation für das Verteilen der Serialisierungen im Netzwerk gezeigt.

3.1 Distributives System

In einer Welt, in der technische Infrastrukturen in ihrer Komplexität kontinuierlich wachsen, werden parallele und vor allem nebenläufige (s. 3.1.1 Nebenläufigkeit und Parallelität) Systeme immer mehr zum Normalfall. Denn wenn verschiedene Systeme zur gleichen Zeit ausgeführt werden und zusammen Aufgaben bewältigen sollen, müssen sie miteinander kommunizieren. Systeme dieser Art benötigen auch die Freiheit ungebunden von Hardware zu arbeiten, sie verlangen ein Netzwerk, in dem sie mit anderen Systemen kommunizieren können und damit ein distributives System darstellen (vgl. Colouris et al., 2012, S. 11).

Systeme können sich auf sehr vielfältige und damit auch unterschiedliche Basistechnologien verteilen. Verschiedene Kommunikationsmechanismen (s. 3.1.2 Kommunikationsmechanismen in Gegenüberstellung), außerdem verschiedene Hardware, Betriebssysteme, Sprachen und Implementationen einer Spezifikation können miteinander zusammenarbeiten. ROS ist Beispiel dafür, denn es ist in verschiedenen Programmiersprachen implementiert, verwendet Message Passing und RPCs zur Kommunikation und kann auf verschiedenen Beriebsystemen, wie Linux,

Windows und Mac OS, ausgeführt werden. Diese Unterschiedlichkeit wird als Heterogenität bezeichnet (vgl. Colouris et al., 2012, S. 32ff).

Je distributiver ein System ist, umso mehr Schnittstellen werden nach außen zu anderen Knoten benötigt. Sind diese Schnittstellen offengelegt, deren Kommunikationsmechanismen einheitlich und die Heterogenität für die Kommunikation kein Hindernis, dann kann das distributive System als offen bezeichnet werden (vgl. Colouris et al., 2012, S. 34).

Zwei weitere wichtige Eigenschaften sind die Kohärenz und die Autonomie. Erst wenn eine oder mehrere Applikationen verteilt ein Ganzes ergeben, jedoch gleichzeitig die einzelnen Module selbstständig agieren können, kann ein System als kohärentes (zusammengehöriges) und autonomes (eigenständiges) distributives System bezeichnet werden. Autonomie kann auch bedeuten heterogen zu sein, indem ein System im distributiven System beispielsweise eine andere Programmiersprache verwendet. Damit ein distributives System kohärent sein kann, ist es eine Lösung, es möglichst zu öffnen, damit die Komponenten möglicht eng zusammenarbeiten können. Insbesondere der Einsatz von Frameworks, wie Apache Camel, dass verschiedenen Protokoll, wie HTTP, mit einer einfachen Schnittstelle verwendbar macht, hilft verschiedene Systeme zusammen zu führen. Ebenso hilft Message Passing, da es dabei nur um Werte geht, die verstanden werden müssen; diese können sprachunabhängig in eine beliebige Zielsprache überführt werden (s. 3.3.2 Bindung an Zielsprachen).

Im Gegensatz zur klassischen Applikation, die auf einem Rechner geöffnet und, wenn diese nicht mehr benötigt wird, geschlossen wird, ist ein verteiltes System in Regel kontinuierlich verfügbar, wenn auch einzelne Komponenten nicht immer verfügbar sind (vgl. Tanenbaum, 2003, S. 18). Mit Fehlern, wie ein anderes System nicht erreichen zu können, muss ein einzelnes System umgehen können (s. 3.1.4 Fehlertransparenz).

Im Zuge der größer werdenden Komplexität erhöhen sich auch die Anforderungen an ein distributives System. Dabei stellen sich Anforderung an die Handhabbarkeit, Erweiterbarkeit und Kosteneffizienz. Erweiterbarkeit bedeutet, dass weitere Computer verwendet werden können, um die Leistungsfähigkeit zu erhöhen (vgl. Tanenbaum, 2003, S. 18). Größere Komplexität bedeutet auch, dass die Heterogenität zunimmt und dadurch wartebare, offene Schnittstellen wichtig werden, die die Kosten für Wartung unter Kontrolle halten (vgl. Colouris et al., 2012, S. 56ff).

3.1.1 Nebenläufigkeit und Parallelität

Wenn es zwei Ereignisse a und b gibt, die in einem Prozess hintereinander ablaufen, dann gilt die Relation $a \rightarrow b$. Wird eine Nachricht von einem Prozess gesendet (Ereignis a) und von einem anderen Prozess empfangen (Ereignis b), gilt dieselbe Relation. Diese Relation ist transitiv, d.h. es gilt $(a \rightarrow b \land b \rightarrow c) \leftrightarrow a \rightarrow c$. Diese Relation nennt sich happend-before Relation. Wenn es zwei Ereignisse gibt, für die keine happend-before Relation existiert, so sind diese nebenläufig. Das bedeutet, dass sie potenziell zur gleichen Zeit (parallel) ablaufen können, jedoch darüber zur Laufzeit keine Aussage getroffen werden kann (vgl. Colouris et al., 2012, S. 623f, vgl. Tanenbaum, 2003 S. 289).

3.1.2 Kommunikationsmechanismen in Gegenüberstellung

Kommunikationsmechanismen sind entscheidend für die Kommunikation in einem distributiven System. Drei wichtige werden Kommunikationsmechanismen hier vorgestellt. Das Request-reply Protokoll stellt eine Möglichkeit dar, eine Client/Server Architektur zu realisieren. Eine Anfragenachricht wird vom Client an den Server geschickt, dort wird ein dafür vorgesehener Code ausgeführt und eine Antwortnachricht zurückgeschickt. Dies geschieht aus Sicht des Client blockierend. Die Anfragenachricht gitl erst als erfolgreich, wenn die Antwortnachricht erhalten wurde. Sollte eine Nachricht nicht ankommen, so bleibt dem Client nur die Möglichkeit die Nachricht erneut zu senden, die eventuell auch ein zweites Mal ausgeführt wird. Auch im Fall von TCP als Basistechnologie kann bei ausgefallener Antwortnachricht die TCP Acknowledge Nachricht keinen Vorteil bringen, denn in jedem Fall muss der gesamte Prozess wiederholt werden (vgl. Colouris et al., 2012, S. 203ff). Der Client hat keinerlei Anhaltspunkt, ob vor, während oder nach der Verarbeitung ein Fehler passiert ist (s. 3.1.5 Ausführungsgarantie).

Darauf aufbauend ist der Remote Procedure Call (RPC) zu nennen, der, definiert durch eine Interface Description Language (IDL), einen entfernen Prozeduraufruf ermöglicht. Die IDL enthält eine sprachunabhängige Definition von Nachrichten, die sich basierend auf Basistypen aufbauen lassen (s. Bindung an Zielsprachen 3.3.2). D.h. alle Schnittstellen, die distributiv verwendet werden sollen, müssen mit einer IDL definiert sein. Das klassische Beispiel eines Architekturmusters auf Basis von RPC ist Corba (Common Object Request Broker Architecture). Architekturmuster sind allgemeingehaltene Architekturen, für die eine Vielzahl von

Implementationen vorhanden sind. Corba hat eine eigene IDL und für verschiedene Sprachen Adapter, die es ermöglichen ein entferntes Objekt zu einer dem Client bekannten Schnittstelle zu erzeugen. Auf Basis von Request-reply wird der Aufruf zum Server gesendet, dort wieder in einen Methodenaufruf umgewandelt und ausgeführt (s. Serialisierung 3.3.1). Erreicht wird dadurch eine sprachenunabhängige Benutzung eines entfernten Objektes, dessen Implementationsdetails verborgen sind. Gleichzeitig wird jedoch in Kauf genommen, dass es verschiedene Referenzen auf verschiedenen Systemen geben kann, die sich in einem unterschiedlichen Zustand befinden. Es gibt keine global gültige Referenz (vgl. Colouris et al., 2012, S. 2011ff).

Java geht mit Remote Method Innovation (RMI) einen anderen Weg. RMI basiert auf auf einem RPC, jedoch setzt es eine reine Java Implementation eines distributiven Systems voraus. Die IDL ist dabei das javaeigene Interface, welches von einem Remote Interface erben muss und damit das Fehlerhandling durch RemoteExceptions mit ins Boot nimmt. Alle Objekte müssen die javaeigene Serialisierung verwenden, die auch nur von Java verstanden wird. Proxyobjekte können so automatisch erstellt werden. Referenzen sind bei RMI global gültig. Dazu kommen ein integrierter Namensdienst, distributive Garbage Collection und ein distributiver Classloader, sodass die Möglichkeit geben ist, verteilte Aufrufe zu tätigen, ohne dabei sich um die technische Realisierung zu kümmern (vgl. Colouris et al., 2012, S. 230ff). Besonders deutlich wird das durch die Unterstützung von Mobile Code, der beispielhaft genutzt wird um Java Bytecode vom Server in einem Java Applet im Browser des Clients nachzuladen und dort auszuführen. Es eröffnen sich dabei nicht nur neue Möglichkeiten, sondern auch Fragen nach der Sicherheit. Mobile Code ermöglicht es auch schädlichen Code auf einem Rechner einzuschleusen. Außerdem stellt sich die Frage wie gut die Kapselung der Komponenten dann noch gewährleistet ist, wenn jeglicher Code überall ausgeführt werden kann (vgl. Colouris et al., 2012, S. 66ff).

Aus einer anderen Perspektive blickt die Service-oriented architecture (SOA) auf die Welt. Der Kern von SOA ist es, Dienste auf Basis asynchroner Nachrichten anzubieten, ohne dabei auf ein Request-reply Protokoll angewiesen zu sein. SOA ist ein Architekturmuster, welches in der Idee völlig sprachenunabhängig ist, aber anders als Corba nicht aus der Objektperspektive, sondern auf der Dienstperspektive arbeitet. Architekturmuster sind abstrakt und ermöglichen verschiedene Impelementierungen, d.h. ein Dienst kann auf verschiedene Arten realisiert werden (vgl. Colouris et al., 2012, S. 429f). Beispielsweise Corba zwingt den Nutzer von Corba, Corba als Abhängigkeit in jedes einzelne System im distributiven System zu integrieren, um u.a. Transaktionen, Authentifizierung und Events zu ermöglichen.

Eine SOA ermöglicht eine möglichst geringe technische Kopplung, indem Nachrichten verwendet werden. Dabei stellen sich die Fragen nach Referenzen nicht mehr, da ausschließlich der Wert (der Inhalt der Nachricht) verwendet wird. Nachrichten sind unveränderbar. Globale Referenzen wie in RMI sind dabei gar nicht mehr von Interesse, da eine Nachricht beliebig verteilt werden kann. Die Asynchronität ermöglicht, dass ein Teil eines distributives Systems nicht auf einen anderen Teil warten muss. Synchrone Request-reply Aufrufe führen dagegen zu einem blockierenden Zustand, in dem die abhängigen Prozesse nicht parallel arbeiten können (vgl. Colouris et al., 2012, S. 414f).

Eine mögliche Implementierung einer SOA ist ein Webservice auf Basis von XML Nachrichten durch REST. Auch hier findet sich eine IDL - die Webservice Description Language (WSDL) - wieder. Die Besonderheit bei XML ist, dass dadurch eine sehr lose Kopplung erreicht wird, da XML keine einheitliche Softwarelösung zum Verarbeiten benötigt. XML ist ein offener Standard und ist nicht spezifisch für eine Programmiersprache. Der Zugriff auf die in XML strukturierten Daten bleibt in der Hand des jeweiligen Systems. Besonders ist auch die mögliche Einbindung eines Verweises (URI) auf die passende WSDL, sodass eine Nachricht identifiziert werden kann. Auch wenn das Request-reply Protokoll keine Grundlage für SOA darstellt, kann es trotdzem benutzt werden, indem eine Request- und eine Reply-Nachricht versendet werden. Dies äußert sich auch in den XML Nachrichten, deren Tags signalisieren können, ob es sich um eine Anfrage oder um eine Antwort handelt. (vgl. Colouris et al., 2012, S. 400ff).

Ein Architekturmuster, das auch als SOA verstanden werden kann, ist das Aktorenmodell (s. 3.2 Aktoren). Auf Basis von Nachrichten können leichtgewichtige Prozesse Nachrichten austauschen und sind dabei an kein Request-reply Pattern gebunden. In der Implementierung Akka sind Nachrichten auf der Basis der Java Serialisierung, Protobuf oder einer eigenen Implementation möglich.

Request-reply ist grundlegend und einfach aufgebaut, mit RPC wird auf einem höheren Abstraktionslevel zu gearbeitet. RPC ist von der Idee her nicht objektorientiert, es kann jedoch für die Umsetzung von der Objektorientierung verwendet werden, wie in RMI. Man kann Java RMI als Versuch einer kompromisslosen Umsetzung der Objektorientierung im distributiven System verstehen. Kritisch kann man die Sprachbindung an Java sehen, auch wenn Java anstrebt plattformunabhängig zu sein. SOA überwindet viele Probleme von RPC, bleibt allerdings sehr abstrakt und fällt in Praxisisbeispielen wie REST auf Paradigmen wie Request-reply zurück. XML wirkt sperrig, sorgt allerdings für lose Kopplung und nimmt endgültigen Abstand von von globalen Referenzen im distributiven System.

3.1.3 Transparenz

In distributiven Systemen sind eine Reihe von Besonderheiten zu beachten. Beispielsweise kann ein Webservice von einem Server auf einen anderen umziehen, dadurch ändert sich die IP (der Ort), trotzdem muss der Webservice erreichbar bleiben. Es gibt im Grunde zwei Ansätze damit umzugehen: Zum einem, dass man das Problem zum Benutzer delegiert. Jede Komponente müsste sich darum kümmern, auf welchem Server sich die gerade benötigten Module befinden. Zum anderen das distributive System bietet eine Abstraktion an, den Ort zu verbergen (beispielsweise durch einen Namensdienst wie DNS oder DHCP realisierbar). Je mehr verborgen ist, umso transparenter ist das System. Transparenz ist ein wichtiges Instrument um ein distributives System wartbar zu halten und vor Fehlern zu schützen, indem die Komplexität reduziert wird. Positionstransparenz beschreibt die beispielhaft angesprochene Problematik. Der Lösungsansatz von beispielswise DNS ist es eine Abstraktion zwischen logischen URI und den durch die technische Infrastruktur bedingten IP Adressen einzuführen (vgl. Tanenbaum, 2003, S. 21ff).

Transparenzen sind wichtig, jedoch sind diese kein Allheilmittel, denn je nach Anwendungsgebiet können Transparenzen auch Funktionalitäten erschweren. Befinden sich beispielsweise zwei Dienste in einem lokalen Netz, ist die Anbindung sehr gut. Sind die Dienste weit entfernt, kann die Entfernung eine Anwendung verlangsamen. Durch die Positionstransparenz ist keine Aussage mehr darüber möglich, ob der Dienst seinen Ort geändert hat. Ist bekannt, dass der Dienst langsamer ist, kann darauf reagiert werden (vgl. Tanenbaum, 2003, S. 21ff).

Von Bedeutung ist Nebenläufigkeitstransparenz, mit der verborgen wird, dass mehrere Benutzer gleichzeitig eine Ressource benutzen. Ein typisches Konzept ist das Sperren von kritischen Zuständen (s. 3.1.6 Shared state concurrency), das dem Aktorenkonzept direkt gegenüber steht (s. 3.2 Aktoren). Im Aktorenkonzept können Nachrichten jederzeit gesendet werden. Die Nachrichten werden über Nachrichtenschlangen an die Aktoren weitergeleitet. Persistenztransparenz beschreibt das Verbergen der Speicherung der Daten. Dabei ist sowohl das Medium auf dem gespeichert wird (nicht flüchtige Medien, wie Festplatten, oder auch flüchtige, wie der Arbeitsspeicher) zu verbergen als auch die Form (beispielsweise objektorientiert oder relational). Möglich ist der Einsatz einer objektrelationalen Abbildung (z.B. Hibernate) um Objekte in einer Datenbank zu speichern (vgl. Tanenbaum, 2003, S. 21ff).

3.1.4 Fehlertransparenz

Charakteristisch für ein distributives System ist, dass Teile davon ausfallen können. Dabei ist es wichtig, dass das System darauf angemessen reagieren kann, ohne das Leistung des Gesamtsystems darunter maßgeblich leidet. Außerdem müssen Gegenmaßnahmen eingeleitet werden, die eine Wiederherstellung des ausgefallenen Teils ermöglichen. Die Leistungsfähigkeit des Gesamtsystems lässt sich an mehreren Gesichtspunkten erkennen. Zum einem bezeichnet die Verfügbarkeit die Wahrscheinlichkeit, dass ein System zu einem bestimmten Zeitpunkt für eine gewünschte Operation zur Verfügung steht. Abgrenzend zur Verfügbarkeit bezeichnet die Zuverlässigkeit, inwieweit ein System in einem Zeitraum fehlerfrei und verfügbar ist. Ein System mit vielen, sehr kurzen Ausfällen ist hoch verfügbar, jedoch unzuverlässig. Sicherheit bezeichnet, dass ein System auch im Fehlerfall keinen Schaden anrichtet, d.h. angemessen auf den Fehler reagiert. Abschließend ist die Wartbarkeit zu nennen, die den Aufwand bezeichnet, der nötig ist, um einen Fehler zu beheben. Alle vier Aspekte stehen in ständiger Wechselwirkung zueinander, beispielsweise um zuverlässig zu arbeiten, muss man auch sicher arbeiten (vgl. Tanenbaum, 2003, S. 410f).

Der Fehler ist definiert als Ursache eines Ausfalls. Wenn ein System ausfällt, muss der Fehler gefunden werden, um einen weiteren Ausfall zu vermeiden, denn Fehler passieren im Laufe des Betriebs. Entscheidend ist mit dem Fehler umgehen zu können. Damit ein System zuverlässig sein kann, ist eine Fehlertoleranz nötig. Im Gegensatz zu einer Transparenz, die etwas verbergen möchte, ist einer Toleranz darauf eingestellt, dass etwas passiert, und kann darauf reagieren. Reagieren kann im Bestfall bedeuten, den Fehler zu verhindern. Es kann allerdings auch bedeuten, den Fehler vorauszusehen. Dabei muss man zwischen drei Arten von Fehlern unterscheiden: zum einem der vorübergehende Fehler, der schwer zu reproduzieren ist, der periodische Fehler, der immer wieder auftritt und den permanenten Fehler, der nicht zeitlich gebunden ist (vgl. Tanenbaum, 2003, S. 410f).

Ein möglicher permanenter Fehler ist der Absturzfehler, bei dem zuvor alles funktioniert hat und durch den dann der Dienst seine Arbeit verweigert und nur noch durch ein Neustart der Fehler beseitigt werden kann. Ein Auslassungsfehler tritt z.B. bei einem Fehler im Netzwerk auf, die nächste Anfrage kann dabei wieder ordnungsgemäß funktionieren. Ein Timingfehler kann auftreten, wenn im erwarteten Zeitintervall keine Antwort kommt, was auf einen zu hohen Rechenaufwand oder auf Netzwerkprobleme zurückgeführt werden kann. Ist die Antwort falsch, so ist ein Wertfehler aufgetreten. In einem Zustandsautomat könnte der

Zustandsübergang ein anderer sein als der erwartete, dann ist ein Zustandsübergangsfehler eingetreten. Abschließend ist noch der byzantinische oder zufällige Fehler zu nennen, dessen Ursache nicht bestimmbar ist. Byzantinische Fehler tauchen insbesondere im Zusammenhang mit Fremdsystemen auf. Fremdsysteme mit nicht vollständig bekannter Funktionalität weichen von der eigenen Erwartungshaltung ab, sodass es zu einem byzantinischen Fehler kommt (vgl. Tanenbaum, 2003, S. 412ff).

Die Mächtigkeit eines distributiven Systems zeigt sich in der Transparenz, die die vorhandenen Unterschiede kapselt und andere Teile des Systems unberührt lässt. Eine Middleware ist eine Möglichkeit diese Transparenz zu erreichen, indem eine Schicht (Layer) zwischen Betriebssystem und der Applikation gesetzt wird, die eine einheitliche Kommunikationsschnittstelle darstellt. Ein vollwertiges Betriebsystem ist dabei jedoch auch in der Pflicht Zugriffstransparenz gegenüber dem Dateisystem zu bieten. Allerdings muss eine Middleware nicht jede nur mögliche Transparenz bieten. Die Idee dabei ist die Heterogenität, die im verteilten System auftritt, zu verbergen. RMI und Corba stellen klassische Middlewareimplementationen dar. Grundsätzlich gilt: wenn eine Middleware vorhanden ist, die jeder Knoten mit einem Adapter einbinden kann, so kann man im verteilten System eine Schnittstelle anbieten, die über die Middleware angesprochen werden kann. Diese Schnittstelle kann von einem anderen Knoten verwendet werden. Eine Middleware ist dabei allerdings losgelöst von der fachlichen Anwendung, es ist eine vielfältig einsetzbare Software. Beispielhaft sind Laufzeitumgebungen zu nennen, die Bytecode je nach Hardware und Betriebsystem ausführen können. D.h. wenn eine Laufzeitumgebung für einen Rechner existiert, kann jeder Bytecode für diese Laufzeitung auf diesem Rechner ausgeführt werden. So ist das verteilte System nicht sprachenunabhängig, jedoch ist die Sprache unabhängig von der Hardware (vgl. Colouris et al., 2012, S. 32ff).

3.1.5 Ausführungsgarantie

Sobald Daten über das Netzwerk geschickt werden, stellt sich die Frage, ob die Daten auch angekommen sind. Grundsätzlich gibt es drei Fälle, die Ankunfts- und Ausführungsgarantien betreffen. Im Optimalfall wird die Nachricht vom Sender zum Empfänger erfolgreich gesendet, dort erfolgreich ausgeführt (im Sinne von: der Server ist nicht ausgefallen) und die Antwort erfolgreich zurückgesendet. Es könnte jedoch auch der Empfänger vor dem Ausführen oder vor dem Versenden der Antwort ausfallen. Ob die Nachricht nicht beim Empfänger ankommt

oder nicht verarbeitet wird, kann vernachlässigt werden, da dies aus Sicht der Ausführung keinen Unterschied macht.

Es gibt drei klassische Fehlersemantiken, die für die Ausführungsgarantie herangezogen werden können. Zum einem at-least-once delivery, die besagt, dass die Nachricht mindestens einmal verschickt. Es wird die Nachricht bei fehlender Antwort wiederholt gesendet, in der Hoffnung die Nachricht kommt beim Empfänger an und kann ausgeführt werden. Dabei werden Duplikate nicht gefiltert. Sollte ein Timeout für die Antwort für den Fall nur zu klein gewählt worden sein, wird auf die vermeintlich nicht angekommene Nachricht eine erneute geschickt und die Nachricht wird erneut verarbeitet. Sollte in einem Fall der Empfänger nicht erreichbar sein, kann auch keine Verarbeitung für diesen Versuch erfolgen. Daher besteht keine Garantie darauf, dass die Nachricht ankommt, denn dazu muss der Empfänger mindestens einmal empfangsbereit sein. Aufbauend darauf gibt es at-most-once, auch hier gibt es die Wiederholung im Fall eines Timeouts. Jedoch werden hierbei auf Empfängerseite die Duplikate gefiltert, sodass trotz mehrerer Versuche die Verarbeitung niemals doppelt erfolgt. Als letzte Variante gibt es exactly-once, welches den theoretischen Fall beschreibt, dass eine Nachricht genau einmal geschickt und genau einmal verarbeitet wird und somit eine Wiederholung nicht nötig ist - eine Forderung, die in der Praxis keine Umsetzung findet (vgl. Tanenbaum, 2003, S.426ff).

3.1.6 Shared state concurrency

Angenommen, es existiert ein Prozess mit mehreren Threads, so ist es möglich im zugewiesenen Speicher Variablen aus verschiedenen Threads zu nutzen. Die technische Realisierung ist sehr einfach, da die Threads über denselben Speicherbereichen verfügen und direkt lesen und speichern können. Damit wird gemeinsam ein Speicherzustand (shared state) verwendet (vgl. Breshears, 2009, S. 15ff).

Unter der Voraussetzung, dass nur gelesen wird, kann sehr performante und gleichzeitig nebenläufige Software entwickelt werden. Performant ist sie, da ein direkter Speicherzugriff erfolgen kann. Sobald Daten überschrieben werden sollen, kann folgender Fall auftreten: Am Beispiel zweier Threads t1 und t2, die auf eine mit 0 initialisierte Variable x zugreifen, kann folgendes Problem entstehen. Beide Prozesse führen einen Lese- und einen Schreibvorgang aus (z.B. $x = x + 1$). Ein nicht-deterministisches Schedular des Betriebssystems weist den Threads Arbeitsslots zu, in denen diese ihren Programmcode ausführen können. Dabei können

diese jedoch jederzeit unterbrochen werden und der jeweils andere Threads kann zum Zuge kommen. Das Ergebnis kann $x = 1$ oder $x = 2$ betragen, je nachdem, in welcher Reihenfolge die Vorgänge ablaufen, und damit den Programmablauf in einem komplexeren Programm maßgeblich verändern.

$$t1 : read(x) \rightarrow t1 : write(x + 1) \rightarrow t2 : read(x) \rightarrow t2 : write(x + 1) \rightarrow x = 2$$
$$t1 : read(x) \rightarrow t2 : read(x) \rightarrow t1 : write(x + 1) \rightarrow t2 : write(x + 1) \rightarrow x = 1$$

Eine Lösung ist die Synchronisierung mittels atomarer Operationen. Es wird damit verhindert, dass bestimmte kritische Abschnitte durch den Schedular unterbrochen werden und so einen Zustand erzeugen, der fehlerhaft ist. Diese kritischen Abschnitte werden mittels eines Mutex gesperrt und am Ende wieder entsperrt (vgl. Breshears, 2009, S. 68ff).

Werden viele Mutexe gemischt, besteht immer die Gefahr, dass ein gegenseitiger Ausschluss (Deadlock) entsteht, indem jeweils auf den anderen Thread gewartet wird (vgl. Breshears, 2009, S. 55ff). Synchronisierung verhindert einen nebenläufigen Ablauf, da es direkte Abhängigkeiten gibt. Es muss immer auf den Thread, der gerade im kritischen Abschnitt ist, gewartet werden. Deadlocks sind schwer zu finden sind, wenn die Anzahl an Mutexen zunimmt, damit wird die Entwicklung sehr aufwändig.

3.2 Aktoren

Ein Aktor ist ein leichtgewichtiger Prozess mit einem Zustand und einem Verhalten, welche nach außen nicht sichtbar sind. Kommuniziert wird dabei mit Nachrichten. Als Basis dieser Nachrichten dienen unveränderbare Datenstrukturen. Diese Datenstrukturen enthalten nur Werte, die nach außen sichtbar sind. Es gibt keine Methoden, die die Werte verändern. Unveränderbare Datenstrukturen können von verschiedenen Prozessen gleichzeitig verwendet werden. Die Prozesse können unabhängig voneinander Methoden von diesen Datenstrukturen ausführen. Es entsteht kein shared state, der durch Locking geschützt werden muss.

In diesem Abschnitt wird auf die Aktorenimplementation Akka 2.11 eingegangen. Akka ist als flexibles Aktoren-Framework zu sehen, welches in der Sprache Scala 2.10 integriert ist und als Werkzeug verwendet werden kann.

Zu jedem Aktor gehört eine Mailbox von Nachrichten, die der Aktor einzeln in seinem Prozess verarbeitet. Wenn eine Nachricht von einem Aktor zu einem anderen geschickt wird, dann wird diese zunächst in der Mailbox des Aktors gespeichert. Eine Nachricht ist als Arbeitsauftrag zu verstehen, für den der Aktor einen Programmablauf bereithält. Von außen betrachtet, entsteht ein autonomes Verhalten, welches man mit dem von Menschen vergleichen kann (vgl. Typesafe, 2013, S. 11). Menschen kommunizieren miteinander, indem sie miteinander sprechen. Jeder Mensch kann selbst entscheiden, wie er auf eine Nachricht reagiert. Ebenso reagiert ein Aktor auf Nachrichten.

Das Verhalten eines Aktors ist durch die Reaktion des Aktors auf eine Nachricht definiert. Reagieren kann ein Aktor intern, indem er in einen anderen Zustand übergeht (s. 3.2.2 Unveränderbare Zustände) und zusätzlich, indem er Nachrichten an den Absender oder einen anderen beliebigen anderen Aktor verschickt (s. 3.2.1 Kommunikation in einem Aktorensystem und vgl. Typesafe, 2013, S. 14).

Eine Mailbox ist eine nicht-deterministische Warteschlange. Diese enthält Nachrichten, die von einem beliebigen Aktor an den Aktor der Mailbox geschickt werden. Die zeitliche Abfolge der eintreffenden Nachrichten ist damit losgelöst von der Abarbeitung der Nachrichten. Der Aktor bekommt eine eine beliebige Nachricht aus der Warteschlange zugeteilt. Die Nachrichten haben keine zeitliche Ordnung. (vgl. Typesafe, 2013, S. 14)

In einem Aktorensystem existieren eine beliebige Anzahl von Aktoren, die jeweils kleinschrittig Aufgaben lösen können. Lösen bedeutet auch Teilen und Delegieren. Eine große Aufgabe wird aufgeteilt und dessen Teilaufgaben an die dafür zuständigen Aktoren delegiert (s. 3.2.4 Delegation als Sicherheits- und Geschwindigkeitswerkzeug).

3.2.1 Kommunikation in einem Aktorensystem

Um ein Aktor zu erstellen, wird das trait Actor benötigt. Es verlangt die Implementierung der partiellen Funktion receive $Any \nrightarrow Unit$ zur Verarbeitung der Nachrichten. Any ist in Scala der oberste Typ aller Typen, damit kann jede Instanz einer Klasse eine Nachricht sein. Unit ist vergleichbar mit void aus Java. Nachdem der Aktor in seinem Lebenszyklus in den Zustand started übergeht, können die Nachrichten in der Mailbox verarbeitet werden.

Die partielle Funktion wird durch Pattern Matching abgebildet. Das Ziel des Pattern Matching (Musterabgleich) ist es, zu einem gegebenen Muster, wie eine Instanz einer Klasse, ein vorgegebenes und passendes Muster zu finden. Dafür wird ein Muster auf einen (Code)Block abgebildet (s. Codebeispiel 3.1). Wird ein passendes Muster gefunden, wird der zugehörige Block ausgeführt. Das Muster bildet zusammen mit dem Block einen Fall (case). Von oben nach unten wird dazu jeder einzelne Fall geprüft. Ein Match beendet das Matching. (vgl. Esser, 2011, S. 23).

```
1 expression match {
2         case pattern1 => block1
3         ...
4         case patternn => blockn
5 }
```

Codebeispiel 3.1: Aufbau des Pattern Matching

Durch Patternmatching werden einzelne Nachrichten zugeordnet einer dafür vorgesehenen Aktion zugeordnet (s. Codebeispiel 3.2). Im folgenden Beispiel wird die Nachricht des Typs String direkt ausgegeben, und bei allen anderen Nachrichten wird ein Standardtext ausgegeben. Wird ein Unterstrich angegeben, so wird die zugeordnet Aktion für alle Nachrichten ausgeführt. Dier hier dargestellte receive Implementierung ist total, denn es gibt für alle Nachrichten einen passenden Fall. Da die receive Funktion, konzeptionell gesehen, partiell ist, muss jedoch nicht für jede Nachricht ein passenden Fall bereitgestellt werden. Wenn kein Fall getroffen wird, erfolgt eine Exception.

```
1  class PrintActor extends Actor {
2     def receive = {
3         case s: String => println("Message received:" +  s)
4         case _         =>  println("unknown message")
5     }
6  }
```

Codebeispiel 3.2: Beispiel eines Aktors

Nachrichten müssen unveränderbar sein. Dies ist ein reines Paradigma, da die zu Grunde liegende Sprache Scala keine Funktionalität bereitstellt, um die Unveränderbarkeit zu erzwingen (vgl. Typesafe, 2013, S. 212). Da ein Aktor eine beliebige Klasse ist, die nur vom Actor trait erben muss (vgl. Typesafe, 2013, S. 205), stehen einem alle Mittel der Sprache und damit auch die der Objektorientierung offen. Seitenffekte abseits des Message Passing sind möglich

und liegen in Eigenverantwortung des Entwicklers. Ein Seiteneffekt kann das Verändern einer Variable im Aktor sein.

Es ist möglich, dem Aktor eine Nachricht zu senden (s. Formel 3.1), indem der Bang oder Tell Operator verwendet wird. In Scala wird dieser Operator in der Regel durch das Ausrufezeichen ausgedrückt. Tell blockiert nicht, erwartet allerdings auch keine Antwort.

$$actor \; ! \; message \tag{3.1}$$

Der Absender gibt durch das Senden einer Nachricht seine Identität preis, die über die Konstante sender erreichbar ist (s. Formel 3.2) und als Adresse zum direkten Rücksenden einer Nachricht verwendbar ist (vgl. Typesafe, 2013, S. 214).

$$sender \; ! \; message \tag{3.2}$$

Es gibt zwei Fälle der Nichterreichbarkeit des Absenders: Zum einem, wenn der Aktor nicht mehr am Leben ist oder die Nachricht nicht im Aktorenkontext gesendet wurde. Dann wird der nicht Erreichbare durch DeadLetters ersetzt (s. 3.2.6 Referenzierung von Aktoren und vgl. Typesafe, 2013, S. 214). DeadLetters nimmt alle Nachrichten auf, die nicht verarbeitet werden, und dient als tote Referenz für nicht oder nicht mehr vorhandene Aktoren.

Wartet man auf eine Antwort einer Nachricht und möchte dabei nicht blockieren, so ist es möglich mit einem Future zu arbeiten. Mit einem Future kann eine blockierende Operation in einem anderen Thread ausgeführt werden. Am Beispiel eines TestActor (s. Codebeispiel 3.3), der auf Nachricht dem Absender direkt antwortet, soll ein Future gezeigt werden.

```
1 class TestActor extends Actor {
2    def receive = {
3        case "test" => sender ! "thanks for your test"
4        case _      => sender ! "I don't understand you, sorry"
5    }
6 }
```

Codebeispiel 3.3: Aktor mit Versendung der Nachrichten an den Absender

Es ist möglich, mit der Funktion ask (s. Formel 3.3) oder dem Fragezeichenoperator eine Nachricht zu verschicken. Es wird ein Future mit einem Timeout zurückgegeben. Ask kehrt

sofort zurück, und in einem anderen Thread wird auf die Rückkehr der Antwort gewartet. Antwortet der Aktor innerhalb des Timeout nicht, wird eine Exception geworfen.

$$val future = test.ask(message)(duration) \tag{3.3}$$
$$val future = test.ask("test")(5\ seconds)$$

Aktoren sind in einem Aktorensystem organisiert (s. Codebeispiel 3.4). Dieses ist eine im Gegensatz zu Aktoren recht schwergewichtige Einheit, deren Sinn es ist, den Rahmen einer logische Applikation zu bilden und dafür nötige Threads und Dispatcher bereitzustellen (vgl. Typesafe, 2013, S. 11). Aktorensysteme müssen einmal mit einem Namen definiert werden, danach können darin einzelne Instanzen von den Aktoren erzeugt werden (vgl. Typesafe, 2013, S. 206).

```
1 object MyApplication extends App {
2         val system = ActorSystem("MyApplication")
3         val testactor = system.actorOf(Props[TestActor])
4         testactor ! "test"
5 }
```

Codebeispiel 3.4: Instanziierung eines Aktorensystems

3.2.2 Unveränderbare Zustände

Sind mehrere partielle Funktionen implementiert, können verschiedene Verhaltensweisen abgebildet werden. Durch die Parameter der Funktion können gleichzeitig individuelle Daten gehalten werden, die zusammen einen Zustand ergeben. Beispielsweise kann ein Zustand ping durch eine Funktion ping definiert werden (s. Codebeispiel 3.5). Die Parameter sind frei wählbar, sodass mit den Parametern Daten übergeben werden können, die genau für diesen Zustand zur Verfügung stehen. Die Funktion enthält ein Pattern Matching. In diesem Fall kann die Nachricht Ping getroffen werden. Sollte ein Ping verarbeitet werden, wird dem Sender die Nachricht zurückgesendet und es erfolgt mit der Funktion become ein Übergang in den Zustand pong. Der Zustand Ping hält die Konstante statecount. Der Zustand pong erhält den Wert statecount + 1. Es kann damit der Zustandswechsel gezählt werden ohne einen Wert zu verändern. Die partielle Funktionen $Any \nrightarrow Unit$ werden in Akka durch den Alias $type\ Receive$ verborgen.

```
def ping(statecount: Int = 0): Receive = {
    case Ping => {
      sender ! Ping
      become(pong(statecount+1))
    }
}
```

Codebeispiel 3.5: Zustandsfunktion ping

Zustandsübergänge werden über einen Stack realisiert, dem partielle Funktionen übergeben werden können. Die Funktion become des Aktorenkontext verwaltet den Stack und ermöglicht so den Übergang in den neuen Zustand. Angenommen, es gibt die Nachrichten Ping und Pong und zwei gleichnamige partielle Funktionen in einem Aktor Worker (s. Codebeispiel 3.6). Im Zustand ping geht der Aktor in Zustand pong über, wenn eine Nachricht Ping verarbeitet wird. Im Zustand pong geht der Aktor in den Zustand ping über, wenn eine Nachricht Pong verarbeitet wird. Damit der Sender die Zustandsänderung erfährt, wird die Nachricht direkt zurückgeschickt. Der Startzustand ist immer in receive. Die Funktion receive ist deckungsgleich zu ping implementiert. Es wird der Zustandsübergang gezählt. Alle Daten bleiben dabei unveränderbar.

```
class Worker extends Actor {
   def receive = {
     case Ping => {
       sender ! Ping; become(pong())
     }
   }
   def ping(statecount: Int = 0): Receive = {
     case Ping => {
       sender ! Ping; become(pong(statecount+1))
     }
   }
   def pong(statecount: Int = 0): Receive = {
     case Pong => {
       sender ! Pong; become(ping(statecount+1))
     }
   }
}
```

Codebeispiel 3.6: Worker-Aktor für Ping-Pong-Protokoll

Um den Worker mit genügend Nachrichten zu versorgen, wird ein Heater Aktor implementiert (s. Codebeispiel 3.7), der das Ping-Pong-Protokoll ausführt. Hält der Heater sich nicht strikt an das Protokoll, wird nicht durch jede Nachricht ein Zustandswechsel im Worker erfolgen. Die Methode preStart wird bei der Initialisierung des Aktors ausgeführt. Um das Verhalten anzustoßen, muss der Heater einmalig in preStart ein Ping senden.

```
1 class Heater extends Actor {
2     override def preStart = {
3       worker ! Ping
4     }
5     def receive = {
6       case Ping => worker ! Pong
7       case Pong => worker ! Ping
8     }
9 }
```

Codebeispiel 3.7: Heater-Aktor für Ping-Pong-Protokoll

Grundsätzlich ist die Verschachtelungstiefe für die Speicherung der Zustände durch den Stack beliebig. Es kann nicht nur auf den Stack ein neuer Zustand hinzugefügt werden, sondern der aktuelle auch wieder entfernt werden. Zum Entfernen dient die parameterlose Funktion unbecome (vgl. Typesafe, 2013, S. 216f).

Wenn Aktoren durch einen Fehler neugestartet werden, gehen diese in den Startzustand receive über. Ist der Startzustand potentiell der falsche Zustand, kann es zu einem ungewünschten Verhalten kommen, da die Umgebung davon direkt nichts mitbekommt. Wenn beispielsweise die Zustände Online und Offline einen Aktor die Möglichkeit geben sollen, den Status einer Verbindung zu einem Gerät abzubilden, so ist bei einem Fehler der Startzustand unproblematisch. Die Verbindung muss ohnehin neu aufgebaut werden. Das Ping Pong Protokoll würde bei einem Fehler nicht eingehalten werden, wenn der Worker im Zustand pong war, da der Startzustand gleich dem ping ist. Es müsste nicht nur der Heater, sondern auch der Worker in den Startzustand zurückversetzt werden, um das Problem zu lösen. (s. 3.2.5 Supervising im Aktorenkontext). Zustände können daher dazu führen, dass Nachrichten nicht oder falsch verarbeitet werden.

3.2.3 Dispatcher als Abstraktionselement

Wichtig ist, dass ein Aktor als Werkzeug zu verstehen ist. Einfachheit ist das Ziel, dies bedeutet beispielsweise: Es kann viele Aktoren (leichtgewichtige Prozesse) geben, ohne dass dies einen direkten Einfluss auf die Laufzeit hat. Denn Aktoren erhalten keinen eigenen Thread, sondern ihnen werden von einem Dispatcher ein Thread nach einer Strategie zugeteilt. Der Dispatcher hat Zugriff auf seine ihm zugeteilten Aktoren und deren Mailboxen und teilt dem Aktor die nächste Nachricht aus der Mailbox zu (vgl. Typesafe, 2013, S. 256f).

Es gibt vier Dispatchertypen. Standardmäßig wird der so genannte $Dispatcher$ verwendet, der einen Threadpool besitzt, aus dem jeder Aktor, der gerade an der Reihe ist, ein Thread erhält und zum Verabreiten einer Nachricht verwenden kann. Jeder Aktor hat eine eigene Mailbox. Der PinnedDispatcher ist gleich dem Dispatcher, nur das für jeden Aktor ein Thread bereitgehalten wird, der keinem anderen anderen Aktor zugeordnet wird. Dieser Dispatcher eignet sich für Operationen im Aktor, die nicht threadsafe sind. Der BalancingDispatcher ist gleich dem Dispatcher, jedoch gibt es nur eine Mailbox. Dies ist zur Arbeitsteilung gedacht. Der Threadpool kann dann nur von Aktoren des gleichen Typs verwendet werden, da sonst das Verarbeiten der Nachricht zur Laufzeit unterschiedlich verlaufen kann (vgl. Typesafe, 2013, S. 257). Der letzte im Bunde ist der CallingThreadDispatcher. Dieser ist ausschließlich zu Testzwecken gedacht und nutzt immer den Thread des aufrufenden Aktors für den derzeitigen Aktor. Mit dem CallingThreadDispatcher lässt sich ein Programmablauf testweise in einem Thread abbilden und somit einfacher untersuchen.

3.2.4 Delegation als Sicherheits- und Geschwindigkeitswerkzeug

Teilaufgaben zu Delegieren ist ein wichtiges Handwerkszeug eines Aktors. Je kleiner die Aufgabe ist, umso schneller ist diese verarbeitet. Ein aufwändiger Vorgang ist beispielsweise ein Film in anderen Format zu konvertieren. Einzelne Filmabschnitte können nebenläufig von verschiedenen Aktoren konvertiert werden und danach wieder zusammengesetzt werden. Außerdem ist die Behandlung eines Fehlers auch um so klarer, denn der Fehler bezieht sich auf eine kleinere Aufgabe. Es ist nur ein Teil der gesamten Konvertierung fehlgeschlagen, wenn nur in einem Teil ein Fehler aufgetreten ist.

Vor allem wenn die Gefahr besonders groß ist, dass einzelne Teilaufgaben schwer zu bewältigen sind, weil beispielsweise ein Zugriff auf das Dateisystem erfolgt, ist Delegation von Teilaufgaben sehr wichtig. Es ist zum Bespiel denkbar, Aktoren nur für diese eine fehlerträchtige Aufgabe zu erzeugen. Alle Nachrichten, die zu dieser fehlerträchtigen Teilaufgabe führen, werden sofort an diese eigens erstellten Aktoren delegiert und können dort abgearbeitet werden. So ist sichergestellt, dass keine dieser Aufgaben einen Aktor aufhält, da die Aufgaben asynchron verarbeitet werden. Sollte eine Teilaufgabe sehr viel Zeit in Anspruch nehmen, ist das sowohl für das Ergebnis, als auch für die Laufzeit der anderen Teilaufgaben unerheblich. Sollte ein Aktor versuchen, alle diese fehlerträchtigen Aufgaben zu lösen, und damit ein langes Blockieren in Kauf nehmen, kommen alle nachfolgend zu verarbeitenden Nachrichten nicht zum Zuge. Hier ist der Geschwindigkeitsvorteil von Delegation besonders offensichtlich. Sollte etwas fehlschlagen, ist nur der kleine Teil der Aufgabe fehlgeschlagen.

3.2.5 Supervising im Aktorenkontext

Zunächst einmal stehen Aktoren auf gleicher Ebene, jeder Aktor kann mit jedem Aktor kommunizieren. Ein Aktor ist für sich genommen eigeständig. Jedoch können Aktoren auch neue Aktoren erzeugen und dadurch eine Eltern-Kind-Beziehung aufbauen. An die Kinder können Aufgaben delegiert werden. Der Elternaktor wacht über den Kindaktor, indem Exceptions, die vom Kind stammen, behandelt werden. Wichtig ist, dass ein Aktor nur als Kind eines anderen Aktors erschafffen werden kann. Es ist nicht möglich, aus diesem Modell auszubrechen. Da das Überwachen im Fokus steht, kann der Elternaktor auch Guardian genannt werden. Das untergebene Kind ist dazu passend der Subordinate. Da ein Guardian prinzipiell beliebig viele Subordinates erzeugen kann, ist die Möglichkeit gegeben eine Baumstruktur aufzubauen (vgl. Typesafe, 2013, S. 11f).

Sollte in einem Aktor ein Fehler auftreten, wird eine Exception geworfen. Dabei ist wichtig zu wissen, dass ein Aktor einen Lebenszyklus besitzt. Dieser kann gestartet (started) sein, terminiert (terminated) sein oder unterbrochen (suspended) sein. Anders als in RMI, in dem der Client (der der die Anfrage schickt) eine RemoteException erhält, bekommt der zuständige Guardian die Nachricht, dass der Subordinate einen Fehler geworfen hat. Der Guardian hat dann vier Möglichkeiten zur Wahl. Wird der Fehler als vorübergehender Fehler eingestuft, d.h. ein zweiter Versuch hat hohe Chancen auf Erfolg, dann wird der Subordinate ohne Änderung wieder zum Leben erweckt (resume). Wird der Zustand des Aktors vom Guardian als fehlerhaft

eingestuft, dann wird der Aktor neugestartet (restart). Der Aktor geht dazu in den Startzustand über, behält dabei jedoch seine Mailbox. Sollte der Fehler dazu führen, dass der Aktor nicht mehr weiterarbeiten kann, dann wird der Subordinate gestoppt (terminate). Sollte der Fehler nicht eingeschätzt werden können, dann wird der Fehler im Baum nach oben zum Guardian des Guardian gereicht (escalate). Sollte ein restart, resume oder terminate durchgeführt werden, wird dies für den gesamten Teilbaum angewendet. Die Wurzel des Teilbaums ist der betreffende Subordinate (vgl. Typesafe, 2013, S. 15). Diese Hierarchie darf jedoch nicht davon ablenken, dass zusätzlich zum Guardian noch andere Aktoren den Lebenszyklus eines Aktors überwachen können, indem diese sich im Aktorenkontext mit der Methode watch registrieren und Nachrichten erhalten, wenn ein Fehler auftritt (vgl. Typesafe, 2013, S. 17).

Akka nutzt die Fehlersemantik at-most delivery, d.h. man hat keine Garantie auf Ankunft einer Nachricht, wohl aber darauf, keine Duplikate zu verarbeiten. Die Behandlung, dass ein Aktor nicht erreichbar ist oder nicht reagiert, bleibt somit in Hand des sendenden Aktors (vgl. Typesafe, 2013, S. 27). Dagegen ist seit Java 1.5 durch das reformierte Java Memory Model garantiert, dass Operationen auf dem Speicher von verschiedenen Threads unter Beachtung der happend-before Relation abgearbeitet werden. Das bedeutet: Eine Variable, die als volatile gekennzeichnet ist (z.B. die MessageQueue), wird immer geschrieben bevor die Variable gelesen werden kann. Ein Lock wird immer geöffnet, bevor das Lock von einem anderen wieder verwendet werden kann. Dadurch kann auch garantiert werden, dass das Sendeereignis einer Nachricht immer vor dem Empfangsereignis liegt. Außerdem wird immer eine Nachricht verarbeitet, bevor die nächste verarbeitet wird (vgl. Typesafe, 2013, S. 25f).

Sollte beim Verarbeiten der Nachricht eine Exception geworfen werden, so wird die aktuelle Nachricht nicht wieder in der Mailbox gespeichert, sondern diese wird verworfen. Soll die Nachricht verarbeitet werden, muss der Fehler abgefangen werden. Die Mailbox bleibt beim einem Fehler zunächst unangetastet, jedoch kann durch die Methode preRestart die Mailbox geändert werden (vgl. Typesafe, 2013, S. 218f).

Damit der Guardian auf den Fehler reagieren kann, ist eine Abbildung nötig, die Fehler auf die gewählten Reaktionen abbildet (s. Codebeispiel 3.8). Wichtig dabei ist, dass nur das behandelt werden sollte, was auch in die Zuständigkeit passt, alles andere sollte an den Wächter delegiert werden (vgl. Typesafe, 2013, S. 15). Die Abbildung Fehler auf Reaktionen benötigt eine Basisstrategie. Akka unterscheidet zwischen: alle für einen oder einen für einen. Ersteres besagt, dass bei einem Fehler eines Kindes die Gegenmaßnahme für alle Kinder des Guardian gelten und auch rekursiv für die Kinder der Kinder. Letzteres besagt, dass die anderen

Kinder des Guardians von der Gegenmaßnahme unberührt bleiben (vgl. Wyatt, 2013, S. 178). Die Abbildung wird als Konstante supervisorStrategy gespeichert, in der auf die einzelnen Exceptions die Reaktion definiert wird.

```
override val supervisorStrategy =
  OneForOneStrategy {
    case _: ArithmeticException       => Resume
    case _: NullPointerException      => Restart
    case _: IllegalArgumentException => Stop
    case _: Exception                 => Escalate
  }
```

Codebeispiel 3.8: Strategie eines Aktors für Exceptions

Wenn man den Baum weiter betrachtet, ist es unumgänglich sich zu fragen, wie ein Baum entsteht. Aktoren sind in einem Aktorensystem organisiert. In diesem Aktorensystem gibt es genau einen Baum. Die Wurzel dieses Baumes (Root Guardian) ist dabei fest definiert und automatisch der Wächter über den gesamten Baum. Da die Wurzel keinen Guardian haben kann, ist dies ein spezieller Aktor. Die Wurzel hat nur die Aufgabe hat den Abschluss zu bilden. Der Wächter stoppt alle Aktoren, wenn ein Fehler zu ihm hochgereicht wird. Die Wurzel hat genau zwei Kinder, die auch standardmäßig zur Verfügung stehen, wenn ein Aktorensystem gestartet wurde.

Da gibt es den User Guardian, der alle Aktoren in seinen Kinderteilbäumen hat, die vom Nutzer des Aktorensystems erstellt werden. Der System Guardian hat spezielle Aktoren wie Logging als Kinder, die vom Akka im Hintergrund erzeugt werden. Der System Guardian überwacht den User Guardian und übernimmt die Fehlerbehandlung. Der System Guardian verfolgt die Strategie, einen Aktor im Fehlerfall neu zu fstarten, es sei denn, es liegt ein Initialisierungs- oder Terminierungsfehler vor, dann wird der Aktor beendet. (vgl. Typesafe, 2013, S. 16).

Wenn eine Exception zu einem Neustart führt, gelten folgende Besonderheiten: Voraussetzung für den Neustart ist, dass der Aktor, wie auch seine Kinder, unterbrochen (suspended) sind. Um den Neustart durchzuführen, gibt es einige Hookmethoden, die es ermöglichen Operationen auszuführen, wenn sich ein Zustand ändert. Die Hookmethoden sind standardmäßig implementiert und müssen bei Bedarf überschrieben werden. Es wird eine neue Instanz des Aktors angelegt und postRestart aufgerufen, das preStart standardmäßig aufruft. Die Methode preRestart wird auch beim Starten des Aktoren aufgerufen. Bevor der Aktor wieder zum Leben erweckt wird und seine Mailbox verarbeiten kann, werden die Kinder neugestartet.

Zu beachten ist, dass die Kommunikation zwischen den Kindern und dem neu zu startenden Aktor nicht blockierend verläuft (vgl. Typesafe, 2013, S. 17).

3.2.6 Referenzierung von Aktoren

Da Aktoren in einem Aktorensystem organisiert sind, müssen sie sich dort auch zurechtfinden bzw. andere Aktorensysteme ansprechen können um zu kommunizieren. Als Basis dafür werden in Akka Universal Resource Identifier (URI) und Universal Resource Locater (URL) verwendet. Eine URI unterscheidet sich syntaktisch nicht von einer URL, jedoch in der Semantik. Eine URI bezeichnet eine global eindeutige Ressource (z.B. ein Webserver, ein Aktor, ein Aktorensystem). Eine URL bezeichnet den Ort der Ressource. Aufgebaut sind beide aus einem Scheme, einer Authority und einem Path (s. Formel 3.4), die sich zu einem String zusammensetzen (vgl. Colouris et al., 2012, S. 584f).

$$\begin{gathered} Scheme://Authority/Path \\ akka://ActorSystem/ActorName \end{gathered} \tag{3.4}$$

Es gibt dabei zwei Möglichkeiten ein Aktorensystem zu adressieren. Einmal aus Sicht des lokalen Aktorensystems, welches immer über $akka://ActorSystemName$ ansprechbar ist, zum anderen global über eine Adresse in Form $akka://username@host/ActorSystemName$. Der in der Hierarchie höchste Aktor wird dabei auch als erstes genannt. Dies ist immer der User Guardian. Sollte ein Aktor tiefer in der Hierarchie adressiert werden, so muss der Pfad durch den Baum vollständig angegeben werden (s. Formel 3.5). Dadurch kann man jeden Aktor und jedes Aktorensystem durch eine URI adressieren (vgl. Typesafe, 2013, S. 18).

$$akka://myActorSystem/user/myGuardian/myChild \tag{3.5}$$

Akka arbeitet grundsätzlich nur mit diesen Referenzen, es werden keine Aktoren direkt referenziert, sondern Aktorreferenzen (ActorRef) erzeugt, die genau diese Adressen enthalten. Ein Aktor ist über seinen Aktorenkontext in der Lage, an seine eigene ActorRef, die ActorRef seiner Kinder (children) und seines Wächters (parent) zu gelangen. Es ist möglich, aus jedem ActorRef eine wohlgeformte URI als String zu erzeugen (vgl. Typesafe, 2013, S. 18f).

Alle bisher angesprochenen ActorRef sind logische Referenzen, die sich nur in der Hinsicht unterscheiden, ob sie lokal oder global gelten. Physikalische ActorRef sind auch möglich

(s. Formel 3.6). Diese geben den Ort eines Aktors an, der in einem anderen Aktorensystem liegen kann, sind aber logisch im lokalen Aktorensystem eingeordnet und verhalten sich auch so. Damit ist ein Verweis möglich, der RemoteActorRef genannt wird. Eine RemoteActorRef enthält eine URL, da es den Ort explizit angibt. Die URL verweist statt auf user auf remote. Danach folgt die physikalische Adresse (vgl. Typesafe, 2013, S. 19ff).

$$akka://ActorSystem/remote/user@host/user/... \tag{3.6}$$

Grundsätzlich kennt ein Aktorensystem nur die eigenen Aktoren. Unter Angabe einer URI kann das Aktorensystem die Referenz prüfen. Wenn der Aktor lokal nicht existiert, wird die ActorRef an das Aktorensystem verschickt, dass in der URI angegeben wurde (vgl. Typesafe, 2013, S. 19f).

Ist ein Aktor nicht oder nicht mehr erreichbar, so wird ein DeadLetters (s. Formel 3.7) als ActorRef verwendet. DeadLetters ist ein Aktor, der immer vorhanden ist und alle unzustellbaren Nachrichten verwaltet. Insbesondere für das Finden von Fehlern ist dieser interessant (vgl. Typesafe, 2013, S. 31f). Wird eine Nachricht außerhalb eines Aktors an einen Aktor verschickt, dann ist der Abesender der Nachricht DeadLetters, denn nur Aktoren können adressiert werden. Beispielsweise bei der Verwendung eines Future in einem Aktor kann der Absender nicht aufgelöst werden, denn die Verarbeitung erfolgt in einem anderen Thread.

$$akka://ActorSystemName/deadLetters \tag{3.7}$$

ActorRef ist das Fundament für die Lokalisationstransparenz, da es eine Adressierung ermöglicht, mit der zugleich der physikalische Ort verborgen werden kann. Anzumerken ist, dass es ein Unterschied bleibt, ob Nachrichten lokal oder übers Netzwerk verschickt werden. Die Gefahr, dass eine Nachricht verloren geht, ist zwar immer vorhanden, fällt jedoch bei lokaler Versendung deutlich kleiner aus (s. 3.1.5 Ausführungsgarantie und vgl. Typesafe, 2013, S. 24f).

Interessant ist auch, dass die Konfigurationm wie Aktoren in einem Aktorensystem organisiert sind und welche Dispatcher, Mailboxen und Konstanten diese benutzen, als Datei abgelegt werden kann. Die Konfiguration kann zur Laufzeit getauscht werden und damit auch die Position des Aktors sehr einfach verändert werden (vgl. Typesafe, 2013, S. 31f).

3.3 Kommunikation mit heterogenen Systemen

Frameworks, wie das auch in Akka integrierte Apache Camel (vgl. Typesafe, 2013, S. 336f), bieten eine einheitliche Schnittstelle, um mit diversen Kommunikationsprotokollen wie FTP oder HTTP zu arbeiten. Da stellt sich die Frage, weshalb für die Kommunikation eine besondere Betrachtung nötig ist. Um mit Webservices wie einem RSS Feed zu kommunizieren, ist Apache Camel eine einfache, ausreichende Möglichkeit. Jedoch genügen allgemeine Protokolle nicht, um aus der Sicht der statisch typisierten Programmiersprachen, Nachrichten sicher zu senden bzw. zu empfangen.

3.3.1 Serialisierung

Datenstrukturen, die in eine Programmiersprache implementiert werden, können in eine externe Repräsentation überführt werden. Auf Basis eines zuvor definierten Formats können Daten und Typinformationen in eine sequenzielle Darstellungsform überführt werden, dazu werden die Daten aufeinander folgend in beispielsweise einen String geschrieben. Serialisierung bezeichnet das Überführen in eine sequenzielle Darstellungsform, Deserialisierung das Überführen aus der sequenzielle Darstellungsform in eine wählbare Sprache. Serialisierung und Deserialisierung benötigen ein gemeinsames Format, damit die Daten ohne Fehler überführt werden können. Mit Basistypen, für die jeweils eine sequenzielle Darstellungsform definiert werden müssen, können auch komplexere Datenstrukturen serialisiert werden (vgl. Colouris et al., 2012, S. 174f).

Die Kommunikation mit einem Roboter besteht aus vielen kleinen Nachrichten, sodass die Überführung möglichst schnell gehen sollte. Je größer die Nachricht ist, die serialisiert wird, umso weniger Zeit wird im Verhältnis zur Überführung benötigt. Außerdem ist die Rechenleistung des Roboters begrenzt. Sprachen, die eine serialisierte Abbildung von Daten ermöglichen sind beispielsweise JavaScript Object Notation (JSON) oder eXtensible Markup Language (XML). XML gilt als zu voluminös und langsam. JSON dagegen ist leichtgewichtig, jedoch fehlt es an der Mächtigkeit, um beispielsweise starke Typisierung zu realisieren (vgl. Sumaray and Makki, 2012, S. 48:1).

Sprachen wie Java besitzen eine eigene Serialisierung, mit der Objekte automatisiert in einer Datei gespeichert werden können. Diese Serialisierung ist jedoch sehr sprachenspezifisch und

daher nicht geeignet ist, um sprachenunabhängig Daten zu serialisieren (vgl. Colouris et al., 2012, S. 178f). Akka verwendet für Message Passing standardmäßig die Java Serialisierung, die jedoch auch gegen Protobuf oder eine eigene Implementierung durch die Konfiguration ausgetauscht werden kann (vgl. Typesafe, 2013, S. 283ff).

3.3.2 Bindung an Zielsprachen

Um Daten sprachenunabängig zu speichern, ist die Serialisierung eine grundlegender Mechanismus. Wenn nur wenige Datenstrukturen in kleinen distributiven Systemen verwendet werden, ist ein eigenes, minimalistisches Format ein funktionierendes Mittel. Die verwendeten Datenstrukturen, die zwischen verschiedenen Systemen ausgetauscht werden, stellen die Schnittstellen der Systeme dar. Führt man sich vor Augen, dass verschiedene Systeme von unterschiedlichen Autoren mit mehreren Schnittstellen und Zielsprachen entwickelt werden, muss sichergestellt sein, dass ein präzises und einheitliches Format vorhanden ist. Für das Format wird eine möglichst totale Abbildung denkbarer Datenstrukturen benötigt, ohne dabei sprachenspezifisch zu werden (vgl. Nestor et al., 1981, S. 7f).

Die Wartbarkeit der Schnittstellen ist wichtig, um Fehler bei der Kommunikation zu vermeiden. Denn Schnittstellen verändern sich im Zuge der Entwicklung einer Software kontinuierlich. Trotzdem muss die Serialisierung funktionstüchtig bleiben, d.h. die Datenstruktur, die serialisiert wird, muss bei der Deserialisierung auch wieder zur gleichen Datenstruktur werden. Wenn ein Format in einer eigenen Sprache genau einmal definiert wird und im distributiven Systemen verteilt wird, dann kann sichergestellt werden, dass alle Teilsysteme die selben Datenstrukturen verwenden und erwarten (vgl. Nestor et al., 1981, S. 7f).

Eine Lösungsmöglichkeit stellt eine spezielle Sprache dar, die Schnittstellen definiert, die Interface Description Language (IDL). Heute stellt die IDL eine Kategorie dar, in der sich verschiedene Sprachen, wie die Protobuf IDL, einordnen lassen. Die erste IDL (vgl. Nestor et al., 1981) ist sehr mathematisch gesprägt. Kompositioniert aus möglichst allgemeinen Basistypen, wie Wahrheitswert, Zeichen, Zahlen, Listen und Mengen, können mit dieser Sprache programmiersprachenunabhängig eigene Typen definiert werden (vgl. Nestor et al., 1981, S. 18f). Im Einzelnen müssen die Datentypen sehr genau betrachtet werden. In der der IDL werden Zahlen nicht in verschiedene Typen aufgeteilt (s. Formel 3.8). Es gibt nur den Wertebereich der rationale Zahlen (s. Formel 3.9), denn damit sind auch Dezimalzahlen mit fester Anzahl an Nachkommastellen und ganze Zahlen abgedeckt. Rationale Zahlen

werden dazu aus Effizienzgründen in Sprachen, wie Java, gar nicht vollständig abgebildet. Es werden dagegen Dezimalzahlen mit fester Anzahl an Nachkommastellen verwendet. Dies zeigt Limitierungen beim Überwinden der Heterogenität. Die maximale Verallgemeinerung macht es schwierig Daten typsicher zu verwenden.

$$Integer(3) \neq Float(3.0) \tag{3.8}$$

$$\frac{1}{3} \neq 0,3333333 \tag{3.9}$$

Neben XML und JSON gibt es auch die binäre Serialisierung. Die Datenstrukturen werden statt in Strings in eine Folge von Bytes überführt. Apache Thrift und Google Protocol Buffers (ProtoBuf) sind leichtgewichtige, binäre Alternativen. Beide bieten die Möglichkeit Datenstrukturen in einer DSL zu definieren. Verteilt man alle Definitionen von Datenstrukturen an alle Systeme, die diese Datenstrukturen verstehen sollen, können die Datenstrukturen ausgetauscht werden. Der Aufbau der Datenstruktur ist in der serialisierten Form nicht vorhanden. Die Überführung in die Zielsprache funktioniert nur mit Hilfe der Definition aus der DSL. Der Protobuf Compiler kann den für die Serialisierung und Deserialisierung nötigen Code in der gewünschten Zielsprache generieren. (vgl. Sumaray and Makki, 2012, S. 48:2f).

Am Beispiel der Datenstruktur $case\ class\ Foo(i : Int, s : String, l : List[String])$ werden die IDLs von Protobuf (s. Codebeispiel 3.9) und Thrift (s. Codebeispiel 3.10) vorgestellt. Protobuf bezeichnet Datenstrukturen explizit als $message$, die in diesem Fall i, s und l als Konstanten enthalten. Die Konstanten werden mit Name, Typ und einer Reihenfolge, die durch Zahlen definiert ist, angegeben. Listen werden durch das Schlüsselwort $repeated$ realisiert. Das Schlüsselwort $required$ kennzeichnet die Konstante als notwendig, dagegen kann bei dem Schlüsselwort $optional$ die Konstante weggelassen werden (vgl. Google, 2013).

```
1 message Foo {
2   required int32 i = 1;
3   required string s = 2;
4   repeated string l = 3;
5 }
```

Codebeispiel 3.9: Beispiel für eine Protobuf-Datenstruktur

Thrift baut die Datenstruktur ähnlich auf. Thrift verwendet das Schlüsselwort $struct$ statt $message$, es unterstützt statt $repeated$ explizit den Typ $list$ (vgl. Apache, 2013).

```
struct Foo {
    1: required int i,
    2: required string s,
    3: required list<string> l,
}
```

Codebeispiel 3.10: Beispiel für eine Thrift-Datenstruktur

Eine besondere Herausforderung ist es union types sprachenunabhängig zu verwenden. In der Sprache C++ werden union types aus Effizienzgründen unterstützt. Es steht ein Typ für mehrere andere Typen. Man kann einen union type als „überladenen Datentyp"auffassen. In Java oder Scala werden diese Typen nicht unterstützt, da diese die Typsicherheit vermindern. Es ist zur Laufzeit nicht bekannt, welcher Typ darin enthalten ist. Es können, beim Zugriff auf die Daten, Fehler entstehen. In zeitkritischen Anwendungen werden vermischte Typen verwendet (s. 4.3 Kommunikation mit dem Nao). Wenn solche in Schnittstellen von anzubindenden Fremdsystemen vorkommen, müssen diese auch in der IDL abbildbar sein (s. 3.11). Dazu werden mehrere optionale Typen definiert, von denen einer in der konkreten Instanz belegt ist. Welcher Typ belegt ist, kann über entsprechenden Prädikate, wie hasBar, für die Variable bar abgefragt werden.

```
message uniontype {
  optional Foo foo = 1;
  optional Bar bar = 2;
  optional Baz baz = 3;
}
```

Codebeispiel 3.11: Beispiel für einen „überladenen Datentyp"

Eine flexible Integration in die Umgebung ist insbesondere durch die Anwendung und Etablierung von Standards möglich. Um so bekannter, anerkannter und verbreiteter eine IDL ist, umso mehr Synergieeffekte können gebündelt werden. Ein Synergieeffekt kann Erfahrung mit der IDL sein. Ein weiterer Synergieeffekt kann die schon vorhandene Unterstützung der IDL in anzubindenden Fremdsystemen sein. Die schon vorhandene Unterstützung ermöglicht ein einfacheres Zusammenarbeiten, da die Formate einheitlich sind. Daher ist eine Verwendung von einer verbeiteten IDL besonders wichtig.

3.3.3 Framebasierte Kommunikation

Wenn nicht nur Sprachen, sondern auch das Netzwerk überwunden werden muss, wird ein Möglichkeit zum Transport der Daten benötigt. In der dezentralen, nebenläufigen Welt müssen Computersysteme miteinander kommunizieren. Verschiedene Kommunikationsmechanismen kommen zusammen, die gebündelt werden müssen um eine Einheit als distributives System zu bieten aber auch ohne den einzelnen Systemen das Design vorzuschreiben (vgl. Hintjens, 2013, S. 11).

ØMQ ist ein auf TCP Sockets und TCP Frames basiertes Kommunikationsframework, welches fokussiert ist, Administration und Kosten auf möglichst wenig (Ø = Zero) zu reduzieren (vgl. Hintjens, 2013, S. 8) und zugleich die technische Organisation von TCP wie Sockets nicht mehr in die Hände des Benutzers zu legen (vgl. Hintjens, 2013, S. 12). ØMQ ist in C++ geschrieben und wird im Rahmen dieser Arbeit über Javabindings verwendet. ØMQ übernimmt explizit nicht die Aufgabe die Daten zu schützen. Es ist ein binäres Protokoll, welches dafür gerade steht, dass die versendeten Bytes unverändert und in der gleichen Reihenfolge ankommen. Die Verständlichkeit der Daten legt es jedoch in Hand des Benutzers. Dazu wird insbesondere auf IDLs wie Protobuf verwiesen. Verläuft die Kommunikation ohne IDL, ist der Benutzer in der Pflicht das Bytearray zu kodieren und auch zu dekodieren (vgl. Hintjens, 2013, S. 16f).

Um über ØMQ kommunizieren zu können ist ein ØMQ Context nötig, der einen I/O Thread für die Kommunikation verwendet (vgl. Hintjens, 2013, S. 39). Im ØMQ Context können ØMQ Sockets, über die kommuniziert wird, erstellt werden (vgl. Hintjens, 2013, S. 27).

ØMQ Sockets sind keine TCP Sockets, sondern stellen eine Schnittstelle (vgl. Hintjens, 2013, S. 34) dar, die wie ein Socket zu benutzen ist, jedoch intern die Zahl der TCP Sockets selbständig minimiert. Für ein Tupel (Sender,Empfänger) gibt es genau einen TCP Socket auf jeder Seite. Es kann jedoch beliebig viele ØMQ Sockets geben. Außerdem können ØMQ Sockets aus beliebigen Threads heraus erstellt werden. Die ØMQ Sockets können über den ØMQ Context die internen TCP Sockets verwenden (vgl. Hintjens, 2013, S. 32). Eingehende Nachrichten werden sofort akzeptiert, das Senden von Nachrichten erfolgt ohne zu blockieren. Dadurch ist ein asynchrones Verhalten gewährleistet, welches für Message Passing eine wichtige Anforderung darstellt (vgl. Hintjens, 2013, S. 36). Es können zwar, ohne Geschwindigkeit einzubüßen, viele ØMQ Sockets erstellt werden, diese sind jedoch nicht threadsafe (vgl.

Hintjens, 2013, S. 64). Jede Instanz eines ØMQ Sockets muss in genau einem Thread verwendet werden.

ØMQ Sockets haben genau einen Sockettype. Die Sockettypes beschreiben, wie sich ein Socket verhält. Die Basistypen sind Request (REQ) und Reply (REP). Zusammen bilden diese das Request-reply Pattern. Es muss immer zuerst ein Request gesendet werden und auf diesen kann mit einem Reply geantwortet werden. Die Reihenfolge muss gleich bleiben. Es kann also kein Reply ohne vorherigen Request gesendet werden (vgl. Hintjens, 2013, S. 40) .

Am Beispiel einer Client/Server Kommunikation (s. Codebeispiel 3.12) soll die grundlegende Arbeitsweise von ØMQ mit Strings verdeutlicht werden. Gebunden wird der REP Socket an einen Host und einen Port. Der Server wartet auf Nachrichten im Format eines Bytearrays, die im folgenden Beispiel in einen String umgewandelt werden. Eine konstanter String wird konkateniert und das Ergebnis zurückgeschickt. Strings können in ein Bytearray umgewandelt werden, wenn per Konvention jedes Zeichen genau ein Byte lang ist. Wenn ein Zeichen in einem Byte nicht abbildbar ist, so muss das System, das die Daten liest, damit umgehen können.

```
1  object Server extends App {
2    implicit def byteArrayToString(b: Array[Byte]) =
3          new String(b, 0, b.length - 1)
4
5    val context = ZMQ.context
6    val socket = context.socket(ZMQ.REP)
7    socket.bind("tcp://*:5555")
8
9    while (true) {
10     val request: String = socket.recv(0)
11     socket.send((request + " ist angekommen").getBytes, 0)
12   }
13 }
```

Codebeispiel 3.12: Beispiel für einen ØMQ Server

Der Client ist ähnlich aufgebaut, er hat jedoch den Sockettype REQ. Der Client schickt genau eine Nachricht und wartet auf dessen Antwort.

```
object Client extends App {
  implicit def byteArrayToString(b: Array[Byte]) =
        new String(b, 0, b.length - 1)

  val context = ZMQ.context
  val socket = context.socket(ZMQ.REQ)

  socket.connect("tcp://localhost:5555")
  socket.send("Meine Nachricht".getBytes, 0)
  val answer: String = socket.recv(0)
}
```

Codebeispiel 3.13: Beispiel für einen ØMQ Client

Nachrichten sind in ØMQ auch in mehrere Teile, d.h. in mehrere Frames, zerlegbar. Diese Nachrichten heißen Multipart-Nachrichten. Eine Frame ist in ØMQ ein Teil des Bytearrays, dass die Nachricht enthält. Die Frames werden einzeln über das Netzwerk geschickt. Die Nachrichten werden jedoch nur atomar weitergegeben, d.h. ØMQ setzt die Frames bei der Ankunft wieder zusammen, ohne dass dies außerhalb der API zu erkennen ist. Es kann damit nicht passieren, dass Nachrichten unvollständig sind (vgl. Hintjens, 2013, S. 43). Durch Markierungen (Integer) können beim Senden die Nachrichten konkateniert werden. Die 0 schließt eine Nachricht ab (vgl. Hintjens, 2013, S. 46) .

Eine Besonderheit der Sockettypes stellt Dealer dar, der beliebig viele Nachrichten an einen Reply Socket senden kann, ohne eine Antwort zu erhalten, und ein Router, der beliebig viele Nachrichten an einen Request Socket schicken kann. Es können Nachrichten von einem Socket zu einem anderen Socket delegiert werden, indem auf einem Rechner auf einem Socket einen Nachricht empfangen wird und über einen anderen Socket weitergesendet wird. So werden Dealer und Router kombiniert (Abb. 3.1 Multihreaded Server (Hintjens, 2013, S. 66 Figure 20)) um einen Broker erstellen, der von einer beliebigen Anzahl an Sendern (vom Typ REQ) über einen Router - Dealer an beliebige andere Empfänger (vom Typ REP) Nachrichten verteilt (vgl. Hintjens, 2013, S. 50).

Wenn ein Teil des distributiven Systems nicht erreichbar ist, stellt ØMQ automatisch und damit unabhängig von der Anwendung sicher, dass die Nachrichten ankommen, sollte der Empfänger wieder erreichbar werden. Ein dafür erforderlicher Verbindungsaufbau (s. at-most once in 3.1.5 Ausführungsgarantie) wird von ØMQ intern realisiert. Dazu können Timeouts

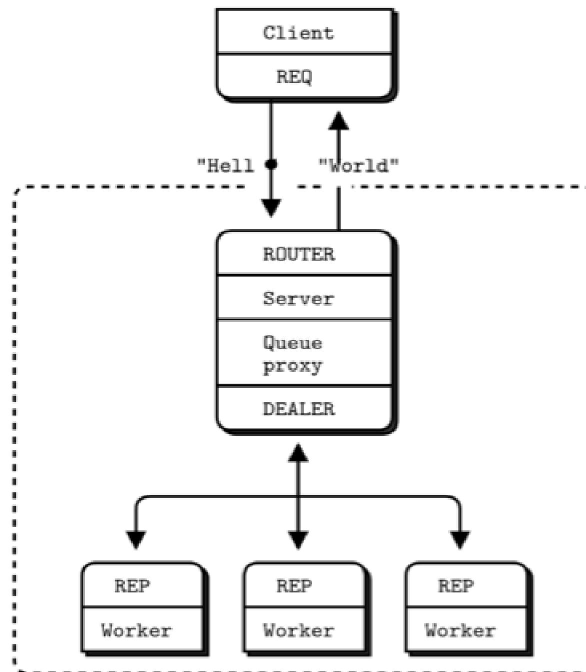

Abbildung 3.1: Multihreaded Server (Hintjens, 2013, S. 66 Figure 20)

definiert werden (vgl. Hintjens, 2013, S. 136ff). Es können somit zuerst Nachrichten an einen Socket verschickt werden und danach erst das Zielsystem gestartet werden. Die Nachricht kommt trotzdem beim Empfänger an. (vgl. Hintjens, 2013, S. 186ff).

4 Realisierung einer distributiven Steuerung

Im Rahmen des Projektes Humanoider Roboter an der HAW Hamburg im Sommersemester 2012 und im Wintersemester 2013 wurde eine Steuerung eines humanoiden Roboters am Beispiel des Naos entwickelt. Im Fokus stand dabei eine Architektur, die insbesondere Fehlertoleranz und Sprachenunabhängigkeit in einem distributiven System mit sich bringt. Entwickelt wurde im Rahmen der Bachelorarbeit die Kommunikation mit Nao, außerdem wurden grundlegende Designentscheidungen für eine Architektur getroffen, die in dem Projekt zum Entwickeln von Applikationen verwendet werden kann.

4.1 Vorstellung des Naoqi SDK

Der Nao von Aldebaran enthält das Gentoo Linux basierte Betriebssystem OpenNao (vgl. Aldebaran, 2013a), das auf die genutzte x86 Architektur zugeschnitten ist. Darauf wird das Metabetriebssystem Naoqi ausgeführt, das den Nao steuert. Naoqi ist dafür in verschiedene C++ oder Python Module aufgeteilt, die spezielle Aufgaben übernehmen, wie Bewegung (ALMotion), Sensoren (ALSensors) oder die Bereitstellung einer Datenbank (ALMemory), in der z.B. Positionen gespeichert werden können (vgl. Aldebaran, 2013d).

Beim Start des Naoqi-Prozesses werden die in der Konfiguration definierten Module initialisiert und über einen Broker zur Verfügung gestellt. Die Module können ausschließlich über den Broker von außen aufgerufen werden. Jeweils für ein Modul ist genau eine API definiert, die der Broker nach außen anbietet.

Für das Verwenden der API von einem entfernten Rechner stellt der Hersteller in mehreren Sprachen (u.a. Python und C++) implementierte SDKs zur Verfügung (vgl. Aldebaran, 2013e),

die für jedes Modul ein Proxyobjekt (s. Codebeispiel 4.1) bereitstellen. Die Proxyobjekte stellen die API durch RPCs bereit.

```
AL::ALTextToSpeechProxy tts("<IP of your robot>", 9559);
tts.say("Hello world from c plus plus");
```

Codebeispiel 4.1: Erstellung eines Proxyobjekts mit dem Naoqi SDK in C++

Für Java existieren automatisch generierte Bindings, die sich noch in einer frühen Entwicklungsphase befinden. Es werden keine Callbacks unterstützt, da Java keine Callbacks direkt unterstützt. Die Proxys müssen für jede Sprache und jede API Version angepasst werden und erhöhen damit den Entwicklungsaufwand für eine vollständige Umsetzung. Die SDks werden für Windows, Linux und Mac bereitgestellt, allerdings für Windows nur in 32 bit (vgl. Aldebaran, 2013f).

Wenn das SDK genutzt wird, stößt man nicht nur bei der Wahl seines Prozessors und seiner Sprache auf Restriktionen, sondern man trifft auch auf Geschwindigkeitsprobleme. Naoqi komprimiert keine Bilder, was jedoch für eine flüssige Darstellung äußerst wichtig ist. Es lässt sich z.B. über 100Mbit Ethernet ein unkomprimiertes Bild mit der Auflösung 1280x960 im Farbraum 24 Bit RGB (1 Pixel entspricht 3 Byte) ca. drei Mal pro Sekunde übertragen (vgl. Aldebaran, 2013g). Drei Bilder pro Sekunde genügen nicht, um denn dem menschlichen Auge ein Video flüssig erscheinen zu lassen. Jedoch wird ein mobiler Roboter, der WLAN 802.11g unterstützt, durch ein Kabel unnötig eingeschränkt. Mit WLAN kann weniger als ein Bild pro Sekunde übertragen werden.

Auch einfache Befehle wie die Abfrage der Laustärkeeinstellung, erscheinen durch die XML basierte Übertragung der Daten sehr schwergewichtig. Dies hat sich beim Entwickeln mit dem SDK gezeigt. Eine Zeitabstand zwischen Anfrage und Antwort von min. 200ms wurde bei eigenen Tests gemessen. Ein RPC ist in den SDKs immer blockierend realisiert. Um nebenläufige Prozesse mit den RPCs abzubilden, wird eine Auslagerung in z.B. Futures benötigt.

Zur Entwicklung von eigener Modulen, die direkt in Naoqi integriert werden, bietet der Hersteller ein Cross-Platform build system qiBuild (vgl. Aldebaran, 2013h). Damit lassen sich auf einem anderen Rechner Module kompilieren, die nativ auf dem Nao ausgeführt werden können. Zusammen mit OpenNao, das ohne Paketverwaltung ausgestattet ist, ist das qibuild ein geschlossenes System. Die Integration neuer Software in einem geschlossenen System ist

schwierig, denn die vorgefertigten Werkzeuge können nicht verwendet werden, sondern die Bibliotheken müssen manuell installiert und gewartet werden.

Die Entwicklung von nativen Modulen ist auch durch die Rechenleistung stark beschnitten, sodass rechenintensive Applikationen wie Bilderkennung nicht lokal ausgeführt werden können, ohne den Roboter zu überlasten. Naoqi nutzt intern eine Synchronisierung der Daten, die speziell für die vom Hersteller ausgegebenen Module definiert ist. Wenn Synchronisiert wird, müssen Threads gegebenenfalls auf andere Threads warten. Damit beispielsweise bei der Bewegung des Roboters das zuständige Modul den Roboter jederzeit unter Kontrolle hat, muss die Wartezeit klein genug sein. Eigene Module können die Wartezeit anderer Module erhöhen und damit deren Verhalten negativ beeinflussen.

Da Naoqi im Gegensatz zu OpenNao nicht Open Source ist, kann die Implementierung auch nicht eingesehen werden. Eigene Erfahrungen zeigten, dass der Nao sehr einfach zu Fall gebracht werden kann, wenn Arme und Beine gleichzeitig bewegt werden. Die Trägheitskraft, die beim schnellen Bewegen der Arme entsteht, beeinflusst das Gleichgewicht. Werden die Arme während des Laufens schnell bewegt, kann der Nao sein Gleichgewicht nicht halten. Daher ist es nötig, die gleichzeitige Bewegung von Armen und Beinen zu kontrollieren.

4.2 Anwendung: Steuerung durch Smartphone und Spielecontroller

Ziel des Projekts war die Realisierung ein konkretes Anwendungsszenarios. Dieses Szenario bestand daraus einen Nao zu steuern. Es wurden verschiedene Controller verwendet, wie z.B. ein Android Smartphone, ein XBOX Spielecontroller und eine Tastatur.

Die Controller haben unterschiedliche Stärken und Schwächen und sind daher bewusst ausgewählt, um die Steuerung auf Vielseitigkeit zu testen. Die Tastatur besteht nur aus Tasten und eignet sich daher für wenige bestimmte Aktionen wie Rechts, Links, Vorwärts, Rückwärts. Eine genauere Steuerung, wie durch Angabe eines Winkels, ist nicht durch einen einzigen Tastendruck abbildbar. Wenn für den Roboter geradeaus als Referenzwinkel (0°) angesehen wird, kann von diesem ausgehend ein Winkel angegeben werden, in den der Roboter laufen soll. Der XBOX Controller besitzt zusätzlich einen Joystick und kann daher einen Winkel abbilden.

Das Smartphone kann Videos darstellen und damit auch das Videobild des Naos. Mit dem Videobild kann der Nao gesteuert werden, ohne das ein direkter Sichtkontakt besteht. Allerdings ist dazu auch eine Bewegung des Kopfs nach rechts, links, oben und unten nötig. Wird der Kopf bewegt, verändert sich auf der Blickwinkel der Kamera im Kopf des Nao. Damit kann man sich in der Umgebung des Nao besser orientieren.

Der Nao hat eine sogenannte „sichere Position “. In dieser Position sitzt der Nao und stützt die Arme auf die Knie. Dadurch kann er nicht fallen und kann gleichzeitig Strom sparen, da seine Motoren in den Gelenken abgeschaltet werden können. Es muss daher durch den Controller ein Aufstehen und Hinsetzen per Knopfdruck möglich sein.

Die Anforderung ist, dass alle Controller über eine App einen Nao steuern können. Die Steuerung wird losgelöst vom Nao auf einem eigenen Rechner ausgeführt. Dazu wird das Aktorensystem Akka verwendet. Das Naoqi SDK wird nicht verwendet. Nachrichten werden, unabhängig von einer Zielsprache, mit der Protobuf IDL definiert und mit ØMQ zum Nao geschickt.

4.3 Kommunikation mit dem Nao

Für den Transport der Nachrichten über das Netzwerk bietet OpenNao eine Unterstützung für ØMQ in der Version 2.2, die auch in der qibuild direkt verwendet werden kann. ØMQ wird derzeit nicht von Naoqi verwendet, jedoch ist es auf OpenNao vorinstalliert. Da ØMQ Bytearrays versendet und Protobuf Bytearrays erstellen und daraus Datenstrukturen erstellen kann, wird ØMQ für die Kommunikation mit Nao verwendet. ØMQ unterstützt Protobuf durch expliziten Hinweis in der Dokumentation (vgl. Hintjens, 2013, S. 16).

Die Verwendung von ØMQ, dass die konventionelle Kommunikation über die Proxyobjekte des SDKs nicht mehr möglich ist, da die Schnittstelle zwischen dem SDK und dem Nao nicht offengelegt ist. Daher wurde ein eigenes Naoqi Modul nötig, dass die Kommunikation über ØMQ und mit Protobuf auf dem Nao ermöglicht.

Die Naoqi API ist nach folgender Struktur aufgebaut: Es gibt Modulnamen wie z.B. ALTextToSpeech und Methodennamen wie z.B. getVolume (s. Formel 4.1). Diese beiden Informationen genügen, um die Methode über den Broker aufzurufen. Daraus könnte abgeleitet werden, eine

statische Abbildung $Message \rightarrow Method$ mit einem Dispatcher (switch case) zu realisieren. Es hätten allein für das Modul ALMotion über 80 Methoden mit einem Dispatch versehen werden müssen. Da die API nach einem einheitlichen Schema aufgebaut ist, ist ein reflektiver Ansatz naheliegend. Reflektiv bedeutet, dass die Methoden in diesem Fall über zwei Strings übertragen werden und zur Laufzeit zu einem Methodenaufruf zusammengesetzt und aufgerufen werden (vgl. Colouris et al., 2012, S. 55).

$$
\begin{gathered}
modulname.methodenname \\
ALTextToSpeech.getVolume
\end{gathered}
\tag{4.1}
$$

Für die Parameter verwendet Naoqi generell nur die Basistypen Strings, Integer (32 Bit unsigned), Float (32 Bit signed), Byte und Boolean. Zusätzlich werden Arrays der Basistypen unterstützt. Die Anzahl der Parameter ist durch die Naoqi API festgelegt. Die Methode ALTextToSpeech.getVolume benötigt keinen Parameter. Insbesondere im Bereich der Bewegung gibt es Methoden mit mehreren Parametern. Im Gegensatz dazu benötigt die Methode ALMotion.getCOM (s. Formel 4.2) mehrere Parameter. Damit wird die Position eines Körperteils des Naos abgefragt. Zuerst wird der Name des Körperteils als String angegeben. Danach wird der Koordinatenraum definiert, dazu wird ein Int aus der Menge { FRAMETORSO = 0, FRAMEWORLD = 1, FRAMEROBOT = 2 } gewählt. Außerdem wird ein Boolean benötigt, dass die Verwendung von Sensoren zur Positionsbestimmung aktivieren kann. Für den reflektiven Ansatz bedeutet dies, dass aus einer überschaubaren und konstanten Menge an Typen eine Liste von Parametern nötig ist, um Naoqi Methoden aufzurufen. Naoqi gibt damit keine Naoqi-spezifischen Typen nach außen oder fordert solche durch Parameter in der Naoqi API ein.

$$
ALMotion.getCOM(pname : String, pSpace : Int, usesensors : Boolean) \tag{4.2}
$$

Um einen entfernten Aufruf der Naoqi API über ØMQ zu realisieren, entwickelte David Olszowka das Naoqi Modul HAWActor, das, der Naoqi API nachempfunden, entfernte Methodenaufrufe ermöglicht. Der HAWAktor verwendet den reflektiven Ansatz. Dieser ist unabhängig von Aktualisierungen der Naoqi API. Ein Dispatcher müsste bei jeder Aktualisierung angepasst werden.

Der Rückgabewert wurde bisher noch nicht angesprochen. Viele Methoden, insbesondere im Bereich Bewegung, enthalten gar keinen Rückgabewert. Wenn jedoch ein Rückgabewert vor-

handen ist, dann ist der Typ, wie bei den Parametern, ein Basistyp. Für z.B. ALMotion.getCOM ist der Rückgabewert ein Array von Floats.

Der HAWActor erhält eine serialisierte Request Nachricht (s. Codebeispiel 4.2), die aus Modulname, Methodennamen und Parametern besteht, diese in einen Methodenaufruf umwandelt und den Reply serialisiert und zurücksendet. Der Reply kann auch leer sein. Für den entfernten Methodenaufruf ist eine Protobuf Nachricht definiert mit genau einem Modulnamen, einem Methodennamen und beliebig vielen Parametern.

```
1 message HAWActorRPCRequest {
2   required string module = 1;
3   required string method = 2;
4   repeated MixedValue params = 3;
5 }
```

Codebeispiel 4.2: Request-Nachricht für den HAWAktor

In Naoqi sind die Parameter als union type realisiert. Der HAWAktor verwendet für die Abbildung dieser Parameter den Typ MixedValue (s. Codebeispiel 4.3), der als „überladener Datentyp“ die Zeichenketten, Integer, Float, Byte, Boolean und Array unterstützt. Ein Array kann man dabei als beliebige Wiederholungen von MixedValue auffassen. Überladene Typen sind in C++ sehr effektiv, aber aus Sicht der Typsicherheit nicht gut zu handhaben. Um den gespeicherten Wert verwenden zu können, muss zuerst geprüft werden für welchen Typ der Wert gespeichert wurde. Danach wird der Wert in den Basistyp überführt. Da im Request eine beliebige Anzahl von MixedValue enthalten sein können und MixedValue sich beliebig oft selbst enthalten kann, können verschachtelte Listen von Parametern erstellt werden.

```
1 message MixedValue {
2   optional string string = 1;
3   optional uint32 int = 2;
4   optional float float = 3;
5   optional bytes binary = 4;
6   optional bool bool = 5;
7   repeated MixedValue array = 6;
8 }
```

Codebeispiel 4.3: „überladener Datentyp“ für Parameter und Rückgabewerte

Die Antwort enthält im positiven Fall genau einen Rückgabewert des Typs MixedValue (s. Codebeispiel 4.4). Ein Fehler kann in Form eines Strings angegeben werden. Sollte es eine

Methode nicht geben, wird von einer Fehlernachricht Gebrauch gemacht. Der Rückgabewert ist dann leer.

```
1 message HAWActorRPCResponse {
2   optional MixedValue returnval = 1;
3   optional string error = 2;
4 }
```

Codebeispiel 4.4: Response-Nachricht für den HAWAktor

Wenn der HAWActor mit qibuild kompiliert und vom Naoqi Broker gestartet wurde, ist dieser über das Netzwerk ansprechbar. Der HAWActor nutzt für die Netzwerkkommunikation das Request-reply Pattern von ØMQ, indem dieser einen REP (Reply) Socket anbietet.

Um eine Kommunikation zu ermöglichen, wird der passende ØMQ Socket REQ (Request) benötigt (s. Codebeispiel 4.5). Erstellt wird ein ØMQ Context, der die Sockets verwaltet. Mit dem ØMQ Context wird ein ØMQ Socket des Typs REQ erstellt, der sich zu einem Socket mit einer URL wie z.B. tcp://127.0.0.1:5555 verbindet. Der Request Socket kann somit an den HAWAktor eine HAWActorRPCRequest Nachricht senden und der HAWAktor mit einer HAWActorRPCResponse antworten.

```
1   val context = new ZContext
2   def socket(url: String) = {
3       val sock = context.createSocket(ZMQ.REQ)
4       sock.connect(url)
5       sock
6   }
```

Codebeispiel 4.5: Zugriff auf ØMQ Sockets

Werden die Definitionen der Protobuf-Nachrichten als Datei gespeichert, wird mit dem Protobuf-Compiler die Nachricht in der Zielsprache, wie beispielsweise Java, überführt und kann in das Projekt eingebunden werden. Der Protobuf-Compiler erstellt für jede Nachricht einen Builder, mit dem Instanzen der generierten Klassen erzeugt werden können. Dem Builder werden die Daten mit set Methoden wie setModule(s:String) Stück für Stück übergeben. Dabei verändert die set Methode nicht den Builder, sondern es wird ein neuer Builder erzeugt. Zum Abschluss wird mit der Methode build die Datenstruktur erzeugt und zurückgegeben.

```
1  def request(module: String, method: String,
2        params: List[MixedValue] = Nil) {
3    val param = HAWActorRPCRequest.newBuilder
4        .setModule(module).setMethod(method)
5    for (mixed <- params) {
6      param.addParams(mixed)
7    }
8    param.build
9  }
```

Codebeispiel 4.6: Erstellung eines Requests für den HAWAktor

Parameter werden der Methode request (s. Codebeispiel 4.6) als Liste des Typs MixedValue übergeben. Dort werden die Parameter einzeln dem Builder von HAWActorRPCRequest hinzugefügt. Erstellt werden MixedTypes aus den Scala Basistypen Int, Float, Boolean, Byte und String. In Scala sind die Basistypen Subtypen des Typs AnyVal. Diese haben u.a. die Besonderheit nicht null sein zu können. Aus Kompatibilitätsgründen zu Java gehören Strings nicht dazu, denn diese können null sein. Daher kann der Übergabeparameter array nicht von Any auf AnyVal eingeschränkt werden. Dazu kommt, dass AnyVal noch weitere Typen wie Unit als Subtypen hat. Eine totale Abbildung $AnyVal \rightarrow MixedValue$ ist auch ohne String nicht möglich. So wird für nicht unterstützte Typen eine Exception geworfen.

```
1     implicit def anyToMixedVal(array: Iterable[Any]) = {
2       val mixedVal = MixedValue.newBuilder()
3       for (value <- array)
4         value match {
5           case x: Int => mixedVal.addArray(x)
6           case x: Float => mixedVal.addArray(x)
7           case x: Boolean => mixedVal.addArray(x)
8           case x: Byte => mixedVal.addArray(x)
9           case x: String => mixedVal.addArray(x)
10          case x => throw new UnsupportedOperationException(
11                x.getClass.toString + " is not allowed")
12        }
13      mixedVal.build()
14    }
```

Codebeispiel 4.7: Implizite Konvertierung in MixedValue

Da die Methode anyToMixedVal (s. Codebeispiel 4.7) eine implizite Methode ist, muss diese nicht direkt aufgerufen werden, sondern wird vom Compiler aufgerufen, wenn eine Typkonvertierung nötig ist. Für Arrays ist auf die gleiche Weise eine implizite Methode definiert. Damit ist es möglich die Methode request mit ausschließlichen Scalatypen aufzurufen (s. Formel 4.3). Die Parameterliste wird dabei implizit von List[Any] zu List[MixedValue] konvertiert.

$$val\ req = request(''ALMotion'', ''getCOM'', List(''HeadYaw'', 1, true)) \tag{4.3}$$

Da ØMQ nur Bytes versendet, muss der HAWActorRPCRequest in ein Array von Bytes überführt werden. Darin liegt die Stärke von Protobuf, denn die Konvertierung der Nachricht wird für jede Protobufdatenstruktur mit der Methode toByteArray bereitgestellt.

Die Nachricht, wie HAWActorRPCRequest, hat eine sehr geringe Größe, daher kann sie in einem Frame verschickt werden (s. Formel 4.4). Ein Frame hat keine Größenbeschränkung, allerdings ermöglicht dieser beispielsweise einen Stream (von z.B. Videos), der hier nicht benötigt wird. Als Endmarkierung der Nachricht wird daher die 0 benötigt. Somit genügt es dem Socket das Bytearray von Nachricht req zu übergeben und es wird an die angegeben URL versendet.

$$socket.send(req.toByteArray, 0) \tag{4.4}$$

Die Antwort kann nun über die Methode recv mit Angabe der gleichen Endmarkierung 0 abgefragt werden (s. Formel 4.5). Der Sendevorgang wird in einem Thread in ØMQ verarbeitet. Die Methode send ist daher nicht blockierend. Die Methode recv dagegen ist blockierend.

$$socket.recv(0) \tag{4.5}$$

Die Methode recv hat als Rückgabewert ein Bytearray. Die automatisch generierte Protobufmethode parseFrom(a:Array[Byte]) erstellt aus dem Bytearray eine Protobufdatenstruktur HAWActorRPCResponse (s. Codebeispiel 4.8). Diese enthält die Konstanten error und returnval. Es wird mir der Methode hasError gerüft, ob ein Fehler vorliegt. Diese Methode wird automatisch generiert, da die Konstante error optional ist. Ein Fehler liegt vor, wenn eine Methode nicht existiert oder nicht ausgeführt werden konnte. Im Fehlerfall ist die Fehlermeldung in error gespeichert. Sollte kein Fehler vorliegen, kann mit hasReturnval geprüft werden, ob der optionale Rückgabewert returnval gesetzt ist. Der returnval ist gesetzt, wenn die Methode gefunden wurde, keinen Fehler verursacht hat und als Rückgabetyp in Naoqi kein void gesetzt

ist. Die Nachricht kann auch keinen Wert enthalten. Wenn die Naoqi Methode einen Rückgabewert vom Typ void hat und kein Fehler entstanden ist, sind der Rückgabewert returnval und die Fehlermeldung error nicht gesetzt.

```
1  def answer = {
2    val resp = HAWActorRPCResponse.parseFrom(socket.recv(0))
3    if (resp.hasError) {
4      trace("Error: " + resp.getError)
5    } else if (resp.hasReturnval) {
6      trace("-> " + resp.getReturnval)
7    } else {
8      trace("-> Empty \n");
9    }
10 }
```

Codebeispiel 4.8: Überführung eines Bytearrays in eine Response Nachricht

Da Naoqi keine Komprimierung für Videodaten unterstützt und diese Komprimierung nötig für eine flüssige Videodarstellung ist, entwickelte David Olszowka das Naoqimodul HAWCamServer. Dieser ist ähnlich wie der HAWAktor aufgebaut. Dieser versteht die Request-Nachricht CamRequest (s. Codebeispiel 4.9), die die Konfiguration von Auflösung, Farbbereich und Framerate pro Sekunde zulässt. HAWCamServer verschickt darauf eine Antwort mit genau einem Bild. Die auf JPEG basierte Kompression kann nicht konfiguriert werden, da diese im HAWCamServer fest implementiert ist. Es wird genau ein Bild verschickt, da sonst das Request-reply Pattern nicht eingehalten wird.

```
1 message CamRequest {
2  optional uint32 resolution = 1;
3  optional uint32 colorSpace = 2;
4  optional uint32 fps = 3;
5 }
```

Codebeispiel 4.9: Request-Nachricht für den CamServer

Der HAWCamServer sendet als Antwort einen CamResponse (s. Codebeispiel 4.10). Diese Nachricht enthält das konfigurierte Bild vom Form von Bytes. Bei einem internen Varbeitungsfehlers im HAWCamServer ist außerdem der Fehlerstring error gesetzt. Beispielsweise können Fehler beim Abrufen des Bildes in Naoqi verursacht werden.

```
1 message CamResponse {
2   optional bytes imageData = 1;
3   optional string error = 2;
4 }
```

Codebeispiel 4.10: Response-Nachricht für den CamServer

Die Nachricht CamRequest enthält ausschließlich optionale Werte. Dadurch ist eine Nachricht möglich, die keine Werte enthält. Der HAWCamServer verwendet die CamRequest zur Konfiguration, sofern mindestens ein Wert enthalten ist. Zurückgeschickt wird ein CamResponse, der ein leeres Bild enthält. Wenn der CamRequest Werte enthält, die zu einem Fehler führen, wird die Fehlernachricht error gesetzt. Nur wenn die Nachricht CamRequest leer ist, wird ein Bild zurückgesendet. Es muss zuvor mindestens einmal die Konfiguration gesendet werden. Die Konfiguration kann danach durch eine erneuten CamRequest verändert werden.

4.4 Kommunikationssequenzen im Aktorensystem

Das Aktorensystem naogateway ermöglicht die Kommunikation mit dem Nao im Aktorenmodell. Dazu bietet es an die Naoqi API zu benutzen, außerdem stellt es auch eine spezielle API für den Zugriff auf komprimierte Kameradaten bereit. Dafür verwendet es die zuvor beschriebene Kommunikation mit dem Nao. In Aussicht steht eine API für Audio und die Naoqi eigene Datenbank ALMemory (ein Key-Value Store).

Das Aktorensystem naogateway (Abb. 4.1 Aktorensystem naogateway) hat verschiedene Dienste, die es abbildet. Jeder Dienst ist ein Aktor. Ein Methodenaufruf in der Naoqi API benötigt einen Aufruf und gibt eine Antwort zurück. Diese Request-reply Kommunikation wird durch einen ResponseActor umgesetzt. Der dagegen NoResponseActor verwirft jegliche Antwort, sodass der Entwickler beide Varianten zur Wahl hat und selbst entscheiden kann, ob die Antwort verworfen werden soll. Die Aktoren prüfen nicht, ob die Naoqi Funktion einen verwertbaren Rückgabewert hat oder nicht. Der VisionActor ermöglicht die Abfrage von komprimierten Bildern von der Kamera des Nao. AudioActor und MemoryActor sind derzeit in der Entstehung.

Erschafft und überwacht werden diese Aktoren vom NaoActor. Es existiert der HeartBeatActor, um zu überprüfen, ob der Nao zur Zeit erreichbar ist. In einem fest definierten Intervall

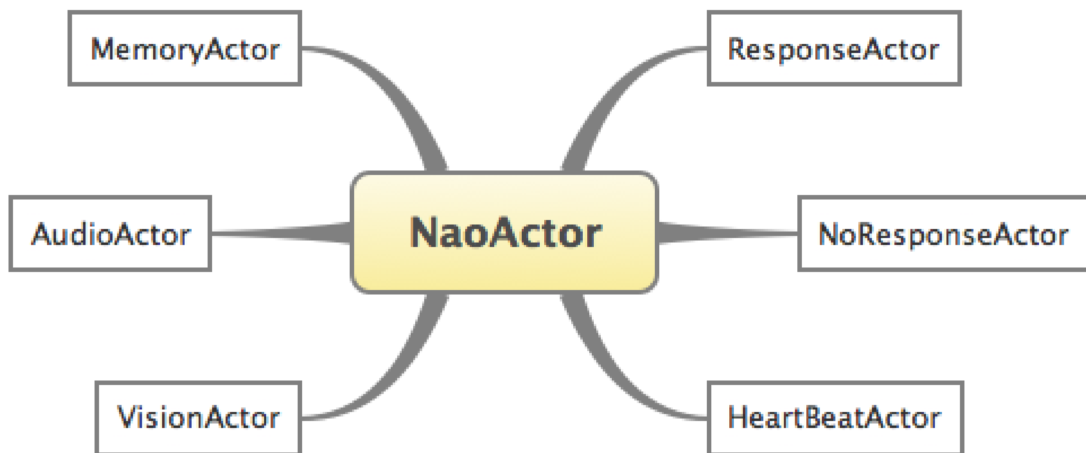

Abbildung 4.1: Aktorensystem naogateway

werden vom HeartBeatActor Testnachrichten geschickt, die den einzigen Zweck haben eine Antwort zu generieren. Ist eine Antwort gekommen, ist sichergestellt, dass der Nao in diesem Moment erreichbar ist. Der NaoActor erhält vom HeartBeatActor Nachrichten, wenn sich der Verbindungsstatus ändert.

Initialisiert wird der NaoActor (Abb. 4.2 Initialisierung) beim Start des Aktorensystems. Damit ist sichergestellt, dass es genau einen NaoActor erstellt wird. In der Konfiguration sind die Zugangsdaten hinterlegt (s. Codebeispiel 4.11). Die Zugangsdaten sind verpackt in einer case class Nao, die den Namen, Host und Port in sich trägt. Diese kann der HeartBeatActor direkt nutzen um mit Testnachrichten zu prüfen, ob der Nao derzeit erreichbar ist. Im Positivfall werden die Kinder gestartet und denen die Zugangsdaten übermittelt. Da ein ØMQ Context von verschiedenen Aktoren nicht gleichzeitig angesprochen werden kann, benötigt jedes Kind seine eigene ØMQ Context Instanz.

```
1 nila {
2     nao.host = "192.168.1.10"
3     nao.name = "Nila"
4     nao.port = 5555
5 }
```

Codebeispiel 4.11: Zugangsdaten des Naos

Der erste Gedanke war, den Aktoren die Zugangsdaten durch Message Passing mitzuteilen. Mit dieser Nachricht gehen die Aktoren in einen neuen Zustand über, in dem sie ihre eigentliche Aufgabe ausführen können. Da dadurch die Möglichkeit bestand, dass ein Aktor eine Nachricht schickt, bevor die Aktoren die Zugangsdaten erhalten haben, ergab sich die Gefahr, dass Nachrichten nicht korrekt verarbeitet werden. Um dieses Problem zu lösen, werden die Daten über den Konstruktor übergeben.

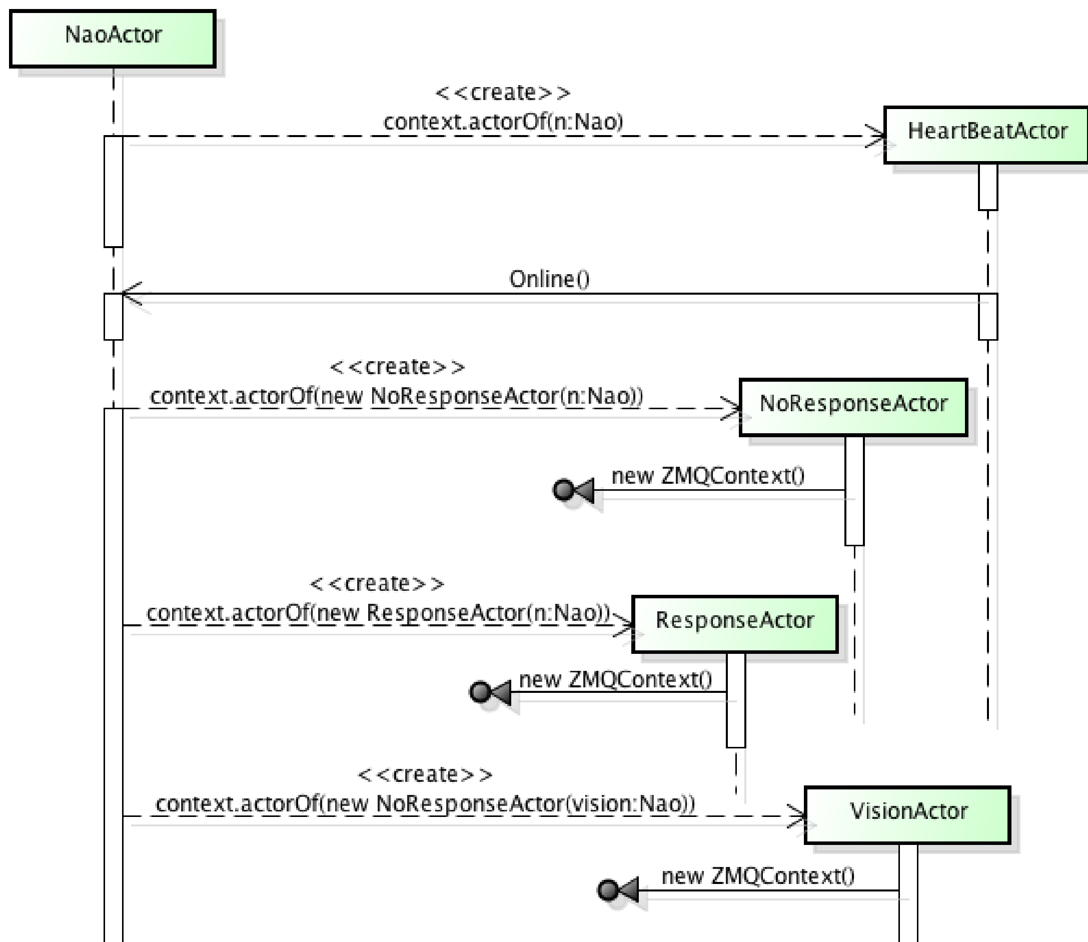

Abbildung 4.2: Initialisierung

Ist der NaoActor mit seinen Kindern initialisiert, kann dem NaoActor eine Connect Nachricht geschickt werden, um die Akorreferenzen seiner Kinder zu erhalten. Der NaoActor antwortet mit dem Tripel (reponseActorRef, noResponseActorRef, visionActorRef). Das Tripel enthält die Aktorreferenzen von dem NoResponseActor, dem ResponseActor und dem VisionActor; diese können verwendet werden, um an einen ausgewählten Aktor einen Request zu schicken. Die

drei Aktoren kann man als Worker auffassen, die auf Anfrage die Kommunikation zwischen einem beliebigen Aktor und dem Nao realisieren.

Die Anfrage an den ResponseActor und den NoResponseActor ist die Nachricht Call (s. Formel 4.6). Call kapselt die Protobuf-Nachricht HAWActorRPCRequest. Der Call enthält einen Modulnamen, einen Methodennamen und eine Liste von Parametern als MixedValue. In den beiden Aktoren wird der Call in die Protobuf-Nachricht umgwandelt und an den Nao geschickt.

$$responseActor\ !\ Call('ALTextToSpeech,' getVolume) \tag{4.6}$$

Außerdem gibt es die Nachricht Answer, die den Rückgabewert HAWActorRPCResponse und den Call enthält. Die Nachricht Answer enthält den Call, um eine Identifizierung zu ermöglichen. Wenn ein beliebiger Aktor einem der drei Worker mehr als einen Call schickt, ist nicht geregelt, wann die jeweilige Antwort zurückkommt. Eine Antwort könnte auch vollständig ausfallen. Da Aktoren nebenläufig arbeiten, kann keine Aussage über das Eintreffen der Nachricht getätigt werden. Sollte ein Fehler in der Protobuf-Nachricht sein, da z.B. die Methode nicht gefunden wurde, wird eine InvalidAnswer zurückgeschickt. Durch den Call ist die Antwort eindeutig, sofern verschiedene Calls versendet werden. Dann kann im Patternmatching (s. Formel 4.7) der case class Answer der Call angegeben werden. Für jeden Call kann nun ein eindeutiges Patternmatching durchgeführt werden.

$$\begin{aligned} case\ Answer(Call('ALTextToSpeech,' getVolume), value) \\ case\ Answer(Call('ALTextToSpeech,' say, List("HelloWorld")), value) \end{aligned} \tag{4.7}$$

Der HAWAktor nutzt strikt das Request-reply Pattern, d.h. wenn ein Aktor ein ØMQ Socket nutzt um den Call zu schicken, darf der Socket nicht direkt genutzt werden um den nächsten Call zu schicken (Abb. 4.3 Request-reply Kommunikation mit dem ResponseActor). Es können zwar ohne weiteres viele Sockets benutzt werden, es muss aber auch sichergestellt sein, dass die Antwort auf einen Call zunächst verarbeitet wird und danach der Socket entweder weiterverwendet oder explizit geschlossen wird. Der Socket muss gechlossen werden, da sonst die Zahl der Sockets mit jedem Call ansteigt. Gleichzeitig darf der Aktor nicht blockieren, da er sonst nicht mehr erreichbar wäre. Blockieren kann der Aktor beispielsweise durch die Methode recv. Wenn eine Nachricht nicht ankommt und ein Aktor durch recv blockiert, reagiert der Aktor nicht mehr auf neue Nachrichten.

Die erste Idee, war einen eigenen Aktor zu verwenden, der nur Nachrichten an den Nao schickt und empfängt. Dieser Aktor wird für jeden Call instanziiert und, nachdem die Antwort weitergeleitet wurde, wieder geschlossen. Das Blockieren wird dann in diesen neuen Aktor delegiert. Man hätte dazu einen Pool von Aktoren anlegen müssen um die Zahl der Aktoren unter Kontrolle zu halten.

Die Lösung war unnötig komplex, denn für solche überschaubaren Aufgaben sind Futures ausreichend. Das Future erhält als Funktion das Warten auf die Antwort, die Umwandlung in die Protobuf-Datenstruktur und den Absender, um im Erfolgsfall die Antwort zurückzuschicken. Der ØMQ Socket wird danach geschlossen. So wird das Request-reply Protokoll korrekt eingehalten ohne zu blockieren.

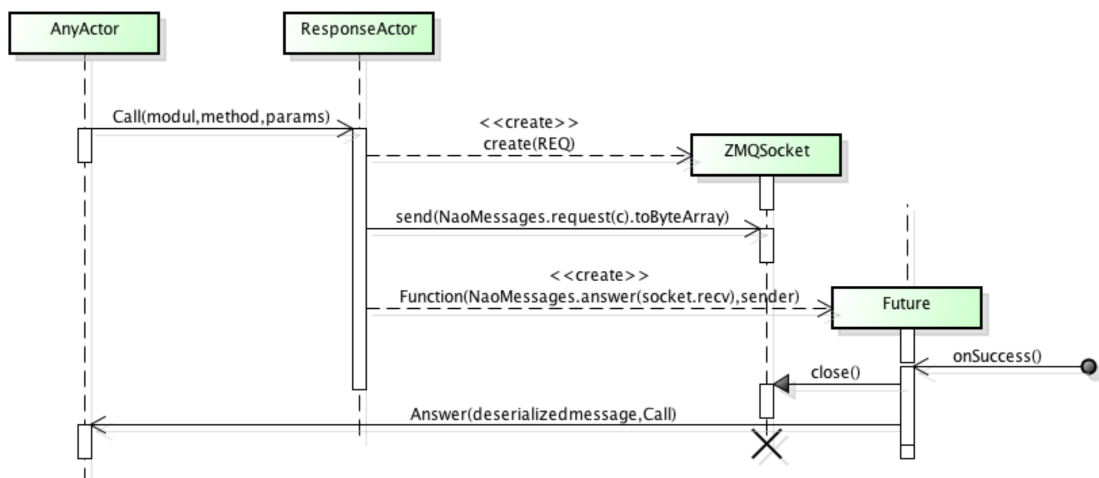

Abbildung 4.3: Request-reply Kommunikation mit dem ResponseActor

Das Senden eines Calls ohne eine Antwort zu erhalten erfolgt über den NoResponseActor (Abb. 4.4 Vorgetäuschte unidirektionale Kommunikation). Der NoResponseActor muss auch das Request-reply Protokoll korrekt einhalten ohne zu blockieren, denn dieser kommuniziert auch mit dem HAWAktor. Der NoResponseActor funktioniert ähnlich wie der ResponseActor. Er wartet auf einen Call, wandelt diesen in ein HAWActorRPCRequest um und schickt diese Nachricht an den Nao. Danach wartet NoResponseActor auf die Antwort. Die Antwort wird jedoch nicht an den Absender des Calls weitergereicht.

Es stellt sich an dieser Stelle die Frage, wie der NoResponseActor mit ankommenden Calls verfährt, während eine Antwort auf den vorherigen Call noch aussteht. Eine erste Lösung

war folgende: Der Aktor verfügt über eine Mailbox, aus der der Aktor nicht nur Nachricht herausnehmen kann, sondern diese auch nach dem Herausnehmen zur Seite gestellt werden können. Dies wird Stashing genannt. Die Nachrichten können nicht direkt in die Warteschlange gelegt werden, da sonst eine Endlosschleife für Stashing entstehen würde.

Hinzu kommt, dass der NoResponseActor durch Stashing zwei Zustände benötigt: Im Zustand communicating empfängt er einen Call und erstellt einen Future für die Antwort. Danach geht er in den Zustand waiting über, indem jeder Call mit Stashing zur Seite gelegt werden. Wenn die Antwort gekommen ist, werden alle Nachrichten, die zur Seite gelegt wurden, wieder in die normale Nachrichtenschlange geschoben. Die Antwort wird nicht weitergeleitet. Verwendet wurde dazu immer nur ein ØMQ Socket. Es kann jedoch erst wieder eine Nachricht gesendet werden, wenn die Antwort erhalten wurde, wodurch der NoResponseActor weniger Calls verarbeiten konnte als der ResponseActor.

Da ØMQ Sockets keine echten TCP Sockets sind und das Erstellen und Schließen daher keine aufwendige Operation ist, kam eine andere Lösung zum Zuge. Für jeden Call wird ein ØMQ Socket verwendet, der danach wieder geschlossen wird. In einem Geschwindigkeitstest von 1000 Calls hintereinander zeigte sich, dass die Verarbeitung signifikant schneller wurde. Der Grund liegt in der Verarbeitung der Answer. Für das Schicken des Calls ist es irrelevant, welche Answer noch nicht verarbeitet wurde. Dadurch vereinfachte sich der Aktor.

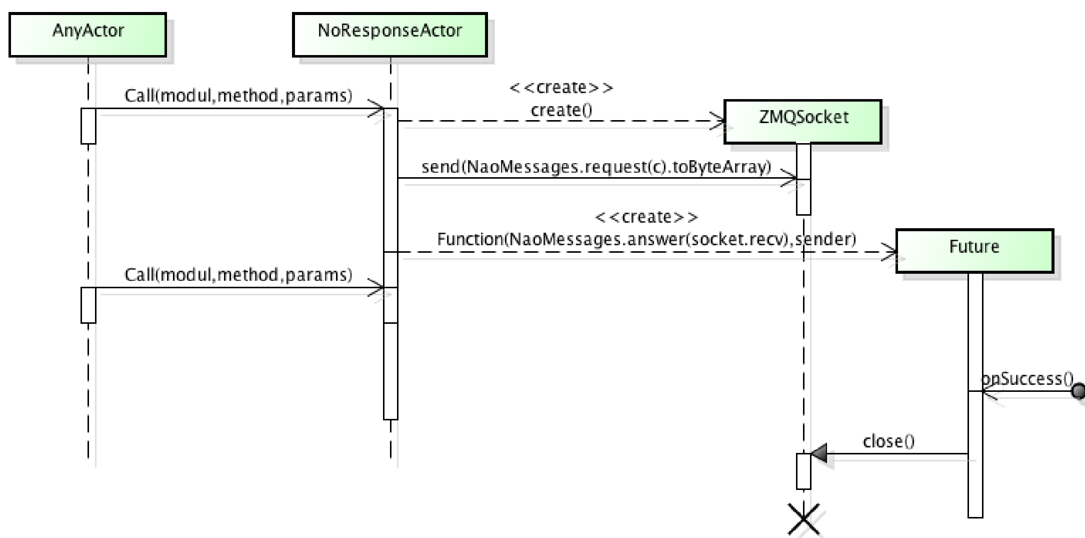

Abbildung 4.4: Vorgetäuschte unidirektionale Kommunikation

Für die Bildübertragung ist VisionActor zuständig. Der VisionActor ist im Grunde aufgebaut wie ein ResponseActor, jedoch verwendet er auf eigene Protobuf-Nachrichten. Der VisionActor kann auf Anfrage einzelne Bilder liefern. Analog zum Call gibt es für Bilder die Nachricht VisionCall (s. Codebeispiel 4.13). VisionCall kapselt den CamRequest, indem es Auflösung, Farbbereich und Framerate pro Sekunde speichert. Diese sind als Enumeration implementiert, da jeweils nur eine sehr geringe Anzahl an Wert möglich ist. Es werden vier Auflösungen, sechs Farbräume (s. Codebeispiel 4.12) und 1-30 Frames pro Sekunde unterstützt. Die Bezeichnungen wurde von der Naoqi API übernommen.

```
1 object ColorSpaces extends Enumeration {
2   type ColorSpace = Value
3   val kYuv = Value(0)
4   val kYUV422 = Value(9)
5   val kYUV = Value(10)
6   val kRGB = Value(11)
7   val kHSY = Value(12)
8   val kBGR = Value(13)
9 }
```

Codebeispiel 4.12: Farbbereiche des HAWCamServers

Wird ein VisionCall an den VisionActor geschickt, wandelt er den VisionCall in eine Cam-Request Nachricht um. Antwortet der HAWCamServer mit einem CamResponse, schickt der VisionActor eine leere CamRequest Nachricht. Der darauf folgende CamResponse enthält ein Bild, dass an den Sender des VisionCalls weitergeleitet wird. Danach geht der VisionActor in einen neuen Zustand über, in dem er weiterhin einen VisionCall versteht, jedoch auch die Nachricht Trigger. Wird dem VisionActor eine Nachricht Trigger gesendet, sendet er eine leere CamRequest Nachricht an den HAWCamServer und leitet die Antwort CamReponse an den Sender der Trigger Nachricht.

```
1 VisionCall(resolution:Resolutions.Value,
2            colorSpaces:ColorSpaces.Value,
3            fps:Frames.Value)
```

Codebeispiel 4.13: Nachricht VisionCall für die Anfrage an den VisionActor

Der HeartBeatActor soll in einem bestimmten Intervall erneute Testnachrichten schicken, um den Nao nicht unnötig zu belasten. Dafür wird auch ein Timer benötigt. Es sollen auch der ResponseActor und der NoResponseActor in der Lage sein eine Verzögerung für bestimmte

Nachrichten einzuleiten. Beispielsweise ist es nicht sinnvoll, innerhalb von wenigen Millisekunden den Arm nach oben und wieder nach unten zu bewegen. Der Nao kann Bewegungen in der gegebenen Zeit nicht ausführen. Auch hierfür ist ein Timer nötig, um eine Verzögerung zu realisieren.

Man hätte diese Timer auch in die Aktoren verlagern können, die den ResponseActor oder NoResponseActor benutzten. Wenn in einer neuen Version des Naoqis die Verzögerung unterstützt wird, wäre die Anpassung sehr aufwändig geworden. So müssen nur der ResponseActor und NoResponseActor angepasst werden.

Ein nicht blockierender Timer kann mit Hilfe eines Futures realisiert werden (s. Codebeispiel 4.14). Akka unterstützt dies direkt durch das Pattern after. Der Funktion after werden eine Zeitangabe (Duration) und ein ExecutionContext (z.B. der Akka Schedular) übergeben. Als Aufgabe wird definiert: eine Trigger Nachricht an sich selbst zu schicken. Der ExcutionContext führt nach Ablauf der Zeitangabe die übergebene Aufgabe aus. Die Trigger Nachricht an sich selbst zu schicken ermöglicht ein Verhalten, je nach aktuellem Zustand, zu variieren. Der Timer ist vom Zustand des Aktors unberührt.

```
1  import akka.pattern.after
2  class TimerActor extends Actor {
3        def receive = {
4          case TimeOut => /* do something */
5          case _ => after(2000 millis,
6                using = context.system.scheduler){
7               Future{
8                 self ! Trigger
9               }
10            }
11       }
12 }
```

Codebeispiel 4.14: Verwendung des after pattern zum zerzögernden Ausführen

Die Verzögerung für den ResponseActor und den NoResponseactor wird durch das trait Delay gelöst. Darin wird die Konfiguration des Namespaces responseactor.delay geladen (s. Codebeispiel 4.15). In diesem Namespace werden die Verzögerungszeiten in Millisekunden angegeben. Da die Konfiguration nur Kleinbuchstaben zulässt, werden Modulnamen und Methodennamen vollständig kleingeschrieben.

```
responseactor.delay {
      almotion.setposition = 200
}
```

Codebeispiel 4.15: Konfigurationsbeispiel für Verzögerungen

Eine Methode delay, der Methodenname und Modulname übergeben werden, sucht die dazu passende Konfiguration und gibt die Verzögerung in Millisekunden zurück. Sollte eine Methode in der Konfiguration nicht definiert sein, wird eine Verzögerung von 0 zurückgegeben. Der Request wird durch das Pattern after erst nach der Verzögerung an den Nao geschickt.

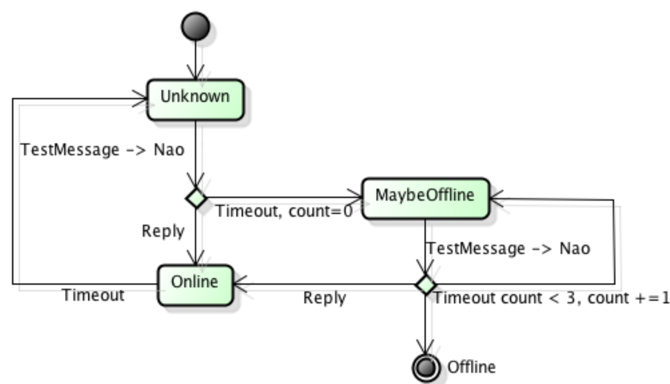

Abbildung 4.5: Heartbeat Zustandsautomat

Der HeartBeatActor testet die Verbindung zum Nao. Der HeartBeatActor versendet dazu Testnachrichten in einem über die Konfiguration definierten Timeout. Wenn der Nao auf die Testnachrichten antwortet, ist dieser erreichbar (Abb. 4.5 Heartbeat Zustandsautomat). Als Testnachricht wird Call('test, 'test) verwendet. Modul und Methodenname sind frei wählbar. Wichtig ist, dass ein Modulname und ein Methodenname angegeben sind. Kommt eine Antwort innerhalb des Timeouts, wird der Nao als erreichbar angenommen, wobei nach einem Timeout dieser Zustand wieder überprüft wird. Sollte auf Grund einer Testnachricht ein Timeout erfolgen, vermutet der HeartBeatActor, dass der Nao nicht erreichbar ist. Dies wird durch den Zustand MaybeOffline abgebildet. Es werden jedoch drei Chancen eingeräumt. Wird das Timeout drei Mal überschritten werden, geht der HeartBeatActor von einem nicht erreichbaren Nao aus und terminiert. Der NaoActor kann nun den HeartBeatActor neustarten. Das Starten und Neustarten erfolgen, indem eine Trigger Nachricht an den HeartBeatActor gesendet werden. Zum Verschicken der Testnachrichten verwendet der HeartBeatActor einen eigenen

ResponseActor. Es ist ein eigener ResponseActor, um die Last auf dem normalen ResponseActor zu reduzieren.

Prinzipiell kann der HeartBeatActor auch direkt einen ØMQ Socket nutzen. Dies bedeutet, dass das Pattern after direkt auf einen ØMQ Socket angewendet wird. Ein ØMQ Socket wird von ØMQ auf deren native Bibliothek abgebildet. Im Test gab es in der nativen Bibliothek zmq Fehler, sobald das Timeout erreicht wurde. Wird die Kommunikation in einem Aktor gekapselt, tritt der Fehler nicht auf. Daher wird auf einen eigenen ResponseActor gesetzt.

Der Zustandsübergang des HeartBeatActors erfolgt über eine einzelne Methode step. Der Methode step werden sowohl für den positiven als auch für den negativen Fall der Nachfolgezustand als partielle Funktion übergeben. Außerdem wird für die übergebenen Nachfolgezustände eine Nachricht angegeben. Die Methode step schickt daraufhin die Testnachricht mit der Methode ask (Fragezeichenoperator) an den ResponseActor (s. Codebeispiel 4.16). Dabei wird implizit ein Timeout angegeben.

```
1  implicit val timeout = Timeout(d)
2  val answering = response ? c
```

Codebeispiel 4.16: Message Passing mit Timeout

Wird eine Nachricht innerhalb des Timeouts vom ResponseActor zurückgeschickt, wird onSuccess ausgeführt (s. Codebeispiel 4.17). Darin wird die erwartete Antwort verarbeitet. Die Antwort an sich ist irrelevant. Entscheidend ist nur, dass eine Antwort zurückgeschickt wird. Als erstes wird die positive Nachricht an den caller geschickt. Der caller ist standardmäßig der NaoActor. Danach wird der Timer mit einem Timeout aktiviert, dass in der Konfiguration für den positiven Fall (on) definiert ist. Das Timeout reduziert die Anzahl an Testnachrichten, da sonst eine unnötige Last erzeugt wird. Am Ende wird mit become in den neuen Zustand übergegangen. Der Negativfall onFailure verläuft analog zum Positivfall. Es wird die negative Nachricht an den NaoActor geschickt, die Verzögerung (off) dem Timer übergeben und in den Nachfolgezustand für den negativen Fall übergegangen.

```
answering onSuccess {
    case _ => {
      caller ! succMessage
      delay(on)
      become(suc)
    }
}
answering onFailure {
    case _ => {
      caller ! failMessage
      delay(off)
      become(fail)
    }
 }
```

Codebeispiel 4.17: Reaktion auf Timeout des Futures

Der NaoActor erhält durch den HeartBeatActor regelmäßige Statusinformationen und kann darauf reagieren. Zunächst wurde bei jedem Step die Statusnachricht an den NaoActor gesendet. Allerdings ist es für den NaoActor nur wichtig, ob der Nao in den Zustand online oder offline übergangen ist. Wenn der Nao online ist, kann der NaoActor in den Zustand communicating übergehen, indem er die Connect Nachricht versteht und die Referenzen seiner Kinder nach außen gibt. Sollte der Nao offline sein, muss der NaoActor alle vorhandenen Sockets schließen, damit angestaute Nachrichten nicht unkontrolliert ausgeführt werden, wenn der Nao wieder erreichbar ist. Dazu wirft der Nao eine RuntimeException. Der übergeordnete User Guardian von Akka startet per Standardkonfiguration den NaoActor neu und zerstört dabei alle Sockets.

Durch den Neustart geht der NaoActor in den Startzustand über. In diesem Zustand schick der NaoActor eine initiale Trigger Nachricht an den HeartBeatActor. HeartBeatActor versucht wiederum Testnachrichten zu schicken. Ist dies möglich, erhält der NaoActor die Nachricht Online und kann nun wieder in den Zustand communicating übergehen. Zustände, wie Maybeoffline, sind für diesen Ablauf unerheblich und können daher gespart werden. Damit wird auch die Last für den NaoActor reduziert.

4.5 Verteilung

Die Kommunikation zwischen dem Nao und dem Aktorensystem ist die Schlüsselstelle zur Steuerung des Naos. Wenn Applikationen entwickelt werden, die den Nao steuern, ist für die Kommunikation mit dem Nao die Komponente naogateway nötig, der als Service für das Restsystem zur Verfügung steht. Der naogateway ist ein Aktorensystem, das unabhängig von den anderen Komponenten auf einem Rechner läuft (Abb. 4.6 Verteilungssicht). Der naogateway wird für einen in der Konfiguration hinterlegten Roboter, wie nila (s. Codebeispiel 4.11), gestartet. Dabei wird der NaoActor gestartet und mit dem Namen des Roboters belegt (s. Formel 4.8). Die Applikationen können dann über den naogateway mit dem Nao kommunizieren.

$$akka : //naogateway/users/nila \tag{4.8}$$

Im Rahmen des Projekts ist eine AndroidApp, die das Kamerabild des Naos darstellt und eine Bewegungsteuerng ermöglicht, entstanden. Es ist eine Desktop Applikation zur Steuerung mit der Tastatur oder dem XBOX Controller entwickelt worden. Außerdem ein Watchdog, der Statusinformationen des Naos, wie Batteriezustand, überwacht. Die Idee ist es, dass alle Applikationen das Aktorensystem RobotMiddleware verwenden. RobotMiddlware stellt unabhängig vom Nao Operationen wie Laufen bereit, die intern in Call Nachrichten ungewandelt werden um mit dem naogateway zu kommunizieren. Applikationen sind unabhängig voneinander

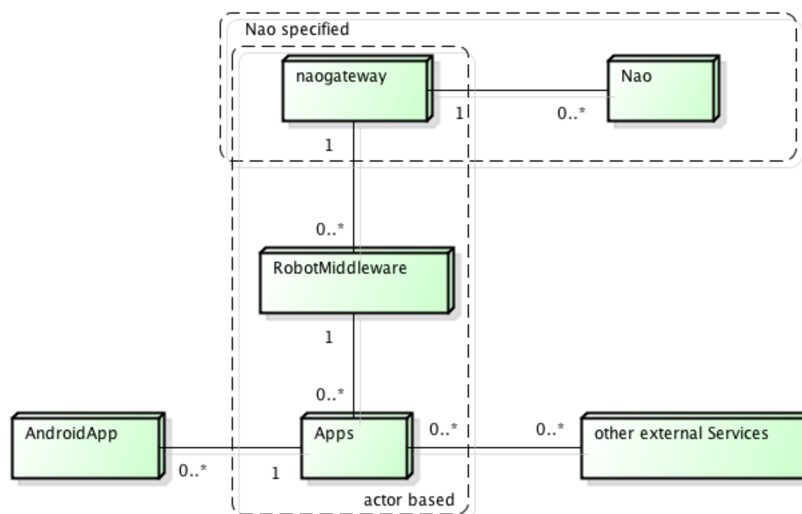

Abbildung 4.6: Verteilungssicht

agierende Softwarepakete, die die Anwendungsfälle abbilden und dabei RobotMiddleware benutzen. Jedoch können diese auch den Zugang zu externen Diensten erlangen, ohne dabei RobotMiddleware oder naogateway zu berühren.

Die RobotMiddleware bietet ein trait RobotInterface, welches die hochsprachlichen Methoden wie Laufen bereitstellt und beim Aufruf die Methoden in Nachrichten umsetzt, die über den eigenen Aktor NaoTranslator an naogateway delegiert werden, dort in Naoqi spezifische Aufrufe umgewandelt werden und zu Nao geschickt werden. Die Antwort wird über naogateway wieder zum NaoTranslator zurückgeleitet (Abb. 4.7 Kommunikationsschema zwischen den Aktorensystemen).

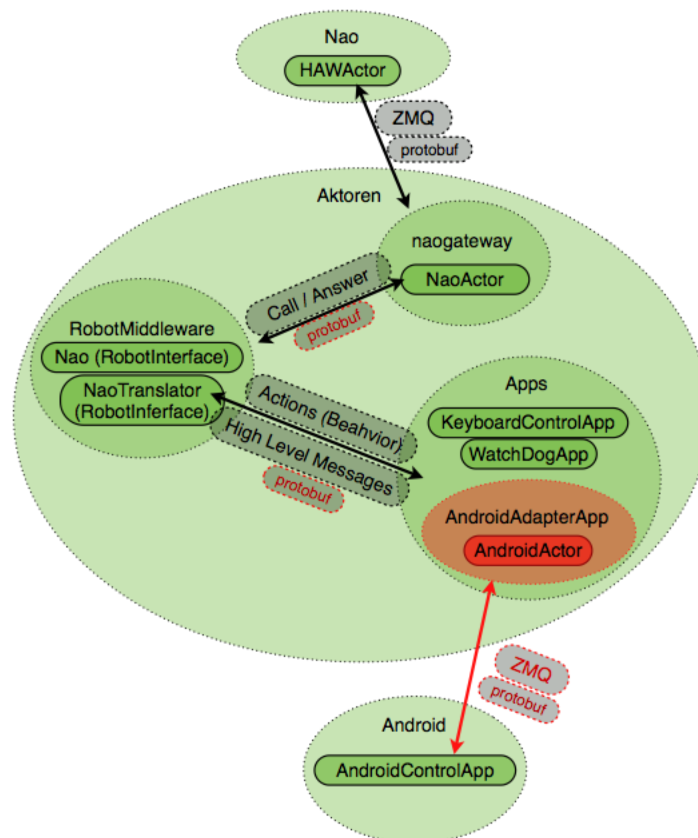

Abbildung 4.7: Kommunikationsschema zwischen den Aktorensystemen

Applikationen agieren in einem eigenen Aktorensystem, um völlig unabhängig zu sein. Die Apps haben die Möglichkeit, auf die NaoInterface API zuzugreifen und gleichzeitig externe Dienste zu nutzen, um eine vollwertige App zu schaffen, die ein Nao steuern kann. Die An-

droidApp benötigt eine Serverapplikation, da sie selber nicht Teil des Aktorensystems ist. Die AndroidApp kommuniziert ausschließlich über einen ØMQ Socket. Die Idee ist, Nachrichten vollständig über Protobuf abzubilden und dabei von speziellen Roboterschnittstellen, wie Naoqi, zu abstrahieren. Akka unterstützt nativ Protobuf als Serialisierer, sodass sich dieses sich ins System einbinden lässt. Anzumerken ist, dass bisher nur die Kommunikation zwischen dem naogateway und dem Nao über Protobuf erfolgt. Die AndroidApp kommuniziert derzeit direkt über ØMQ mit dem Nao. In der Abbildung 4.7 sind die noch nicht umgesetzten Bereiche rot markiert.

4.6 Strukturierung auf Dateisystemebene

Der naogateway ist in seiner Ordnerstruktur wie folgt aufgebaut:

- README.md - Die Readme enthält eine Anleitung zum Installieren in einer Ubuntu VM (mit allen nötigen Abhängigkeiten), außerdem eine Kompilierungsanaleitung, Startparameter und Konfigurationsbeispiele.
- communication - Der Ordner enthält Sequenzdiagramme für die Kommunikation. Außerdem ist der Heartbeat als Zustandsdiagramm dargestellt.
- scaladoc - In diesem Ordner ist das generierte Scaladoc, dass ähnlich dem Javadoc Klassen- und Methoden dokumentiert, zu finden.
- hosted.conf - Dies ist ein Konfigurationsbeispiel um mit dem naogateway, der in der Virtual Box VM ausgeführt wird, vom Host aus zu kommunizieren .
- build.sbt - Diese Datei enthält die Konfiguration des Simple Build Tools für die Kompilierung des naogateway.
- src/main/ressources/application.conf - Die naogateway Standardkonfiguration wird in dieser Datei geispeichert.
- src/main/scala - Im Ordner scala befindet sich der Quellcode. In scala befinden sich dazu die Quellcode-Ordner. In den Quellcode-Ordnern befinden sich die Packages.
    - naogateway - In Quellcode-Ordner naogateway befinden sich die Aktoren und die Mainclass des Naogateways.
        - naogateway - Die Aktoren befinden sich im Package naogatway. Die Scala traits für Aktoren (z.B. Anbindung von zmq) werden im Package naogateway.traits gespeichert.

    * test - Beispiele um einzelne Aktoren zu testen, befinden sich im Package test.
        · simple - Lokale Tests für einzelne Aktoren im naogatway sind im Ordner simple abgelegt.
        · remote - Tests for die Kommunikation zwischen zwei JVMs befinden sich im Ordner remote.
        · timer - Ein Test für Timeouts ist im Ordner timer.
        · directZMQ - Tests für die Kommunikation mit dem Nao ohne Aktoren befinden sich in dem Ordner directZMQ.
  – naogatewayValue - Verwendete Nachrichten für die Kommunikation mit dem naogateway befinden sich ausschließlich in Quellcode-Ordner naogatewayValue.
    * naogateway/value - Die Nachrichten sind im Package naogateway.value organisiert.
  – zeromq - Der jzmq Adapter zum Ansprechen der Bindings wird als Quelltext ausgeliefert und ist im Quellcode-Ordner zeromq abgelegt.

Da SBT (vgl. Typesafe Simple Build Tool, 2013) zum Kompilieren verwendet wird, ist ein Teil der Ordnerstruktur vorgegeben. Der Quellcode muss sich in src/main/scala oder src/main/java befinden. Der Ordner src/test ist für Testcode gedacht. Allerdings sind beim Testen die Imports aus dem src/main Ordner nicht auflösbar gewesen, sodass der Testcode in src/main/scala gespeichert wurde. SBT kompiliert Java und Scala, kann Quellcode und Classdateien in jar Dateien packen oder daraus ein Scaladoc generieren. Außerdem integriert es Maven-Repositories, die es einem erlauben Abhängigkeiten dynamisch nachzuladen. Wichtig ist dafür die Konfigurationsdatei sbt.build (s. Codebeispiel 4.18). In dieser muss der Projektename, eine Version, der Scalacompiler und die mainClass definiert werden. Außerdem können die Abhängigkeiten, sofern vorhanden, definiert werden.

```
1 name := "naogateway"
2 version := "1.0"
3 scalaVersion := "2.10.1"
4 mainClass in (Compile, run) := Some("naogateway.NaogatewayApp")
5 libraryDependencies += "com.typesafe.akka"
6       % "akka-actor_2.10" % "2.1.2"
7 libraryDependencies += "com.typesafe.akka"
8       % "akka-remote_2.10" % "2.1.2"
```

Codebeispiel 4.18: build.sbt - SBT Konfigurationsdatei

Wenn SBT auf einem Rechner installiert ist, genügt im Projektverzeichnis ein einziger Befehl (s. Formel 4.9) zum Kompilieren und Starten. Statt des Befehls run kann auch compile zum Compilieren und doc zum Generieren des Scaladocs verwendet werden.

$$sbt\ run \tag{4.9}$$

Es gibt mehrere Ordner für den Quellcode. Zur Kommunikation mit dem naogateway ist nur der Ordner naogatewayValue nötig, denn dort sind alle Nachrichten zur Kommunikation mit dem naogateway gespeichert. Damit sind die Abhängigkeiten für die Nutzer des naogateways von der Implementierung der Aktoren getrennt. Erst in den Quellcodeordnern ist die Packagehierachie abgebildet.

5 Fazit und Ausblick

Im Nachhinein sind folgende Verbesserungsmöglichkeiten aufgefallen. Ein wichtiger Punkt sind die verwendeten Bibliotheken. Eingesetzt werden die Java Bindings für ØMQ, da diese schon länger verfügbar sind. Die ersten Committs der Scala Bindungs sind weniger als ein Jahr alt. Langfristig sind die Scala Bindings die klügere Wahl, da die restliche Implementierung in der gleichen Sprache geschrieben ist und zusätzlich die Bindings sehr sehr kompakt gehalten sind. Es gibt auch eine Scala-Variante für Protobuf, die noch sehr jung ist und im Test nicht funktioniert hat, weswegen die Java-Variante verwendet wird. Auch hier macht es Sinn weiter zu testen, denn die Scalavariante ist nicht nur sehr kompakt, sondern auch in der Lage Besonderheiten von Scala zu verwenden, wie beispielsweise implizite Methoden.

Der HAWAktor und auch der HAWCamServer entstanden unabhängig von dieser Bachelorarbeit. Diese mussten, so wie sie auf dem Nao installiert sind, verwendet werden. Das Request-reply Pattern des HAWCamServer und des HAWCamServer macht die Kommunikation an manchen Stellen umständlich. Es werden beispielsweise nur einzelne Bilder versendet. Daraus folgt, dass kein Stream zur Verfügung steht. Das einzelne Anfordern von Bildern ist aufwändig. An dieser Stelle könnte optimiert werden, indem auf andere Sockettypes wie Router in ØMQ gesetzt wird.

Ist eine zentrale Instanz des naogateway vorhanden, die ein Roboter ansteuert, können verschiedenen Anfragen von verschiedenen Stellen verarbeiten werden und die passenden Antworten an den korrekten Absender zurückgeleitet werden. Denn alle Anfragen werden über genau einen naogateway geleitet. Eine zentrale Instanz dagegen ist ein Single Point of Failure und mindert so die Verteilbarkeit. Wünschenswert wäre an dieser Stelle ein Aktor auf dem Nao, der direkt angesprochen wird. Hardwarenahe Aktoren wie libcppa (vgl. Charousset, Dominik, 2013) sind vorhanden. Der Einsatz von libcppa scheitert derzeit am C++ 11 Standard, für den der Compiler in qibuild nicht verfügbar sind. Die Bibliothek libcppa kann ohne einen C++ 11 kompatiblen Compiler nicht übersetzt werden.

Der NaoActor liefert die Referenzen seiner Kinder NoResponseActor, ResponseActor und VisionActor als ActorRef nach außen. Da an einer ActorRef zunächst einmal nicht erkennbar ist, welcher Aktor dahinter steckt, ist hierbei zu überlegen auf TypedActors auszuweichen. TypedActors haben einen besonderen Typ, anhand dessen ein Aktor zugeordnet werden kann.

Die Identifizierung einer Answer durch den darin gespeicherten Call erwies sich als funktionsfähig, jedoch als unpraktikabel, da der Code unübersichtlich und unnötig lang wird. Jede Referenz des Call muss gespeichert werden, um beim Verarbeiten der Answer darauf zugreifen zu können. Außerdem muss diese Referenz bei jedem Zustandswechsel mitgeführt werden. Die Identifizierung lässt sich vereinfachen, indem dem Call ein serialisierbares Token, wie einen String oder eine Enumeration, übergeben wird. Das Token wird in der Answer statt dem Call gespeichert. Dann ist es möglich im Patternmatching das Token anzugeben. Zusätzlich könnten mehrere Calls mit gleichem Inhalt unterschieden werden. Wenn man ein kleines Token wählt, ist das Objekt deutlich kleiner als ein vollständiger Call und lässt sich daher schneller serialisieren, verschicken und deserialisieren.

Wenn man in die Zukunft blickt, sind viele Anwendungsszenarien für humanoide Roboter denkbar. Smartphones können mit kleineren Applikationen - Apps - ausgestattet, werden. Eine ähnliche Zukunft ist für humanoide Roboter denkbar. Jedoch werden die Apps nicht auf dem Roboter, sondern auf einem Rechner im lokalen Netzwerk ausgeführt.

Eine fehlertolerante Softwarearchitektur ist nötig, um Roboter außerhalb des Labors produktiv einzusetzen. Denn dort ist nicht der Entwickler verfügbar, der den Roboter auffangen kann, wenn dieser stürzt. Dort ist kein Kabel sofort verfügbar, wenn der Akku schwächelt. Es ist auch keiner vorhanden, der nach einem Absturz einen Neustart durchührt.

Die Arbeit mit dem Nao gestaltete sich als schwierig, da eine Änderung der Software auf dem Nao an viele Restriktionen mit sich brachte. Die Buildumgebung lässt nur vom Hersteller freigegebene Software zu, eine Nutzung einer Paketverwaltung ist auf dem spezielle angepassten Linux nicht möglich. Daran zeigt sich, dass auch das sowohl das Betriebssystem, als auch das Metabetriebssystem des Naos noch etwas Zeit benötigen, um außerhalb des Labors verwendet werden zu können.

Die Entwicklung mit Akka gestaltete sich sehr einfach und inuitiv, was für einen produktiven Einsatz besonders wichtig ist. Gleichzeitig ist eine offene Schnittstelle auf Basis einer IDL ein

wichtiges Element um ein distributives System zu entwickeln, damit verschiedene Systeme miteinander kommunizieren können.

Was eine Architektur alleine nicht leisten kann, ist die Etablierung als Softwarestandard. Zum Softwarestandard entwickelte sich das rein auf Nachrichten spezifizierte ROS. Es ist sicher eine grundlegende Untersuchung Wert auf Basis der ROS Spezifikation eine Implementierung mit dem Aktorenmodell umzusetzen. Denn ROS verkörpert viele ähnliche Konzepte, wie Message Passing, Verwendung einer IDL und die Abstrahierung der Roboter von den Apps. Fehlertolleranz und Sprachenunabhängigkleit bleiben dort derzeit noch ein Nischenthema. Da kann die hier vorgestellte Architektur anknüpfen.

Inhaltsverzeichnis der angehangenen CD

- BA_Sigurd_Sippel.pdf - Bachelorarbeit in der Digitalversion
- naogateway - Projektordner mit Quellcode und Dokumentation
- Quellen - Ordner mit verwendeten, digitalen Quellen
  - Aldebaran (2013a).html
  - Aldebaran (2013b).html
  - Aldebaran (2013c).html
  - Aldebaran (2013d).html
  - Aldebaran (2013e).html
  - Aldebaran (2013f).html
  - Aldebaran (2013g).html
  - Aldebaran (2013h).html
  - Apache (2013).html
  - Charousset, Dominik (2013).html
  - Chibani, A., Amirat, Y., Mohammed, S., Hagita, N., and Matson, E. T. (2012a).pdf
  - Chibani, A., Schlenoff, C., Prestes, E., and Amirat, Y. (2012b).pdf
  - Einhorn, E., Langner, T., Stricker, R., Martin, C., and Gross, H.-M. (2012).pdf
  - Google (2013).html
  - Hintjens, P. (2013).pdf
  - Nestor, J., Wulf, W. A., and Lamb, D. A. (1981).pdf
  - Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009).pdf
  - ROS (2012)
  - Sarabia, M., Ros, R., and Demiris, Y. (2011).pdf
  - Sumaray, A. and Makki, S. K. (2012).pdf
  - Typesafe (2013).pdf
  - Typesafe Simple Build Tool (2013).html

Codebeispielverzeichnis

3.1 Aufbau des Pattern Matching 20
3.2 Beispiel eines Aktors 20
3.3 Aktor mit Versendung der Nachrichten an den Absender 21
3.4 Instanziierung eines Aktorensystems 22
3.5 Zustandsfunktion ping 23
3.6 Worker-Aktor für Ping-Pong-Protokoll 23
3.7 Heater-Aktor für Ping-Pong-Protokoll 24
3.8 Strategie eines Aktors für Exceptions 28
3.9 Beispiel für eine Protobuf-Datenstruktur 33
3.10 Beispiel für eine Thrift-Datenstruktur 33
3.11 Beispiel für einen „überladenen Datentyp“ 34
3.12 Beispiel für einen ØMQ Server 36
3.13 Beispiel für einen ØMQ Client 37

4.1 Erstellung eines Proxyobjekts mit dem Naoqi SDK in C++ 40
4.2 Request-Nachricht für den HAWAktor 44
4.3 „überladener Datentyp“ für Parameter und Rückgabewerte 44
4.4 Response-Nachricht für den HAWAktor 45
4.5 Zugriff auf ØMQ Sockets 45
4.6 Erstellung eines Requests für den HAWAktor 46
4.7 Implizite Konvertierung in MixedValue 46
4.8 Überführung eines Bytearrays in eine Response Nachricht 48
4.9 Request-Nachricht für den CamServer 48
4.10 Response-Nachricht für den CamServer 49
4.11 Zugangsdaten des Naos 50
4.12 Farbbereiche des HAWCamServers 55
4.13 Nachricht VisionCall für die Anfrage an den VisionActor 55

4.14 Verwendung des after pattern zum zerzögernden Ausführen 56
4.15 Konfigurationsbeispiel für Verzögerungen 57
4.16 Message Passing mit Timeout 58
4.17 Reaktion auf Timeout des Futures 59
4.18 build.sbt - SBT Konfigurationsdatei 63

Glossar

- Akka - Aktorenframework von Typesafe
- Aktor - leichtgewichtiger Prozess
- Aktoreneferenz - Referenz eines Aktors auf Basis einer URI
- Aktorensystem - Verwaltung von Aktoren
- Aldebaran - Hersteller des Nao Roboters
- Apache Camel - Kommunikationsframework für Protokoll wie HTTP
- Dispatcher - Weist Aktoren sowohl Threads als auch Nachrichten zu
- Frames - Serialisierte Nachricht in ØMQ
- Guardian - Wächter eines Aktors oder mehrerer Aktoren
- HAWAktor - Naoqi Modul für Naoqi Methodenaufrufe über ØMQ
- HAWCamServer - Naoqi Modul für Bereitstellung des Videobildes über ØMQ
- Heartbeat - Verbinungstest
- Mailbox - Nachrichtenschlange des Aktors
- Nao - humandoider Roboter
- Naoqi - Metabetriebssystem des Nao Roboters
- OpenNao - Betriebssystem des Nao Roboters
- Protocol Buffers - Framework für Serialisierung
- Root Gurdian - Der Wächter und Wurzel eines Aktorensystem
- System Guardian - Wächter der intern von Akka erstellten Akoren
- User Guardian - Wächter der vom Benuter erstellten Akoren
- ØMQ Context - Verwaltung von ØMQ Sockets
- ØMQ Socket - Schnittstelle zum Senden und Empfangen von Nachrichten über ØMQ
- ØMQ SocketType - Beschreibt das Verhalten eines ØMQ Socket (s. Request-reply Pattern)

Abkürzungsverzeichnis

- API - Application Programming Interface
- CORBA - Common Object Request Broker Architecture
- HTTP - Hypertext Transfer Protocol
- IDL - Interface Description Language
- JSON - JavaScript Object Notation
- MIRA - Middleware for Robotic Applications
- REP - Reply ØMQ SocketType
- REQ - Request ØMQ SocketType
- REST - Representational state transfer
- RMI - Remote Invocation Interface
- ROS - Robot Operating System
- RPC - Remote Procedure Call
- SBT - Simple Build Tool
- SDK - Software Development Kit
- SOAP - Simple Object Access Protocol
- TCP - Transmission Control Protocol
- SOA - Service-oriented architecture
- URI - Universal Resource Identifier
- URL - Universal Resource Locator
- WSDL - Webservice Description Language
- XML - eXtensible Markup Language
- ØMQ - ZeroMQ oder ZeroMessageQueue

Literaturverzeichnis

Aldebaran (2013a). *Naoqi.* Online unter: www.aldebaran-robotics.com/documentation/getting_started/software_in_and_out.html Abruf 03.05.2013.

Aldebaran (2013b). *Naoqi.* Online unter: www.aldebaran-robotics.com/documentation Abruf 02.04.2013.

Aldebaran (2013c). *Naoqi.* Online unter: www.aldebaran-robotics.com/en/Pressroom/Photography/nao.html#10 Abruf 03.05.2013.

Aldebaran (2013d). *Naoqi.* Online unter: www.aldebaran-robotics.com/documentation/dev/naoqi/index.html Abruf 12.05.2013.

Aldebaran (2013e). *Naoqi.* Online unter: www.aldebaran-robotics.com/documentation/dev/sdk.html Abruf 12.05.2013.

Aldebaran (2013f). *Naoqi.* Online unter: www.aldebaran-robotics.com/documentation/dev/cpp/install_guide.html Abruf 12.05.2013.

Aldebaran (2013g). *Naoqi.* Online unter: www.aldebaran-robotics.com/documentation/naoqi/vision/alvideodevice.html Abruf 03.05.2013.

Aldebaran (2013h). *Naoqi.* Online unter: www.aldebaran-robotics.com/documentation/qibuild Abruf 08.04.2013.

Apache (2013). *Thrift.* Online unter: thrift.apache.org/docs/idl Abruf 02.04.2013.

Breshears, C. (2009). *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications.* O'Reilly Media.

Charousset, Dominik (2013). *libcppa*. Online unter: `neverlord.github.io/libcppa/manual/index.html` Abruf 25.04.2013.

Chibani, A., Amirat, Y., Mohammed, S., Hagita, N., and Matson, E. T. (2012a). Future research challenges and applications of ubiquitous robotics. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 883–891, New York, NY, USA. ACM.

Chibani, A., Schlenoff, C., Prestes, E., and Amirat, Y. (2012b). Smart gadgets meet ubiquitous and social robots on the web. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 806–809, New York, NY, USA. ACM.

Colouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2012). *Distributed Systems: Concepts and Design (5th Edition)*. Addison-Wesley. ISBN 978-0-13-2143011.

Einhorn, E., Langner, T., Stricker, R., Martin, C., and Gross, H.-M. (2012). Mira-middleware for robotic applications. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 2591–2598. IEEE.

Esser, F. (2011). *Scala für Umsteiger*. Oldenbourg. ISBN 348-6-59-6934.

Google (2013). *Protocol Buffers*. Online unter: `developers.google.com/protocol-buffers/docs/overview` Abruf 25.02.2013.

Hintjens, P. (2013). *Code Connected Volume 1: Learning ZeroMQ*. CreateSpace Independent Publishing Platform. ISBN 148-1-26-2653.

Nestor, J., Wulf, W. A., and Lamb, D. A. (1981). *IDL, Interface Description Language*. PhD thesis, Carnegie Mellon University Pittsburgh.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3.

ROS (2012). *ROS*. Online Unter: `www.ros.org/wiki/ROS/Introduction`. Abruf 21.03.2013.

Sarabia, M., Ros, R., and Demiris, Y. (2011). Towards an open-source social middleware for humanoid robots. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International*

Conference on, pages 670–675. IEEE.

Sumaray, A. and Makki, S. K. (2012). A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, pages 48:1–48:6, New York, NY, USA. ACM. doi.acm.org/10.1145/2184751.2184810.

Tanenbaum, A. (2003). *Verteilte Systeme*. Pearson Studium. ISBN 382-7-37-0574.

Typesafe (2013). *Akka Documentation 2.1.1*. Online unter: `doc.akka.io/docs/akka/2.1.1/Akka.pdf` Abruf 04.03.2013.

Typesafe Simple Build Tool (2013). *Simple Build Tool*. Online unter: `www.scala-sbt.org` Abruf 26.04.2013.

Wyatt, D. (2013). *Akka Concurrency*. Artima Inc. ISBN 978-0-98-1531663.

Versicherung der Selbständigkeit

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

Hamburg, 07. Juni 2013 Sigurd Sippel