Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

David Olszowka

Humanoide Robots in der local Cloud –
Design und Programmierung eines APIs mithilfe des Aktorensystems Akka, ZeroMQ und Google Protocol Buffers

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

David Olszowka

Humanoide Robots in der local Cloud - Design und Programmierung eines APIs mithilfe des Aktorensystems Akka, ZeroMQ und Google Protocol Buffers

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang 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. Thomas Canzler

Abgegeben am 02.06.2014

David Olszowka

Thema der Bachelorarbeit

Humanoide Robots in der local Cloud - Design und Programmierung eines APIs mithilfe des Aktorensystems Akka, ZeroMQ und Google Protocol Buffers

Stichworte

NAO, Roboter, Aktor, Akka, ZeroMQ, Protobuf

Kurzzusammenfassung

In dieser Arbeit wird eine Schnittstelle für humanoide Roboter entworfen und am Beispiel des NAOs implementiert. Die Schnittstelle dient dazu, Anwendungen unabhängig vom eigentlichen Robotertyp zu machen und die Kommunikation mit dem Roboter so weit, wie möglich zu abstrahieren. Die Schnittstelle wird mithilfe des Aktorensystems Akka umgesetzt und die Anbindung des NAOs an die Schnittstelle erfolgt mit dem ZeroMQ und Protocol Buffers.

David Olszowka

Title of the paper

Humanoid robots in the local cloud – API design and implementation with the actor system Akka, ZeroMQ and Google Protocol Buffers

Keywords

NAO, Robots, Actor, Akka, ZeroMQ, Protobuf

Abstract

This paper is about the design and implementation of an abstract API used for humanoid robots. The API should make applications, which work with robots, independent of the actual robot type. The API is implemented using the actor framework Akka. To show, how the robot would implement this API, the NAO will be used to implement the API using ZeroMQ and Google's Protocol Buffers.

Inhaltsverzeichnis

1 Einleitung

Roboter nehmen dem Menschen immer mehr anstrengende körperliche Arbeit ab. Sie sind aus dem Produktionshallen inzwischen nicht mehr wegzudenken.
Die Produktionshallenroboter sind perfekt an ihre Arbeitsschritte angepasst und daraufhin programmiert. Im Gegensatz zu Menschen haben die meisten Roboter ein sehr kleines Feld an Aufgaben, die sie erledigen können und sind nicht vielseitig einsetzbar. Da sich in einem Fertigungsbetrieb die Arbeitsabläufe nicht häufig ändern, stellt dies kein Problem dar. Roboter, die Menschen in möglichst vielen Bereichen helfen sollen, sind dem Menschen als humanoide Roboter nachempfunden. Diese Roboter verfügen über die gleiche Anatomie, wie Menschen und können somit prinzipiell die gleichen Bewegungen ausführen.
Bei so vielen möglichen verschiedenen Aufgaben, wie dem Erkennen von Sprache, der Erledigung von Einkäufen, dem Aufräumen und noch vielem mehr, ist es nicht mehr vertretbar, den Roboter für jede Aufgabe neu zu programmieren, oder ihn mit Programmen für jede mögliche Aufgabe zu versehen. Manche Aufgaben, wie Spracherkennung und Bildinterpretation sind sehr Rechenintensiv. Die Rechenleistung in diesen Robotern ist jedoch meist gering, um die Akkulaufzeit nicht zu beeinträchtigen. Deshalb bietet es sich an, die Aufgaben nicht vom Roboter selbst, sondern von einer externen Berechnungseinheit durchzuführen.

Der Roboter erfüllt dann lediglich die Funktion einer Schnittstelle, über welche die Recheneinheit mit der Welt interagieren kann. Das hat den entscheidenden Vorteil, dass der Roboter in neue Aufgaben integriert werden kann, ohne dass dieser angepasst werden muss. Zudem können so mehrere Roboter von der Recheneinheit koordiniert an der gleichen Aufgabe arbeiten, ohne komplizierte distributive Algorithmen im Roboter zu implementieren.

Bisherige Schnittstellen zur Einbindung von Robotern in Computernetzwerke sind auf jeden Robotertyp einzeln zugeschnitten. Dies widerspricht der Idee, eine Anwendung zu entwickeln, die anschließend für jeden Roboter verwendet werden kann.

1.1 Zielsetzung und Abgrenzung

Die Zielsetzung dieser Arbeit besteht darin, ein abstraktes API für die Ansteuerung von Robotern zu entwerfen und am Beispiel des NAOs zu implementieren. Dieses API soll die Funktionalität der meisten humanoiden Roboter abbilden und es soll möglich sein, alle Roboter in dieses API einzubinden. Des Weiteren soll das API erweiterbar sein, sodass neue Funktionalität hinzugefügt werden kann, ohne dabei bereits entwickelte Anwendungen zu beeinträchtigen.

Bei dem Entwurf und der Umsetzung des APIs für den NAO steht die Performance im Vordergrund. Anwendungen, die das API verwenden sollen durch die stark begrenzten Ressourcen humanoider Roboter so wenig eingeschränkt werden, wie möglich. Der Entwurf und die gewählten Bibliotheken müssen darauf ausgelegt sein, Kommunikations- und Verarbeitungszeiten möglichst gering zu halten.

Der Aspekt der Sicherheit des Systems wird in dieser Arbeit nicht behandelt. Es wird davon ausgegangen, dass das System nur in einem geschützten Umfeld eingesetzt wird und somit auf entsprechende Authentifikations- und Autorisationsverfahren verzichtet werden kann.

2 Gewählte Mittel

Die gewählten Mittel müssen entsprechend den Anforderungen an das Gesamtsystem ausgewählt werden. Die zeitnahe Ausführung von Anfragen ist eine wichtige Anforderung, die an so ein System gestellt wird. Da die Steuerung von Robotern in Echtzeit geschieht, müssen alle Anfragen so zeitnah, wie möglich verarbeitet werden. Nur so kann eine Anwendung korrekt entscheiden, wie der Roboter auf eine bestimmte Situation regieren soll.
Dies führt dazu, dass sowohl die Kommunikation mit dem Roboter, als auch die interne Verarbeitung so wenig Overhead haben muss, wie möglich.

2.1 NAO

Der NAO, ein 58cm großer humanoider Roboter des französischen Roboterherstellers Aldebaran Robotics, ist das einzige Glied des Systems, welches für diese Arbeit vorgegeben war.

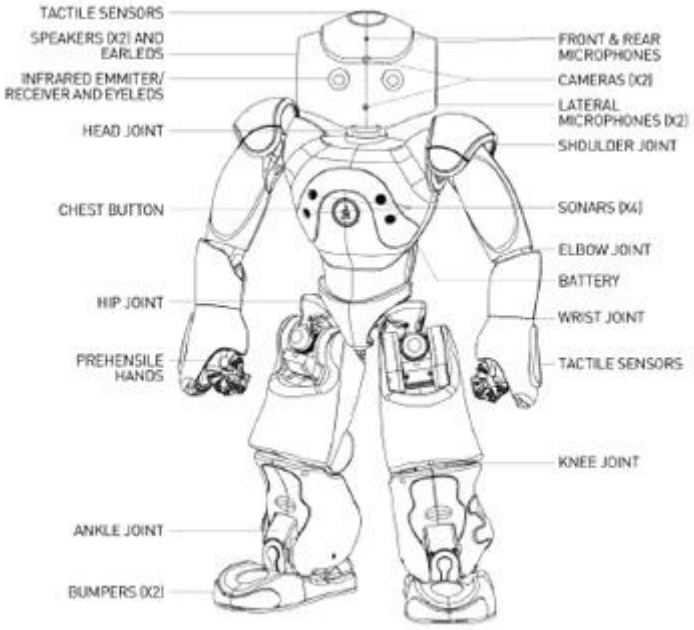

Abb. 1 NAO

Der NAO verfügt über folgende Ausstattung:

- Zwei Frontkameras á 1288x968 Pixel
- Vier Mikrofone
- Zwei Ultraschall-Entfernungssensoren
- Zwei Infrarotsender & -Empfänger
- Neun Berührungssensoren
- Acht Drucksensoren
- Einem Beschleunigungssensor
- Zwei Gyroskope
- Intel ATOM 1,6 GHz CPU
- W-Lan Modul
- 27,6 Wattstunden Batterie, die für ungefähr 1,5 Stunden Betrieb reicht.[1]

Er verfügt somit über eine ganze Reihe an Möglichkeiten, Informationen von der Umwelt aufzunehmen und mit der Umwelt zu interagieren.

[1] vgl. [Alde2013a]

Die Software des NAOs läuft auf einer eigenen GNU/Linux Distribution, die auf Gentoo basiert, genannt OpenNAO.
Auf dem Betriebssystem laufen lediglich ein Netzwerkmanager, welcher für die Verbindung mit dem W-Lan und eingehende SSH/FTP-Verbindungen zuständig ist, und das NAOqi.
NAOqi ist das Framework, welches den gesamten NAO steuert. Es ist in C++ geschrieben, in Module unterteilt und bietet Schnittstellen für das Ansprechen der einzelnen Module an.[1]
Die Robotermodule und Anwendungsbeispiele in dieser Arbeit wurden unter der Verwendung der NAOqi-Version 1.14.5 erstellt und auf diese angepasst.

Zum Ansprechen der Schnittstellen bietet das NAOqi folgende Möglichkeiten.
Aldebaran bietet für viele populäre Sprachen, darunter C++, Python, Java und die .Net-Sprachen, ein SDK zum Entwickeln von entfernten Modulen an. Diese Module kommunizieren über spezielle Proxy-Klassen mit dem NAOqi, welches auf einem wählbaren TCP-Port auf eingehende Anfragen wartet, sie an das Modul weiterleitet und die Antworten wieder zurück an den Proxy sendet. Die Proxy-Klassen bieten also ein RPC-Interface an.
Das Problem an dieser Methode ist jedoch das verwendete SOAP-Protokoll zur Kommunikation. Dieses generiert Informationen, die für den RPC-Aufruf nicht relevant sind und serialisiert die zu übertragenden Informationen im XML Format in einen SOAP-Envelope zusammen mit nicht relevanten Header-Daten. Bereits die Verwendung des REST-Protokolls mit JSON-Serialisierung weist einen deutlichen Performancegewinn gegenüber SOAP auf.[2]

Die zweite Möglichkeit, das NAOqi anzusprechen ist das Kompilieren einer C++-Bibliothek, die auf den NAO gespielt wird und beim Start von NAOqi in den NAOqi-Prozess geladen wird. In diesem Fall fällt die umständliche Netzwerkkommunikation weg, was die Reaktionszeit des NAOs deutlich verbessert. Damit die Bibliothek auf der NAO-Hardware ausgeführt werden kann, bietet das NAOqi-SDK ein eigenes Build-Tool namens qiBuild für Linux an. Dieses Tool enthält einen Cross-Compiler für die die NAO-Architektur, der es erlaubt die NAO-Module auf beliebigen Maschinen zu bauen.

Um eine Anwendung zu schreiben, die den NAO so performant, wie möglich nutzt, müsste diese direkt als NAO-Modul geschrieben werden. Doch dieser Ansatz ist, wie eingangs bereits erwähnt aus verschiedenen Gründen nicht sinnvoll. Deshalb wird in dieser Arbeit unter anderem ein Modul entworfen, welches zwar im NAO sitzt, aber nichts weiter tut, als eingehende Anfragen an das NAOqi weiter zu geben und die Antworten zurück zu leiten. Der Fokus liegt darauf, dies so performant, wie möglich zu gestalten. Durch die Wahl der Implementierung als lokales Modul, ist die Implementierungssprache für das Robotermodul auf C++ festgelegt. Dementsprechend müssen die Bibliotheken, die zur Kommunikation mit dem Aktorenframework genutzt werden C++ unterstützen.

[1] vgl. [Alde2013b]
[2] vgl. [MuGra2009] S. 1428

2.2 Aktorenframework Akka

Bei der Integrierung des NAOs stellt sich die Frage, wie er so integriert werden kann, dass sich die Schnittstelle nicht unnötig kompliziert gestaltet und von technischen Details abstrahiert. Dies ist auch für den Aspekt der Wiederverwendbarkeit wichtig, da im Optimalfall eine Anwendung für Roboter geschrieben wird und es anschließend keine Rolle spielt, ob dieser Roboter der NAO, eine neuere Version, oder ein anderer humanoider Roboter ist.
Der simpelste Ansatz wäre die Lösung, wie sie momentan von Aldebaran selbst umgesetzt ist. Der NAO lauscht auf einem konfigurierbaren TCP-Port im lokalen Netzwerk auf eingehende Anfragen, bearbeitet diese und antwortet wieder über TCP.
Dieser Ansatz ist nicht sonderlich wiederverwendbar, da bei jeder Änderung des Ports oder IP-Adresse die Anwendung, die den NAO nutzt angepasst werden muss. Das Problem lässt sich zwar durch Discovery-Protokolle oder Namensdienste lösen, jedoch gibt es noch eine Einschränkung bei diesem Ansatz.
Und zwar muss das Robotermodul für alle Roboterversion die gleichen Nachrichten akzeptieren und die gleichen Funktionen anbieten. Den Nachrichtenumfang nachträglich zu erweitern oder zu ändern könnte wieder Änderungen in den Anwendungen nach sich ziehen. Dadurch, dass die gesamte Abstraktion in dem Robotermodul stattfinden müsste, ist das Gesamtsystem starr und Neuerungen sind nicht einfach einführbar.

Deshalb wurde hier ein Aktorensystem als Grundlage gewählt. Aktorensysteme zeichnen sich vor allem durch Zuverlässigkeit und Skalierbarkeit aus. Aktorensysteme werden in der distributiven Programmierung eingesetzt und verzichten dabei auf jegliche Formen der Synchronisation im Code. Dadurch wird der Code für distributive Anwendungen verständlicher und ist nicht anfällig für typische Locking-Probleme.

Ein Aktorensystem besteht aus einer Menge von Unterprozessen, den Aktoren. Diese Aktoren können Nachrichten empfangen und senden. Der Nachrichtenaustausch ist dabei die einzige erlaubte Form der Kommunikation zwischen zwei Aktoren.
Zum Senden und Empfangen von Nachrichten erhält jeder Aktor eine logische Adresse, die den Aktor nicht an eine physikalische Adresse (einen Speicherbereich oder einen Computer) bindet. Durch die Verwendung der logischen Adressen können Aktorensysteme so gut skalieren. Es spielt für die anderen Aktoren im System nämlich keine Rolle, ob sich der andere Aktor auf einer anderen Maschine befindet. Dadurch können aufwendige Berechnungen durch Auslagern einzelner Aktoren auf verschiedene Maschinen innerhalb des Aktorensystems massiv parallelisiert werden, ohne die Aktoren selbst anpassen zu müssen.

Mithilfe der Aktoren kann man die gesamte Kommunikation zwischen dem NAO und dem Aktorensystem dadurch abstrahieren, dass man einen Aktor im Aktorensystem hat, welcher mit dem NAO direkt kommuniziert und diesen zum restlichen Aktorensystem hin repräsentiert. Dadurch, dass der gesamte technische Code für die Kommunikation in diesem einen Aktor gekapselt ist, muss auch nur dieser Aktor angepasst werden, sollte sich das Interface zu dem NAO ändern. Die Anwendungen, die den NAO nutzen bleiben dadurch unverändert funktionsfähig.

Das Aktorensystem Akka ist für die Programmiersprachen Java und Scala verfügbar. Die Kernkonzepte sind aus dem Aktorensystem der Programmiersprache Erlang übernommen. Akka führt mit den Sprachmitteln von Scala eine eigene Aktoren-DSL ein, die an die Syntax von Erlang erinnert.

```
1. val system = ActorSystem("system")
2. actor(system, "pong")(new Act {
3.   become {
4.     case "ping" => sender ! "pong"
5.   }
6. })
```

Codebeispiel: Erstellen eines einfachen Aktors in Scala

Aktoren können in Akka auf zwei Weisen erstellt werden. Es kann die *actor*-Methode zusammen mit dem *Act*-Trait verwendet werden, um einfache Aktoren anzulegen, die nur eine *receive*-Methode benötigen, um ihre Aufgaben zu erfüllen. Die *Receive*-Methode wird mit *become* im Aktor installiert. Sie hat den Typ *PartialFunction[Any,Unit]* und gibt an, wie der Aktor eingehende Nachrichten behandeln soll. Wird eine Nachricht nicht behandelt, dann wird sie verworfen. Darin unterscheidet sich das Aktorensystem von Akka und Erlang. In Erlang werden nicht behandelte Nachrichten so lange wieder an den Aktor gegeben, bis sie irgendwann verarbeitet werden.
Muss ein Aktor zur *receive*-Methode zusätzlich Unteraktoren starten, einen Zustand halten oder auf Zustandsänderungen reagieren können, dann sollte aus Gründen der Übersicht eine Klasse angelegt werden, welche von *akka.actor.Actor* erbt und die *receive*-Methode definiert.

```
class PongActor extends Actor {
  def receive = {
    case "ping" => sender ! "pong"
  }
}
```

```
val system = ActorSystem("system")
system.actorOf(Props[PongActor], "pong")
```

Codebeispiel: Definieren und starten eines einfachen Aktors in Scala

Wenn die Klasse des Aktors definiert wurde, kann dieser gestartet werden, wie es das rechte Codebeispiel zeigt. Sobald der Aktor gestartet wird, ist er über eine URI im Aktorensystem eindeutig identifizierbar. In dem Codebeispiel würde die URI „akka://system/user/pong“ lauten. Diese URIs können genutzt werden, um dem Aktor

Nachrichten zu senden. Dabei spielt es keine Rolle, welche Klasse den Aktor mit der angegebenen URI implementiert. Die URI-Schnittstelle ist somit eine sehr lose gekoppelte Schnittstelle, da die Implementierung sogar zur Laufzeit jederzeit durch eine ganz andere Implementierung ausgewechselt werden kann. Die sehr lose Kopplung hat jedoch auch einen Nachteil. Die URI allein reicht nicht, damit ein Anwendungsentwickler die Schnittstelle nutzen kann. Dafür wird zusätzlich eine genaue Beschreibung der Schnittstelle benötigt, die angibt, welche Nachrichten akzeptiert werden und wie die Antworten aussehen. Bei enger gekoppelten Schnittstellen, wie den Java-Interfaces erhält ein Anwendungsentwickler in der Schnittstellendefinition die Beschreibung der akzeptierten Parameter und der Rückgabewerte. Diese Informationen kann wiederum von einer IDE dazu verwendet werden, ein falsches Benutzen der Schnittstelle bereits zur Entwicklungszeit zu erkennen und zu unterbinden.
Diese Informationen sind bei der Aktorenschnittstelle nicht vorhanden, da anhand der URI keine Aussage über die Art des Aktors getroffen werden kann. Die Semantik des Aktors, der durch die URI identifiziert wird, ist erst zur Laufzeit festgelegt. Deswegen ist es wichtig, das Verhalten eines Aktors nach außen hin genau zu definieren und zu beschreiben. Andernfalls kann der Anwendungsentwickler den Code des Aktors kennen, um die Schnittstelle zu verwenden.

2.3 ZeroMQ

ZeroMQ ist ein Kommunikationsframework, welches mit dem Ziel entwickelt wurde, die Interprozesskommunikation von den tatsächlich genutzten Mitteln zum Transport der Informationen zu entkoppeln und Socketarten auf einer Abstraktionsschicht anzubieten, die an der zu lösenden Aufgabe orientiert ist.
Des Weiteren wurde ZeroMQ mit dem Ziel entwickelt, die Kommunikation so performant, wie möglich durchzuführen. Um die Performanz zu gewährleisten haben sich die Entwickler unter anderem für C als Implementierungssprache für ZeroMQ entschieden.
Die Implementierung selbst wurde auf alle bekannten Sprachen portiert. Entweder durch den Aufruf der kompilierten C-Bibliothek über ein natives Interface der Sprache oder im Fall von C++ durch das Aufsetzen einer objektorientierten Hülle auf den C-Kern.

Entwickelt man selbst eine Anwendung, die Nachrichten vermittelt, so stößt man bei klassischen TCP & UDP Sockets schnell auf folgende Probleme. UDP ist unzuverlässig und bietet keine Fehlerbehandlung für verloren gegangene Pakete. TCP bietet zwar diese Fehlerbehandlung und stellt eine at-most-once Semantik sicher, liefert jedoch nur einen Bytestrom und keine klar abgegrenzten Pakete. Diese Paketgrenzen und Fehlertoleranz müssen dann von Hand implementiert werden.
ZeroMQ bietet beides. Es verwendet TCP, um einen zuverlässigen Nachrichtentransfer zu gewährleisten und behandelt die zu versendenden Daten als Nachrichten fester Größe. Ein TCP-Packet ist in ZeroMQ jedoch nicht äquivalent zu einer Nachricht. ZeroMQ kann mehrere kleine Nachrichten in einem TCP-Packet zusammenfassen, um eine möglichst hohe Übertragungsrate sicher zu stellen. Um die eben genannten Anforderungen zu gewährleisten, verwendet ZeroMQ ein eigenes Nachrichtenformat, welches die übertragenen TCP-Daten inkompatibel zu gängigen Anwendungs-protokollen, wie HTTP, FTP und SMTP macht. Diese Einschränkung ist für diese Arbeit jedoch nicht relevant, da die gesamte Netzwerkkommunikation ausschließlich über ZeroMQ-Sockets durchgeführt wird.
Um in allen Anwendungsbereichen hohe Peformance sicherzustellen, bietet ZeroMQ das Protokoll „pgm://“ für Multicast-Nachrichten, „ipc://“ für Interprozesskommunikation auf der gleichen Maschine und „inproc://“ für Intraprozesskommunikation an.[1]

Welche Alternativen gibt es zu ZeroMQ?
Da das Implementieren aufwendiger Kommunikationsprotokolle mit TCP-Sockets zu aufwendig und fehleranfällig ist, stellt dies keine echte Alternative dar. Da ZeroMQ bereits auf Übertragungsgeschwindigkeit optimiert ist, ist ein Mehrwert durch eine Eigenentwicklung fraglich.
Aus der Java-Welt gibt es eine sehr weit verbreitete und beliebte Alternative: Netty.
Netty ist eine nicht-blockierende Kommunikationsbibliothek die viele Protokolle direkt unterstützt. Die nicht-blockierende Eigenschaft ist insbesondere in Hinblick auf die Skalierbarkeit wichtig. Sowohl Netty, also auch ZeroMQ sind nicht blockierende

[1] vgl. [Zero2013]

Kommunikationsframeworks. Gemeint ist damit, dass der Thread, der die Nachrichten vom Socket liest und sendet nicht blockiert bis neue Nachrichten angekommen sind, da dies die Skalierbarkeit stark beeinträchtigt.
In der klassischen Herangehensweise wird ein Thread für das Annehmen von neuen Verbindungen erstellt. Dieser wird von der *Accept*-Methode solange blockiert, bis eine neue Verbindung verfügbar ist. Ist eine neue Verbindung geöffnet worden, dann wird ein Thread für diese Verbindung gestartet. Der Verbindungsthread wartet dann auf eingehende Daten, indem er von der *Receive*-Methode blockiert wird, bis neue Daten am Socket eingegangen sind.
Dies funktioniert bei Anwendungen mit geringer Nutzerlast zwar sehr zuverlässig und ist einfach zu implementieren. Bei einer großen Anzahl an eingehenden Verbindungen schlägt sich die enorme Anzahl an Threads und die ständigen Kontextwechsel, welche durch das Blockieren und Wecken verursacht werden, negativ auf die Performance der Anwendung nieder. Zudem muss beachtet werden, dass jeder Thread Speicher für seinen Stack benötigt und somit die Anzahl der möglichen Threads durch die Systemressourcen beschränkt ist.
Nicht blockierende Kommunikation basiert darauf, dass genau ein Thread prüft, ob eingehende Verbindungen anliegen. Wenn keine Verbindungen anliegen, dann werden alle offenen Verbindungen von dem gleichen Thread der Reihe nach auf neue Nachrichten geprüft und diese je nach Implementierung direkt verarbeitet oder an Threads weitergegeben, welche die Nachrichten verarbeiten.
In Netty gibt es einen Handler-Thread, an welchem man Nachrichtenhandler für eingehende Nachrichten anmelden kann. Dieser Thread fragt alle Verbindungen nach neuen Nachrichten ab und ruft für die Nachrichten die entsprechenden Nachrichtenhandler auf.
Das besondere an Netty ist die Tatsache, dass die Handler komponierbar sind und man für eine verschlüsselte Kommunikation, deren Nachrichten über HTTP getunnelt sind einfach einen Handler für das auspacken der Daten aus dem HTTP anmelden kann. Dieser reicht die ausgepackten Daten an einen Handler weiter, der die Daten entschlüsselt und diese wiederum an den eigentlichen Nachrichtenhandler weiterreicht.[1]
Dies gilt nicht nur für die Empfangsrichtung. Für das Senden bietet Netty den gleichen Mechanismus. Damit bietet Netty eine gut wartbare Möglichkeit, bestehende Nachrichtenprotokolle beliebig zu erweitern.
Im Gegensatz zu ZeroMQ ist in Netty in Google Protocol Buffers zur Serialisierung bereits eingebunden. Netty bietet also eine größere Funktionsvielfalt und eine bessere Abstraktion von der Kommunikation als ZeroMQ. Trotzdem wurde für diese Arbeit ZeroMQ gewählt. Der Hauptgrund dafür liegt darin, dass das Modul, welches auf dem NAO installiert wird, in C++ geschrieben sein muss und Netty nur für Java verfügbar ist und auf dem NAO selbst keine JVM verfügbar ist, auf welcher die Bibliothek laufen könnte. Würde der NAO eine JVM anbieten, dann wäre jedoch der Performanceverlust durch die Nutzung des JNIs das ausschlaggebende Argument gegen Netty.

[1] vgl. [Ayedo2013]

ZeroMQ bietet verschiedene Kommunikationsmodelle[1] an. Jede Aufgabenstellung lässt sich auf die Anwendung eines oder mehrerer dieser Modelle zurückführen.

Grundsätzlich gibt es nur folgende drei Socketpaare in ZeroMQ:

PUB(lisher) & SUB(scriber). Bei diesen Sockets kann der Publisher Nachrichten an beliebig viele angemeldete Subscriber schicken. Damit ein Subscriber Nachrichten erhalten kann, muss sich dieser an dem Publisher anmelden. Ein Subscriber kann sich auch an beliebig vielen Publishern anmelden und erhält alle Nachrichten in einer fairen Warteschlange.
Dieses Verhalten entspricht dem aus der objektorientierten Programmierung bekannten Observer-Pattern. Die Subscriber entsprechen den Observern und werden von den Publishern über Statusänderungen benachrichtigt.

PUSH & PULL. Dieses Socket-Paar verhält sich ähnlich zu dem vorherigen. Jedoch werden alle von einem PUSH-Socket gesendeten Nachrichten so lange in einer Warteschlange gehalten, bis ein PULL-Socket explizit eine Nachricht abfragt. Diese Nachricht wird dann nur an diesen Socket zugestellt und nicht an alle verbundenen Sockets verteilt.
Erreicht die Anzahl der Nachrichten in der Warteschlange eine einstellbare Höchstgrenze, dann lässt sich auf dem PUSH-Socket so lange nichts mehr senden, bis wieder eine Nachricht abgerufen wurde.
Dieses Paar von Sockets eignet sich zur Implementierung von Lastverteilung in Netzwerkanwendungen. Je nach Last können beliebig viele Worker hinzugeschaltet werden. Die Kommunikation ist wie auch beim PUB&SUB Paar unidirektional und somit nicht ohne Anpassung für alle Anwendungen als Lastverteiler geeignet, da Antworten auf anderem Weg zu dem Anfragenden gelangen müssen.
Jeder PUSH-Socket kann von beliebig vielen PULL-Sockets abgefragt werden und jeder PULL-Socket kann sich mit beliebig vielen PUSH-Sockets verbinden. Dieses Socketpaar bietet also eine Lösung für das Erzeuger-Verbraucher-Problem in verteilten Anwendungen.

REQ(uest) & REP(ly). Das letzte Socket-Paar stellt eine zustandsbehaftete, bidirektionale Verbindung zwischen zwei Endpunkten dar. Auf jede Nachricht, die von dem REQ-Socket gesendet wird, muss eine Antwort von dem REP-Socket folgen, ehe eine weitere Nachricht gesendet werden kann. Da in Netzwerkanwendungen Request-Reply Protokolle häufige Anwendung finden (HTTP, FTP, SMTP…), die paarweise Socketbindung jedoch zu einschränkend ist, da so nur 1:1 Verbindungen aufgebaut werden können, stellt ZeroMQ zwei weitere Hilfssockets zur Verfügung. Nämlich ROUTER und DEALER. Der ROUTER-Socket ist das REP-Äquivalent mit der Eigenschaft, dass sich mehrere REQ-Sockets mit einem ROUTER-Socket verbinden können. Die am ROUTER-Socket eingehenden Nachrichten werden durch einen Verbindungscode, welcher jeder Nachricht vorangestellt wird, dem Absendenden Socket zugewiesen. Der DEALER-Socket ist das REQ-Äquivalent, welches sich mit mehreren REP-Sockets verbinden kann. Der DEALER-Socket verteilt alle Nachrichten

[1] Im Original: „messaging patterns“

gleichmäßig an alle verbundenen REP-Sockets. Während der ROUTER-Socket lediglich dazu dient, eine Antwort an den korrekten REQ-Socket zu senden.
Durch die Erweiterung um die ROUTER- und DEALER-Sockets lassen sich auch 1:N-Verbindungen implementieren. Jedoch muss hier beachtet werden, dass es keine richtigen Verbindungen sind, da die Anfragen vom DEALER gleichmäßig verteilt werden. Es kann also nicht sichergestellt werden, dass zwei aufeinander folgende Nachrichten von dem gleichen REP-Socket beantwortet werden. Deshalb muss bei der Verwendung darauf geachtet werden, dass die Kommunikation, vom Request-Reply abgesehen, zustandslos ist.
Da man auch ROUTER-Sockets mit DEALER-Sockets verbinden kann, lassen sich damit auch N:M-Verbindungen implementieren.

Für die Intraprozesskommunikation bietet ZeroMQ noch einen PAIR-Socket an, mit dem sich zwei Endpunkte bidirektional Nachrichten schicken können. Dieser Socket-Typ ist jedoch aufgrund von technischen Einschränkungen nur für die Intraprozesskommunikation geeignet.[1]

Folgende Kombinationen von Sockets sind erlaubt:

PUB	SUB
PUSH	PULL
REQ	REP
REQ	ROUTER
DEALER	REP
DEALER	ROUTER
PAIR	PAIR
DEALER	DEALER
ROUTER	ROUTER

Tbl. 1 - Erlaubte Socketkombinationen

[1] vgl. [Zero2013]

2.4 Google Protocol Buffers

Immer, wenn Prozesse Daten austauschen müssen, stellt sich die Frage nach der Serialisierung. Man muss die Daten aus einem Prozess 1:1 in den anderen Prozess übertragen und dies muss zudem noch performant sein. Auf unterschiedlichen Maschinen wird der gleiche Speicherblock unterschiedlich interpretiert und auch unterschiedliche Programmiersprachen interpretieren Daten unterschiedlich und nutzen unterschiedliche Repräsentationen im Speicher.
Ein Beispiel dafür sind die zweidimensionalen Arrays in Java und C++.

```
1.  char ary[3][5];
```

Dieses Beispiel reserviert in C++ einen durchgängigen Speicherbereich von 3x5 *char*-Feldern, was im Speicher wie folgt aussehen würde:

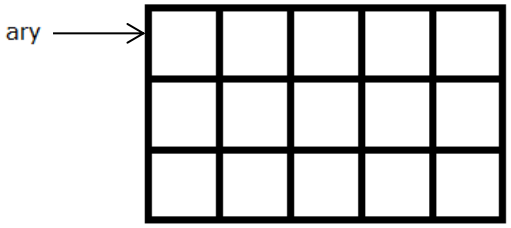

Abb. 2 Speicherbelegung von Arrays in C++

```
1.  char[][] ary = new char[3][5]();
```

Dieses Beispiel erzeugt in Java hingegen verkettete Array-Struktur, die wie folgt aussieht:

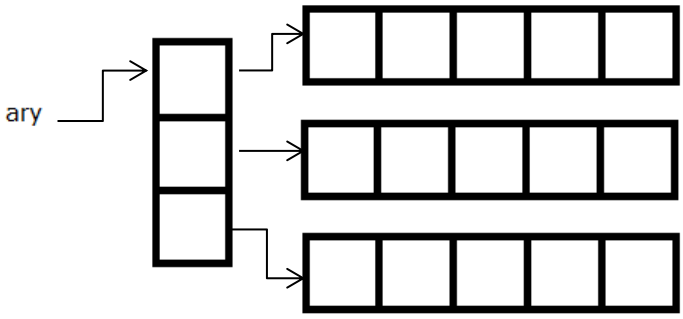

Abb. 3 Speicherbelegung von Arrays in Java

In diesem Beispiel müsste noch berücksichtigt werden, dass *char*-Felder in C++ meist 1 Byte groß sind und in der Systemcodepage codiert sind, während sie in Java immer 2 Byte belegen und in Unicode codiert sind. Bereits bei einer so einfachen Struktur, wie dem eben beschriebenen Array ist die Serialisierung und Deserialisierung zwischen mehreren Sprachen keine triviale Aufgabe.

Für diese Art von Problemen hat Google die Serialisierungsbibliothek Protocol Buffers (kurz: Protobuf) entwickelt. Protocol Buffers wurde hauptsächlich für den schnellen Datenaustausch zwischen Java, C++ und Python entwickelt. Inzwischen gibt es Portierungen für die meisten bekannten Sprachen.
Protocol Buffers nutzt eine eigene Nachrichtenbeschreibungssprache. Die Nachrichtendefinitionen sind sprachunabhängig. Sie enthalten lediglich die Namen der Felder, deren Typen und eine Angabe darüber, ob das Feld notwendig, optional, oder eine Sammlung von Objekten ist. Jedem der Felder wird eine für diese Nachricht eindeutige Zahl zugewiesen. Diese dient dazu, die Nachricht beliebig erweitern zu können, ohne Anwendungen zu beeinträchtigen, die noch das alte Format nutzen.
Hier eine Beispielhafte Definition einer Nachricht:

```
1. message SearchRequest {
2.   required string query = 1;
3.   optional int32 page_number = 2;
4.   optional int32 result_per_page = 3;
5. }[1]
```

Diese Nachrichtendefinition kann dann von dem mitgelieferten Compiler in Java-Code, C++-Code oder Python-Code übersetzt werden. Dieser Code besteht aus einer Klasse, welche die Nachricht repräsentiert und entsprechende Methoden zum Serialisieren und Deserialisieren anbietet. Das hat gegenüber der standartmäßigen Java-Serialisierung den großen Vorteil, dass die entsprechenden Routinen bereits zur Compile-Zeit vorliegen und das Objekt nicht zur Laufzeit mithilfe von Reflections zerlegt und zusammengesetzt wird.
Für Scala, welches auf Java aufsetzt und somit ebenfalls mit dessen Serialisierung Performanceeinbußen hinnehmen musste, wurde mithilfe von Typklassen, impliziten Objekten und Makros eine typsichere, performante und einfach zu benutzende Lösung für das Problem namens „Scala Pickling“ entworfen.[2]
Scala Pickling generiert den Serialisierungscode zur Compilezeit, wenn der Pickler im Code genutzt wird. Die Pickler arbeiten komplett im Hintergrund und an den Klassen selbst muss nichts geändert werden. Zudem kann Scala Pickling beim Deserialisieren prüfen, ob der erhaltene Bytestream tatsächlich zu dem erwarteten Typ passt.
Im Folgenden ein kurzes Beispiel, welches die Simple Nutzung eindrucksvoll demonstriert:

[1] [Proto2013]
[2] vgl. [OOPSLA2013]

```
1. import scala.pickling._
2. import json._
3. val pckl = List(1,2,3,4).pickle
4. val lst = pckl.unpickle[List[Int]][1]
```

Codebeispiel: Verwendung von Scala-Pickling

Bei dem Beispielcode ist zu beachten werden, dass die Klasse *List* von Scala die Methode *pickle()* nicht besitzt. Erst durch den Import von *scala.pickling._* wird eine Methode importiert, welche die Liste implizit in ein Objekt konvertiert, welches diese Methode kennt. Dieses Objekt weiß auch, wie eine Liste serialisiert werden muss.

Da Scala Pickling auf Scala allein beschränkt ist, ist es für die Aufgabenstellung dieser Arbeit jedoch nicht geeignet. Den großen Vorteil, dass man von der Bibliothek an sich nichts merkt, kann Scala Pickling nur innerhalb von Scala-Anwendungen ausspielen, da die Klassendefinition als Definition des Nachrichtenformats dient. Sollte das Pickling für die Kommunikation zwischen Scala und weiteren Sprachen weiterentwickelt werden, so müsste den anderen Sprachen ebenfalls eine Definition der Klasse in der entsprechenden Sprache zur Verfügung gestellt werden. Sobald also mehrere Prozesse aus verschiedenen Sprachen miteinander kommunizieren, kann die Serialisierung nicht mehr „unter der Hand“ geschehen und man erkennt, dass Protocol Buffers nicht schlecht entworfen wurde, sondern dass es nicht möglich ist, die Serialisierung einfacher in die Sprache zu integrieren.

Die Serialisierungsbibliothek MessagePack wäre hier als Alternative zu Protocol Buffers zu erwähnen. MessagePack bietet, wie auch Protocol Buffers Implementierungen in allen wichtigen Sprachen und ist laut eines Benchmarks von MessagePack[2] in der C++-Implementierung viermal so schnell, wie Protocol Buffers.
Bei diesem Benchmark muss jedoch beachtet werden, dass lediglich das reine Serialisieren und Deserialisieren gemessen wurde. MessagePack wird durch Compilermakros direkt in die zu serialisierende Klasse eingebunden und fügt dieser die zur Serialisierung notwendigen Methoden hinzu. Dadurch ist MessagePack sehr gut geeignet, um bereits bestehende Klassen serialisierbar zu machen.
Doch MessagePack hat dadurch auch einige Trade-Offs, die zur Wahl von Protocol Buffers geführt haben. Im Gegensatz zu Protocol Buffers definiert MessagePack keine eigene Nachrichtenbeschreibungssprache und generiert keine eigenen Klassen, sondern integriert seine Funktionalität in bereits bestehende Klassen. Dadurch weiß MessagePack nichts von der Semantik der Klasse. In dieser Arbeit müssen vor allem Parameter und Rückgabewerte mit dynamischen Typen serialisiert und deserialisiert werden. Ein solcher Parameter hat genau einen von vielen möglichen Typen. Protocol Buffers weiß genau, welche Felder einer Nachricht gesetzt wurden und serialisiert nur diese Felder. MessagePack hat diese Information jedoch nicht und serialisiert immer alle Felder des Objekts.

[1] [Pickl2013]
[2] vgl. [MsgB2013] und [Cooper2010]

Dadurch werden durch Protocol Buffers weniger Bytes erzeugt, die über das Netzwerk transportiert werden müssen. In umgekehrter Richtung bietet Protocol Buffers auch die Möglichkeit eine deserialisierte Nachricht abzufragen, welche Felder gesetzt waren, um darüber den tatsächlichen Typ der Parameter zu ermitteln. Diese Abfrage bietet MessagePack nicht und müsste durch zusätzliche Informationsfelder implementiert werden, welche wiederum das serialisierte Objekt größer machen würden.

Die Serialisierung von MessagePack mag zwar für große Objekte schneller sein, als Protocol Buffers, in diesem Fall ist jedoch Protocol Buffers aufgrund der speziellen Anforderungen die bessere Wahl. Ein letzter Punkt, der noch für die Wahl von Protocol Buffers spricht ist die bereits vorhandene Protocoll Buffers-Bibliothek auf dem Nao.

3 Realisierung

3.1 Design

Jeder Roboter wird im Aktorensystem durch genau einen Front-End-Aktor repräsentiert. Dieser Aktor hat die Aufgabe, alle Anfragen des Aktorensystems für den Roboter zu übersetzen und an diesen weiter zu leiten.

Damit die Roboter untereinander austauschbar sind, muss es eine einheitliche Nachrichtenschnittstelle geben, die von allen Front-End-Aktoren implementiert wird. Zusätzlich muss die Schnittstelle erweiterbar sein, falls neue Robotertypen eingebunden werden sollen, die neue Funktionalität mitbringen. Außerdem muss die Schnittstelle berücksichtigen, dass es auch Roboter gibt, die weniger Interaktionsmöglichkeiten mit der Umwelt besitzen und manche Aktionen gar nicht durchführen können.

Die Erweiterbarkeit ist durch die lose Kopplung der Nachrichtenschnittstelle von Akka direkt gegeben. Man kann jederzeit neue Nachrichten in die Schnittstelle aufnehmen, ohne die benutzende Anwendung anpassen zu müssen. Das NAOqi API[1] von Aldebaran passt auf die restlichen Anforderungen. Es trennt die Funktionalität durch Module in Funktionsgruppen auf und deckt die volle Funktionalität des NAOs ab, welchen man als typischen humanoiden Roboter ansehen kann. Durch die Aufteilung der Funktionalität in Module lassen sich Roboter mit weniger Funktionalität in das System integrieren, indem sie das entsprechende Modul nicht implementieren. Damit die Anwendungen auf die verschiedenen Roboter reagieren können, wird zusätzlich ein Info-Modul eingeführt, welches Informationen über den Roboter und die von ihm unterstützte Funktionalität liefert.

Im Folgenden wird die Umsetzung des Designs vom NAO bis hin zum Aktor vorgestellt.

3.2 NAO Module

Für das Schreiben von NAO Modulen stellt Aldebaran Entwicklern das NAOqi-SDK und das Tool qiBuild zur Verfügung. Das NAOqi-SDK beinhaltet alle Bibliotheken, die auch im NAO enthalten sind und einen NAO-Simulator. Mittels qiBuild werden eigene Module gegen diese Bibliotheken gelinkt und können dann in das NAOqi-SDK eingebunden und lokal im Simulator getestet werden.

[1] vgl. [Alde2013c]

Das qiBuild-Tool ist darauf ausgerichtet, Module für verschiedene Plattformen zu entwickeln. Schließlich kann der NAO eine andere Prozessorarchitektur benutzen, als die Entwicklungsmaschine. Die Konfiguration des Zielsystems geschieht über sogenannte Toolchains. Die Toolchains enthalten die Informationen, für welche Architektur das Modul gebaut werden soll, und wo die verwendeten Bibliotheken und das NAOqi-SDK zu finden sind.

Ein eigenes NAOqi-Modul muss nur folgendes C++-Interface implementieren, um vom NAO ausgeführt zu werden:

```
1. class CoreActorModule : public AL::ALModule
2. {
3. public:
4.   CoreActorModule(boost::shared_ptr<AL::ALBroker> broker, const std::string &name);
5.   virtual ~CoreActorModule();
6.   virtual void init();
7. };
```

Codebeispiel: Ein minimales NAOqi-Modul

Wenn das Modul gebaut wird, muss es nur genau so benannt werden, wie die Klasse. Dann kann der NAO das Modul ausführen.

Aufgrund der fehlenden Reflections und Laufzeitumgebung kann ein Modul in C++ nicht abfragen, welche anderen Module existieren oder deren Methoden aufrufen. Das NAOqi hat diese Einschränkung durch die *ALModule* Superklasse und den *ALBroker* aufgehoben. Die Klasse *ALModule* bietet allen Modulen die Möglichkeit über die Methode *bindMethod()* ihre Methoden anderen Modulen anzubieten. Andere Module können sich dann vom *ALBroker* eine Referenz auf das Modul holen und über *execute()* die angebotenen Methoden aufrufen. Alle vorinstallierten NAO-Module, die Teil des NAOqi APIs sind, bieten ihre Methoden auf diese Weise anderen Modulen an.
Für das gesamte NAOqi gibt es genau einen Broker, der jedes Modul kennt, welches vom NAO geladen wurde. Jedes Modul erhält diesen Broker im Konstruktor beim Initialisieren des Moduls übergeben und kann ihn dann im weiteren Verlauf dazu nutzen, mit anderen Modulen zu interagieren.

```
1. boost::shared_ptr<ALModuleCore> module(broker->getModuleByName("ALTextToSpeech"));
2. if (module.get() != NULL) {
3.   try {
4.     ALValue result;
5.     ALValue parameter;
6.     module->execute("getCurrentLanguage", parameter, returnVal);
7.     ...
8.   } catch(AL::ALError& e) {
9.     ...
10.  }
11. }
```

Codebeispiel: Aufruf einer Methode über die reflektive Schnittstelle von NAOqi

Nachdem das Modul instanziiert wurde, wird die Methode *init()* aufgerufen, damit sich das Modul initialisieren kann. NAOqi nutzt für die Initialisierung aller Module nur einen Thread, deshalb kann diese Methode nicht dazu verwendet werden, das Modul aktiv arbeiten zu lassen. Dies würde die Initialisierung aller nachfolgenden Module behindern.
Somit muss ein Modul für lang laufende Aufgaben in der *init()*-Methode eigene Threads starten, um diese Aufgaben abzuarbeiten.
Im Rahmen dieser Arbeit wurden zwei NAO-Module entwickelt. Ein generisches RPC-Modul und ein spezialisiertes Videomodul. Der Hauptfokus liegt auf dem RPC-Modul, welches das Bindeglied zwischen dem NAO und dem Aktor darstellt. Das Videomodul dient dazu, die Kamerabilder abzurufen und wenn möglich zu komprimieren. Da durch die Übertragung der Bilder über ein 802.11g W-LAN die Bildrate bei hoher Qualität nur 0,5 Bilder pro Sekunde erreicht[1] und bei einem 802.11n W-LAN entsprechend 6 Bilder pro Sekunde, soll mit der Komprimierung ein flüssigerer Bildstrom sichergestellt werden.

3.2.1 RPC-Modul

Das RPC-Modul hat die Aufgabe, auf eigehende Nachrichten auf einem ZeroMQ Socket zu warten. Die in der Nachricht angegebene Methode wird über die reflektive Schnittstelle des *ALBroker*s aufgerufen und der Rückgabewert wieder an den Absender der Nachricht zurückgesendet.
Die NAOqi-Module berücksichtigen parallele Zugriffe und sind deshalb untereinander synchronisiert, um unmögliche Aktionen zu vermeiden, wie zum Beispiel dem gleichzeitigen Öffnen und Schließen einer Hand. Aktionen, die sich gegenseitig behindern, werden durch die Synchronisation sequentiell ausgeführt. Zudem sind die meisten Aufrufe blockierend und blockieren den aufrufenden Thread so lange, wie die vom Roboter ausgeführte Aktion andauert.

Die blockierenden Aufrufe legen für die Kommunikation zwischen dem Aktorensystem und dem NAO das Request-Reply-Protokoll für ZeroMQ nahe. Dies stellt sicher, dass zu jeder Anfrage genau eine Antwort wieder beim Absender eingeht. Dadurch weiß der Sender auch, wann eine Aktion abgeschlossen ist.
Problematisch an dem Request-Reply-Pattern in ZeroMQ ist, dass während eine Anfrage läuft, keine weitere Anfrage gestellt werden kann. Da bestimmte Aktionen, wie zum Beispiel das Aufstehen, sehr viel Zeit beanspruchen (ca. 5 Sekunden), wäre der NAO für diese Zeit überhaupt nicht ansprechbar. Dadurch wäre es zum Beispiel nicht mögliche beide Hände des NAOs gleichzeitig zu öffnen, sondern müsste eine nach der anderen öffnen.

[1] vgl. [Alde2014d]

Die sequenzielle Ausführung von Aktionen führt dazu, dass kurzzeitige Statusabfragen sehr hohe Latenzzeiten erreichen können, wenn der NAO gerade eine lang andauernde Aufgabe durchführt.
Aus diesem Grund wurde das Design darauf ausgelegt, mehrere Worker-Threads für die Ausführung der Anfragen zu nutzen. Die einzelnen Worker können bis zur Beendigung ihrer Aufgabe blockiert sein, ohne die gesamte Anwendung dadurch aufzuhalten.

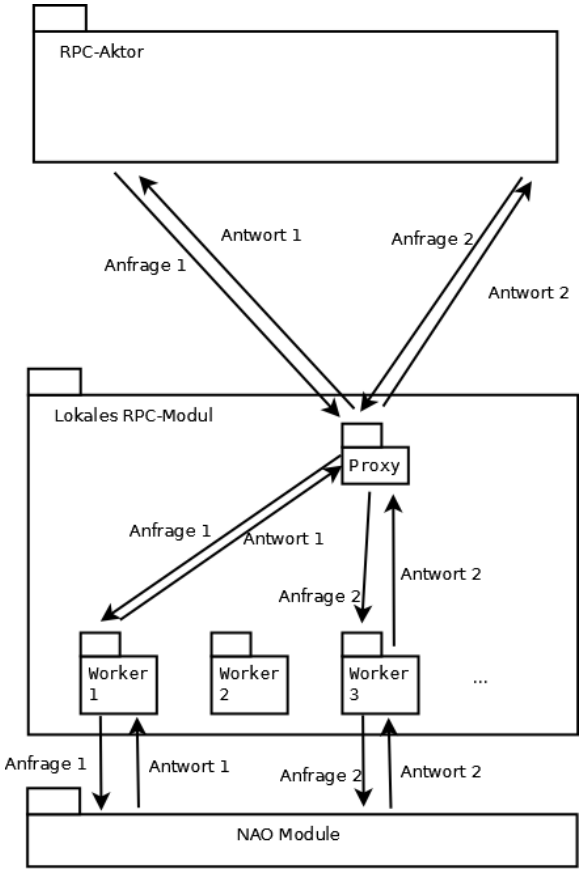

Abb. 4 Kontextsicht des RPC-Moduls

Anhand des Diagramms wird klar, dass das Design nicht dem klassischen Request-Reply-Protokoll folgt, da mehrere Anfragen zur gleichen Zeit im RPC-Modul eingehen können. Trotzdem soll jeder Aufruf für sich garantiert eine Antwort für den Aufruf erhalten. ZeroMQ bietet ein „Extended Request-Reply“[1]-Pattern als Load-Balancing-Pattern an. In diesem Pattern werden die Worker-Threads und der Empfangs-Thread ausschließlich über ZeroMQ-Sockets synchronisiert. Das Implementierungsschema mithilfe von ZeroMQ-Sockets wird in der folgenden Abbildung verdeutlicht.

[1] [Zero2013]

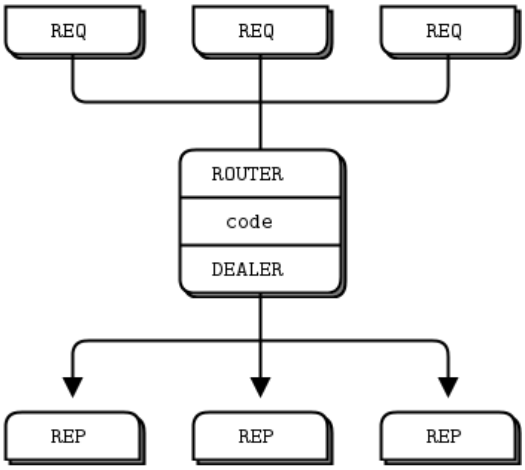

Abb. 5 Extended Request-Reply[1]

ZeroMQ hat folgendes Codebeispiel veröffentlicht, welches die REP-Sockets als Worker-Threads im gleichen Prozess realisiert. Hier der wesentliche Teil:

```
1.  int main() {
2.      zmq::socket_t clients (context, ZMQ_ROUTER);
3.      clients.bind ("tcp://*:5555");
4.      zmq::socket_t workers (context, ZMQ_DEALER);
5.      workers.bind ("inproc://workers");
6.      zmq::proxy (clients, workers, NULL);
7.  }
8.  void *worker_routine (void *arg) {
9.      zmq::socket_t socket (*context, ZMQ_REP);
10.     socket.connect ("inproc://workers");
11. }
```

Codebeispiel zu einem Multithreaded Request-Reply Server[2]

Bei diesem Schema verwendet ZeroMQ die Intraprozessnachrichten um die einzelnen Anfragen der REQ-Sockets auf die REP-Sockets zu verteilen. Diese Nachrichten werden vom DEALER-Socket nach Round-Robin gleichmäßig auf alle verfügbaren REP-Sockets aufgeteilt. Hat ein REP-Socket die Bearbeitung einer Aufgabe abgeschlossen, dann wird dessen Antwort wieder an den anfragenden REQ-Socket zurückgeleitet.
Um das Zurückleiten zu ermöglichen, schreibt ein ROUTER-Socket beim Empfangen von Nachrichten den Absendersocket der Nachricht in die Headerframes der ZeroMQ-Nachricht. Diese Header werden beim Versenden über den DEALER-Socket nicht geändert und die REP-Sockets fügen diese Header ihrer Antwort ebenfalls bei. Dadurch kann der

[1] [Zero2013]
[2] vgl. [Zero2013b]

Router-Socket beim weiterleiten der Antwort entscheiden, an welchen Socket, die Antwort gesendet werden muss.
Das Pattern spezialisiert sich auf die Lastenverteilung von sehr vielen kurz laufenden Aufgaben auf mehrere Worker und setzt dies performant um. Für den Einsatz im NAO-Modul eignet sich dieses Pattern jedoch nicht, weil die Aufgaben gleichmäßig auf alle Worker verteilt werden. Sehr lang andauernde Aufrufe können im NAO einige Sekunden in Anspruch nehmen. Round-Robin für die Verteilung der Aufgaben auf die Worker-Threads zu nutzen, führt dazu, dass die Ressourcen des NAOs in manchen Fällen nicht optimal genutzt werden. Die Problematik wird im folgenden Diagramm kurz verdeutlicht.

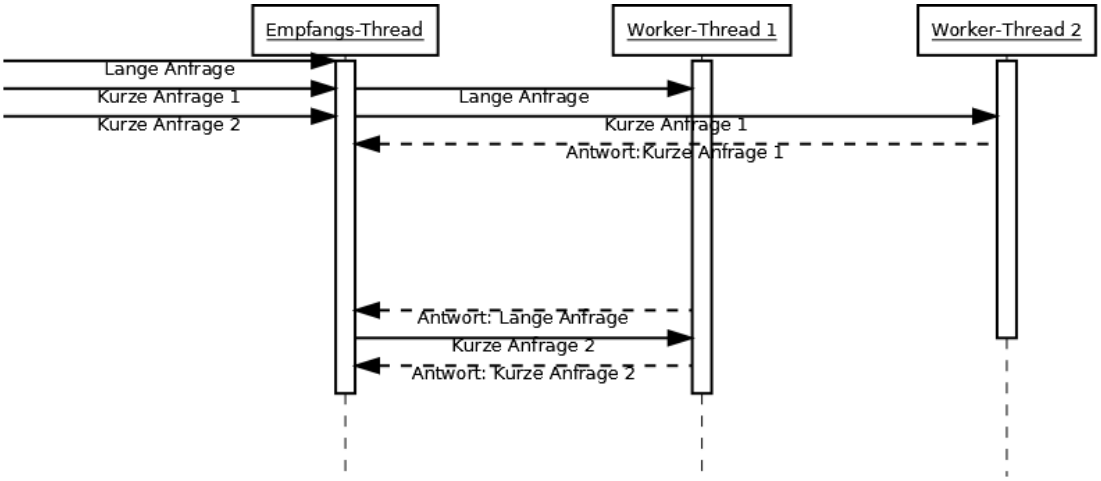

Abb. 6 Lastenverteilung mit Round-Robin

Im Diagramm werden die Anfragen auf zwei Worker-Threads mithilfe von Round-Robin aufgeteilt. Die erste lang laufende Anfrage wird an den Worker-Thread 1 geleitet und die nachfolgende kurze Anfrage an Worker-Thread 2. Sobald der Worker-Thread 2 seine Anfrage abgearbeitet hat, kann er die nächste Anfrage bearbeiten. Laut Round-Robin wird die nächste Anfrage jedoch an Worker-Thread 1 geleitet. Dieser ist aber noch beschäftigt und so verzögert sich die Verarbeitung der zweiten kurzen Anfrage, obwohl ein freier Worker-Thread verfügbar ist.

Das Problem wird umgangen, indem für die Aufteilung der Arbeit auf die einzelnen Arbeiter eine leichte Abwandlung des Work-Pulling-Patterns[1] implementiert wird. Das Work-Pulling-Pattern sieht vor, dass sich jeder Worker beim Empfangs-Thread (auch Master genannt) meldet, sobald er anfragen entgegen nehmen kann. Beim Master eingehende Anfragen werden anschließend nur auf die Worker verteilt, die sich beim Master als bereit gemeldet haben. Sollte kein Worker bereit sein, dann werden die Anfragen in eine Warteschlange eingereiht, bis ein Worker bereit ist.

[1] vgl. [Wyatt2009]

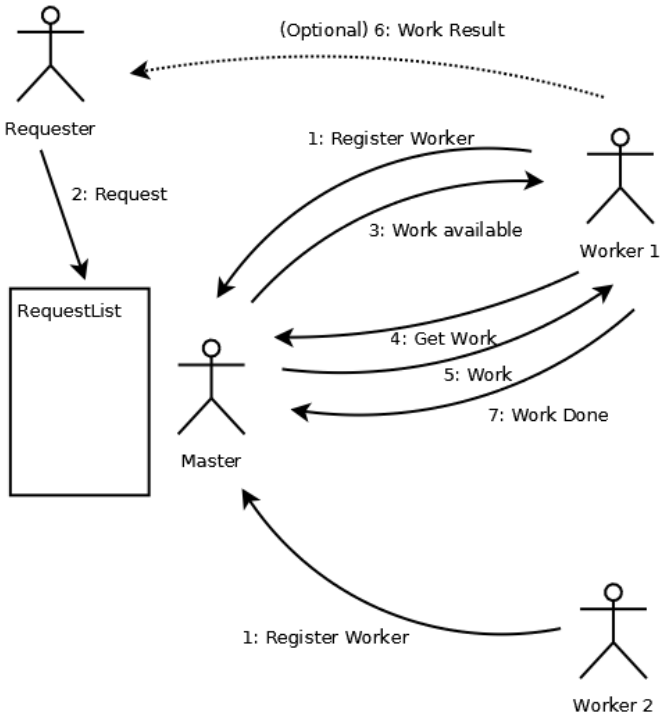

Abb. 7 Work-Pulling-Pattern

Das Work-Pulling-Pattern wurde für diese Anwendung in folgenden Punkten abgewandelt: Der Master startet alle Worker selbst, weil sich die Anzahl der Worker zur Laufzeit nicht ändern wird. Dadurch entfällt das Registrieren der Worker am Master. Die Nachrichten 3-4 fallen weg. Stattdessen wird direkt die Anfrage (Nachricht 5) an den freien Worker gesendet. Die Nachrichten 3-4 dienen im Work-Pulling-Pattern dazu, nicht mehr erreichbare Worker zu erkennen. Im NAO-Modul läuft die gesamte Kommunikation innerhalb eines Prozesses. Die Worker sind somit unabhängig vom Netzwerkzustand immer erreichbar. Außerdem werden Antworten(Nachricht 6) nicht direkt zum Absender der Anfrage gesendet, sondern an den Master geschickt, welcher die Antwort dann an den korrekten Empfänger weiterleitet. Dies ist notwendig, damit die Worker das Request-Reply-Protokoll einhalten, welches von deren REP-Sockets vorgegeben wird. Die 7. Nachricht fällt ebenfalls weg, weil alle Antworten über den Master geleitet werden und der Eingang einer Antwort beim Master bedeutet, dass der Worker seine Anfrage abgearbeitet hat und neue Anfragen entgegennehmen kann.

Der DEALER-Socket-Typ unterstützt ausschließlich Round-Robin zur Verteilung der Aufgaben auf mehrere REP-Sockets. Deshalb wird für jeden Worker ein eigener Socket geöffnet und die Verteilung der Anfragen von dem Modul selbst übernommen.

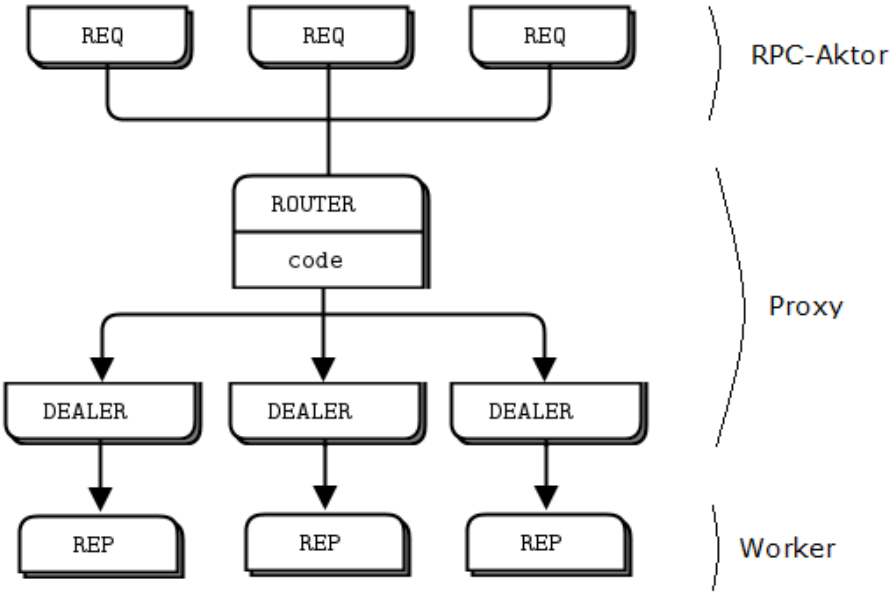

Abb. 8 Extended Request-Reply mit Work-Pulling-Pattern

Bei der Initialisierung des RPC-Moduls wird ein Thread für den Proxy gestartet, welcher wiederum die Worker-Threads startet. Die Worker öffnen den REP-Socket und benachrichtigt den Proxy-Thread über einen Semaphor darüber. Dies ist die einzige Stelle, an welcher explizite Synchronisation notwendig ist, denn der DEALER-Socket kann erst mit dem REP-Socket verbunden werden, wenn der REP-Socket geöffnet wurde. Der Proxy verbindet seine DEALER-Sockets erst, wenn alle Worker ihre REP-Sockets geöffnet haben.

Das angepasste Work-Pulling-Pattern wird im Proxy wie folgt umgesetzt:

```
1.  while (runProxy) {
2.     zmq::poll(items, itemsize, -1);
3.
4.     if ((items[0].revents & ZMQ_POLLIN) && !availableWorkers.empty()) {
5.       socket_t* freeWorker = availableWorkers.front();
6.       routeMessage(frontend, *freeWorker, message);
7.       availableWorkers.pop();
8.     }
9.     for (int c = 1; c < itemsize; ++c) {
10.      if (items[c].revents & ZMQ_POLLIN) {
11.        socket_t* receiver = workers[c-1];
12.        routeMessage(*receiver, frontend, message);
13.        availableWorkers.push(receiver);
14.      }
15.    }
16. }
```

Codeausschnitt für das Work-Pulling-Pattern

Die DEALER-Sockets, die mit den Workern verbunden sind, werden in dem *vector workers* gehalten. Die Liste der verfügbaren Worker wird in der Queue *availableWorkers* gehalten. Sollte über den ROUTER-Socket eine Anfrage beim Modul eingehen und ein Worker verfügbar sein, dann wird die Anfrage an den ersten freien Worker in der Queue gesendet. Anschließend werden alle DEALER-Sockets überprüft. Sollte auf einem Socket eine Antwort eingegangen sein, wird diese über den ROUTER-Socket an den korrekten Empfänger weitergeleitet und der DEALER-Socket in die Queue der verfügbaren Worker eingereiht und damit als bereit markiert.

Um eine Methode aufzurufen, muss sich der Worker über den *ALBroker* die entsprechende *ALModule*-Instanz holen. Anschließend kann auf dem Modul die Methode

```
execute(const std::string& method, const AL::ALValue& params, AL::ALValue& result)
```

aufgerufen werden. Diese Methode hat den Parametertyp *ALValue*. *ALValue* ist eine Klasse, die in einer *union* Werte von allen möglichen Datentypen aufnehmen kann. Im Gegensatz zu einer *union* in C++, kann man *ALValue* abfragen, welchen Typ die *union* momentan hat. Um also alle Aufrufe im NAO auszuführen, muss dieser *ALValue*-Typ auf eine serialisierbare Protobuf-Klasse abgebildet werden.
Die folgende Nachrichtendefinition kann alle möglichen *ALValue*-Objekte darstellen.

```
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. }
```

Diese *MixedValue*-Klasse kann für die Serialisierung/Deserialisierung von Parameter und Rückgabewert verwendet werden. Ein vollständiger Aufruf besteht jedoch aus Modulname, Methodenname und Parameter. Während des Aufrufs können auch Exceptions geworfen werden, die als Fehlermeldungen an den Aufrufer zurückgesendet werden müssen. Daraus folgt die folgende Definition für Anfrage- und Antwortnachricht.

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

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

Die Worker-Threads des RPC-Moduls erhalten vom Proxy die Anfragen, welche sie zu einem *HAWActorRPCRequest* deserialisieren. Über den *ALBroker* erhält der Worker-Thread eine Referenz auf das Modul, dessen Methode aufgerufen werden soll.

Dann werden die *MixedValue*-Objekte in *ALValue*-Objekten konvertiert und die Methode aufgerufen. Sollte ein Fehler aufgetreten sein, wird dieser behandelt, indem die Fehlernachricht in das *error*-Feld des Antwortobjekts geschrieben wird. Sollte kein Fehler aufgetreten sein, wird das Ergebnis von einem *ALValue*-Objekt in ein *MixedValue*-Objekt konvertiert und in das *returnval*-Feld des Antwortobjekts geschrieben.
Das Antwortobjekt wird serialisiert und über an den ZeroMQ-Socket des Workers übergeben. Die Nachricht wird an den Proxy gesendet und dieser leitet sie weiter an den ursprünglichen Absender.

3.2.2 Videomodul

In vielen Anwendungen im Bereich der Roboter spielt die Bildverarbeitung eine zentrale Rolle. So wird Gesichts- und Gestenerkennung dazu verwendet, dem Roboter Anweisungen zu erteilen und die visuelle Erkennung von Hindernissen dient für Roboter in Bewegung als wichtige Informationsquelle.

Damit der NAO jederzeit Bildinformationen an die Anwendung liefern kann, wird ein separates Videomodul implementiert. Das NAOqi *ALVideoDevice* bietet zwei Möglichkeiten, Bilddaten abzufragen. Die Methode *getImageRemote()* liefert eine Kopie des Bildspeichers und die Methode *getImageLocal()* liefert eine Referenz auf den tatsächlichen Bildspeicher und verlangt von dem Modul, dass der Speicher nach der Verarbeitung wieder mit *releaseImage()* freigegeben wird. NAOqi bietet keine Möglichkeit an, einen Bildstream zu erhalten, bzw. benachrichtigt zu werden, wenn ein neues Bild im Kameraspeicher verfügbar ist. Will eine Anwendung einen Stream aktueller Kamerabilder haben, so muss sie manuell in regelmäßigen Intervallen das aktuelle Bild vom *ALVideoDevice* abfragen. Dieses Intervall ist abhängig von der eingestellten Bildrate. Das regelmäßige Abfragen der Bilddaten könnte mit Hilfe von *sleep*-Aufrufen im Videomodul realisiert werden, die an die aktuelle Bildrate angepasst sind. Die Bilddaten könnten dann über einen ZeroMQ-Publish-Socket im Netzwerk an alle verteilt werden, die die Bilddaten benötigen.
Der Ansatz hat jedoch ein paar entscheidende Nachteile. Da das Modul immer aktiv ist, würde es auch dann Bilddaten abfragen und komprimieren, wenn diese gar nicht angefragt werden. Damit würde das Modul die CPU des NAOs durch überflüssige Operationen auslasten. Die Übertragung dieser Bilddaten würde zudem das Funknetzwerk belasten.
Um auch Aufrufe entgegennehmen zu können, die Konfigurationseinstellungen der Kamera ändern, müsste das Modul einen extra Socket nur für diese Anfragen öffnen.

Um sowohl die CPU des NAOs, als auch das Funknetzwerk zu entlasten, wird das Videomodul so implementiert, dass alle Anfragen über einen Reply-Socket angenommen werden. Der Timer, welcher dafür sorgt, dass regelmäßig Bilddaten abgefragt werden, wird vom Videoaktor implementiert. Dieser sendet in regelmäßigen Intervallen Anfragen an das Videomodul, um Bilddaten zu erhalten und speichert das aktuelle Bild zwischen. Die Anfragen werden nur gesendet, wenn innerhalb der letzten x Sekunden ein Aktor Bilddaten angefragt hat. Dadurch wird sowohl das Netzwerk, als auch die CPU des NAOs entlastet,

wenn kein Bildstream benötigt wird. Außerdem hat das Videomodul dadurch eine simplere Schnittstelle. Fragt nun ein Aktor Bilddaten an, kann ihm der Videoaktor direkt die zwischengespeicherten Bilddaten als Antwort senden.

Um die Latenzzeit beim Bildtransfer so weit wie möglich zu reduzieren, müssen Kopien von großen Speicherbereichen vermieden werden. Deshalb wird das Bild mit *getImageLocal()* angefordert und direkt aus dem internen Bildspeicher heraus komprimiert und in einen während der Initialisierung angelegten Puffer geschrieben.

Das Nachrichtenformat zwischen dem Videomodul und der Aktorenseite muss folgende Eigenschaften haben. Mit Nachrichten an das Videomodul müssen die Kameraparameter Bildrate, Farbraum und Auflösung verändert werden können und das reine Abfragen von Bildern muss möglich sein. Die Antwortnachrichten müssen dem Aufrufer Fehler und Bilddaten liefern können. Zur Umsetzung der Anforderungen werden folgende Nachrichtendefinitionen genutzt.

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

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

Wenn Kameraparameter geändert werden müssen, dann werden die entsprechenden Felder der CamRequest-Nachricht gesetzt. Soll ein Bild abgefragt werden, dann werden alle Felder der Nachricht leer gelassen. Protobuf erzeugt daraus eine 0-Byte große Nachricht. Dadurch müssen für das Anfragen von Bildern nur die ZeroMQ-Header übertragen werden.

3.3 Aktoren-Schnittstelle

Die Akka-Aktoren haben die Aufgabe zwischen den NAO-Modulen und dem restlichen Aktorensystem zu vermitteln und Anfragen des Aktorensystems auf Anfragen an die NAO-Module abzubilden. Mit der Verwendung des NAOqi-APIs als Schnittstelle des Aktorensystems, entfällt das Abbilden von Anfragen größtenteils. Die meisten Anfragen müssen lediglich an das RPC-Modul im NAO durchgereicht werden. Sollte ein anderer Robotertyp angebunden werden, dann müsste die Funktionalität des NAOqi-APIs auf das API des speziellen Roboters abgebildet werden.
Dies ist jedoch beim Videoaktor notwendig, da das Videomodul im NAO eine eigene Schnittstelle implementiert.

Als Nachrichtenformat für die Schnittstelle wird folgende Klasse verwendet:

```
case class RobotRequest(module:String, method:String, params:Any*)
```

Ein Aufruf besteht somit aus Modulname, Methodenname und beliebig vielen Parametern. Jeder Aufruf hat einen möglichen Rückgabewert. Dessen Typ ist wie folgt definiert.

```
type Response = Try[Any]
```

Das *Try* gibt der Anwendung die Möglichkeit, auf Fehler zu reagieren. Sollte während des Aufrufs ein Fehler aufgetreten sein, dann hat das *Try* den Typ *Failure* und enthält die aufgetretene Exception. Tritt keine Exception auf, hat *Try* den Typ *Success* und enthält den Rückgabewert vom Typ *Any. Any* ist der Supertyp aller Scala-Klassen und kann somit jeden möglichen Rückgabewert darstellen. Hat die aufgerufene Methode keinen Rückgabewert, dann wird *Unit* zurückgegeben. *Unit* ist das Scala-Äquivalent zu *void* und besitzt im Gegensatz zu Java einen Wert, nämlich *()*.

Der Front-End-Aktor nimmt alle Anfragen entgegen. Dieser Aktor ist im Aktorensystem der Stellvertreter des NAOs und alle Anfragen, die an den NAO gerichtet sind, müssen an diesen Aktor gesendet werden. Um die NAO-Anbindung skalierbar zu halten, sollte der Front-End-Aktor die Anfragen nicht selbst bearbeiten, da er nur eine Anfrage zur Zeit bearbeiten kann. Stattdessen verteilt der Front-End-Aktor die Anfragen auf die zuständigen Unter-Aktoren. Diese Unteraktoren können beliebig feingranular auf Teile des APIs spezialisiert sein. Beispielsweise kann für jedes NAOqi-Modul ein zuständiger Unteraktor eingesetzt werden. Durch das Hinzufügen von weiteren Unteraktoren und Worker-Threads im NAO kann die Anbindung theoretisch beliebig weit skaliert werden. Beim Skalieren muss jedoch berücksichtigt werden, dass der NAO einzelne Anfragen langsamer bearbeiten kann je mehr Anfragen parallel durchgeführt werden, da für jede einzelne weniger CPU-Zeit zur Verfügung steht. Dadurch ist der Umfang in welchem die Schnittstelle skaliert werden kann, durch die Rechenleistung des NAOs beschränkt.

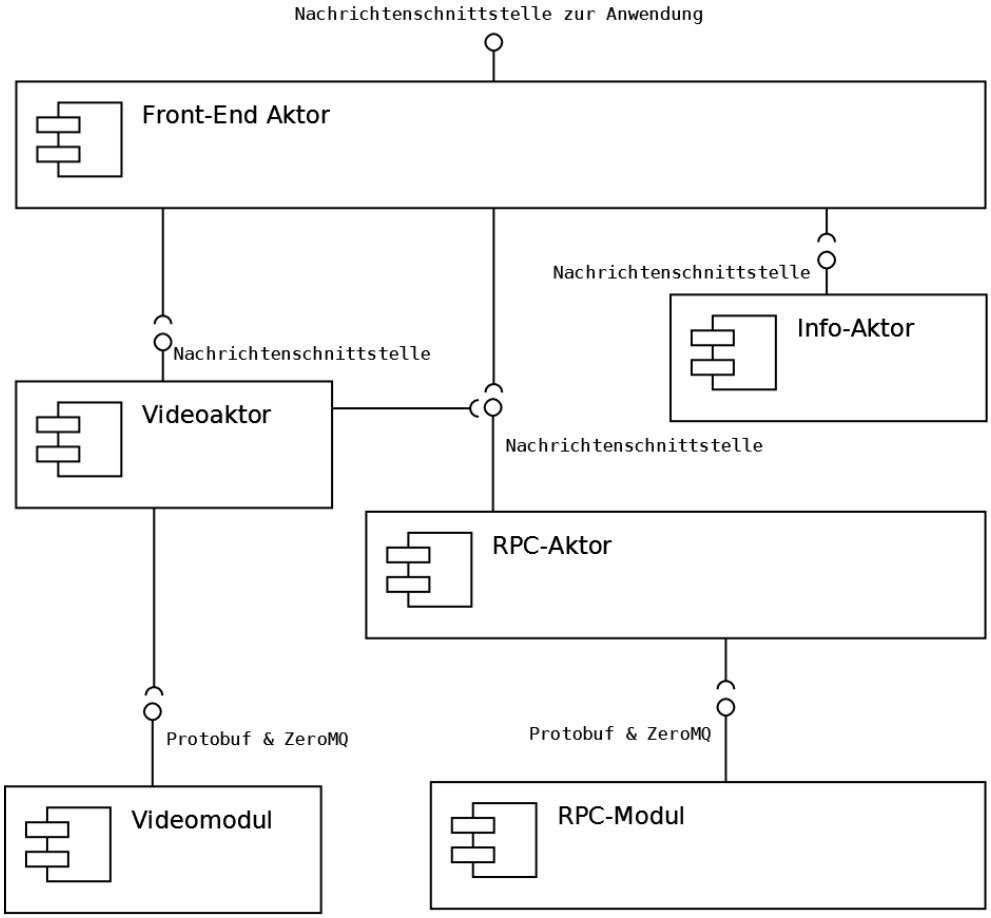

Abb. 9 Design der Aktoren-Schnittstelle

Das RPC-Modul implementiert das NAOqi-API für alle Module. Deshalb kann ein generischer RPC-Aktor alle Anfragen bearbeiten, die nicht an das Videomodul, oder das Info-Modul gerichtet sind. Das Videomodul implementiert nicht die volle Funktionalität des *ALVideoDevice* aus dem NAOqi-API. Es unterstützt lediglich das Abfragen und Setzen von Bildrate, Farbraum und Auflösung, sowie das Abfragen von Bilddaten. Sollte der Videoaktor eine Anfrage erhalten, die das Videomodul nicht unterstützt, wird diese an den generischen RPC-Aktor weitergeleitet und vom RPC-Modul bearbeitet.
Das Infomodul ist ebenfalls als eigenständiger Unteraktor des Front-End-Aktors umgesetzt. Dieses kommuniziert nicht mit den NAO, sondern stellt Informationen über den NAO bereit, wie zum Beispiel eine Liste der unterstützten Module. Die vom Infomodul angebotene Information kann nach Bedarf um beliebige weitere Informationen erweitert werden. Denkbar wäre unter anderem der genaue Name des Robotermodells, eine Auflistung der möglichen Bewegungen und vorhandenen Aktuatoren.

3.3.1 RPC-Aktor

Die Aufgabe des RPC-Aktors ist es, die generische Schnittstelle, welche das RPC-Modul im NAO anbietet, auf das Aktorensystem zu adaptieren. Dazu muss das RPC-Modul die ZeroMQ/Protobuf-Schnittstelle des RPC-Moduls nutzen können. Für Scala gibt es zwei Möglichkeiten, ZeroMQ zu nutzen. Es gibt das native Java-Binding von ZeroMQ, welches von Scala aus genutzt werden kann und die gleiche Abstraktion bietet, wie das C++-Pendant. Akka hat passend zu den Aktoren eine eigene ZeroMQ-Schnittstelle entwickelt, die auf dem Java-Binding aufsetzt. Dabei wird für jeden ZeroMQ-Socket ein Aktor erstellt, welcher den Socket repräsentiert. Wird eine Nachricht vom Typ *ZMQMessage* an diesen Aktor gesendet, dann wird die Nachricht über den Socket gesendet und sendet Antworten als *ZMQMessage*-Objekte zurück.

```
1.  class PongActor extends Actor {
2.    val socket = ZeroMQExtension(context.system).newRepSocket(Array(
3.      Listener(self),
4.      Bind("tcp://localhost:12345")
5.    ))
6.
7.    def receive = {
8.      case msg:ZMQMessage => if (msg.firstFrameAsString == "ping")
9.        socket ! ZMQMessage("pong".toByteArray)
10.   }
11. }
```

Codebeispiel: Einfacher Akka-Actor mit ZeroMQ-Anbindung

Der Vorteil an dieser Schnittstelle ist, dass die Kommunikation mit dem Socket über die Nachrichtenschnittstelle von Akka erfolgt und sich nahtlos in die Aktorenumgebung einfügt. Durch die Aktorenschnittstelle können beliebig viele Aktoren einen einzigen Socket gleichzeitig nutzen.
Bei dem Java-Binding hingegen müsste für den Socket ein eigener Thread gestartet werden, um den RPC-Aktor beim Warten auf Nachrichten nicht zu blockieren. Außerdem sind ZeroMQ-Sockets nicht Thread-Safe und könnten nur mithilfe von Locking von mehreren Threads aus genutzt werden. Im Akka-Binding sammelt der zuständige Aktor des ZeroMQ-Sockets alle Anfragen und führt sie sequenziell aus. Dadurch wird der Socket nur von einem Thread verwendet und es ist keinerlei Locking notwendig.
Aufgrund der einfachen Benutzung und der Vorteile des Akka-Bindings wird dies für die Aktoren verwendet, die mit ZeroMQ-Sockets kommunizieren müssen.

Da Protobuf Java als Zielsprache unterstützt, und Scala Java-Klassen benutzen kann, wird Protobuf über die generierten Nachrichtenklassen für Java in den Aktor eingebunden.

Um das RPC-Modul optimal auszulasten, wird im RPC-Aktor ein Worker-Pool verwendet. Jeder dieser Worker nimmt eine Anfrage entgegen, sendet diese über seinen ZeroMQ-Socket an das NAO-Modul, wartet auf eine Antwort vom NAO-Modul und sendet diese Antwort an den anfragenden Aktor zurück. Da hier ein Worker einen Worker im RPC-Modul

repräsentiert, sollte die Anzahl der Worker im Modul und im Aktor übereinstimmen. Für die Lösung der Anfragenverteilung auf die einzelnen Worker bietet sich auch hier, wie schon im RPC-Modul, das Work-Pulling-Pattern[1] an.

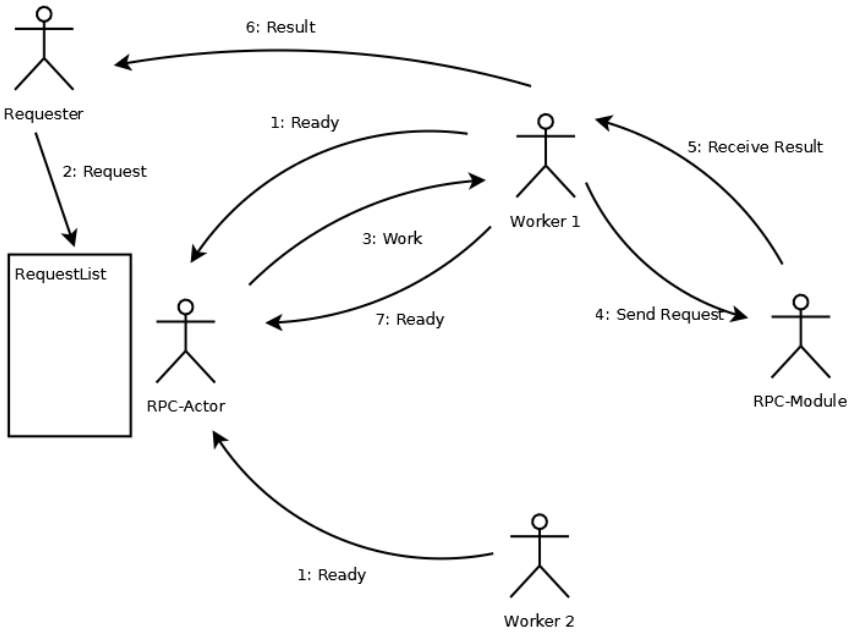

Abb. 10 Work-Pulling-Pattern im RPC-Aktor

Im Gegensatz zu den Workern im RPC-Modul, sind die Worker-Aktoren nicht an ein Request-Reply-Pattern zum Anfragenverteiler gebunden und können die Antworten direkt an die Aktoren senden, welche die Anfrage abgesendet haben. Dadurch kommt die Implementierung im RPC-Aktor dem eigentlichen Work-Pulling-Pattern näher, als die Implementierung im RPC-Modul. Lediglich die Nachrichten „Work Available“ vom Master und das anschließende „Get Work“ vom Worker fehlen. Der Zweck dieser Nachrichten im Work-Pulling-Pattern ist es, nicht mehr erreichbare Aktoren zu entdecken. Die Worker-Aktoren führen triviale Aufgaben durch und warten den größten Teil der Zeit auf eingehende Anfragen, oder auf eine Antwort vom RPC-Modul. Deswegen erzielt man keinen Performancegewinn dadurch, dass man einzelne Worker auf unterschiedliche Rechner im Netzwerk auslagert. Laufen alle Worker im gleichen Prozess, wie der Master, dann sind die beiden Nachrichten überflüssig.

[1] vgl. [Wyatt2009]

```
1.  val idleWorkers:Queue[ActorRef] = new Queue()
2.  val workQueue:Queue[Job] = new Queue()
3.
4.  def receive = {
5.    case Ready => {
6.      if (workQueue.isEmpty)
7.        idleWorkers.enqueue(sender)
8.      else
9.        sender ! workQueue.dequeue
10.   }
11.
12.   case job:Job => {
13.     if (idleWorkers.isEmpty)
14.       workQueue.enqueue(job)
15.     else
16.       idleWorkers.dequeue ! job
17.   }
18. }
```

Codeausschnitt: Work-Pulling-Pattern im RPC-Aktor

Der RPC-Aktor sammelt alle eingehenden Anfragen in seiner *workQueue*, wenn keine Worker-Aktoren frei sind. Die Klasse Job enthält alle Felder, welche die Interfaceklasse *RobotRequest* enthält und enthält zusätzlich eine Referenz auf den absendenden Aktor. Diese nutzt der Worker, um die Antwort der Anfrage direkt an den Absender zurück zu senden. Aus dem Code sieht man zudem, dass der gesamte Ablauf vollständig Nachrichtengesteuert ist und kein Blockieren oder aktives Warten beinhaltet.

```
1.  def receive = {
2.    case Job(requestor, module, method, params) => {
3.      socket ! ZMQMessage(buildRequest(module, method, params).toByteArray)
4.      context.become {
5.        case m:ZMQMessage => {
6.          requestor ! convertResponse(m)
7.          context.parent ! Ready
8.          context.unbecome
9.        }
10.     }
11.   }
12.   case Connecting => context.parent ! Ready
13. }
```

Codeausschnitt: Work-Pulling-Pattern im Worker-Aktor

Der Worker-Aktor meldet sich nach dem Start beim RPC-Aktor als bereit, sobald sich der ZeroMQ-Socket mit dem Zielsocket im RPC-Modul verbindet. Erhält der Worker eine Anfrage, so wird mit *buildRequest()* die Protobuf-Datenstruktur befüllt und anschließend in ein Byte-Array serialisiert und über den Socket an den NAO gesendet. Sobald die Antwort vom RPC-Modul eingeht, wird diese mit *convertResponse()* in den Antworttypen *Try[Option[Any]]* konvertiert und an den Absender der Anfrage zurückgesendet. Gleichzeitig meldet sich der Worker beim RPC-Aktor bereit, neue Anfragen zu bearbeiten.

3.3.2 Videoaktor

Eine naive Implementierung des Aktors für das Videomodul könnte wie folgt aussehen.
Der zuständige Videoaktor erhält die Anfrage, das aktuelle Bild zu übermitteln. Dieser leitet die Anfrage an das Videomodul im NAO weiter. Das Videomodul fragt das aktuelle Bild vom *ALVideoDevice* ab, komprimiert es und sendet es an den Videoaktor zurück. Dieser sendet das empfangene Bild an den anfragenden Aktor(nachfolgend als Anwendung bezeichnet) als Antwort zurück.

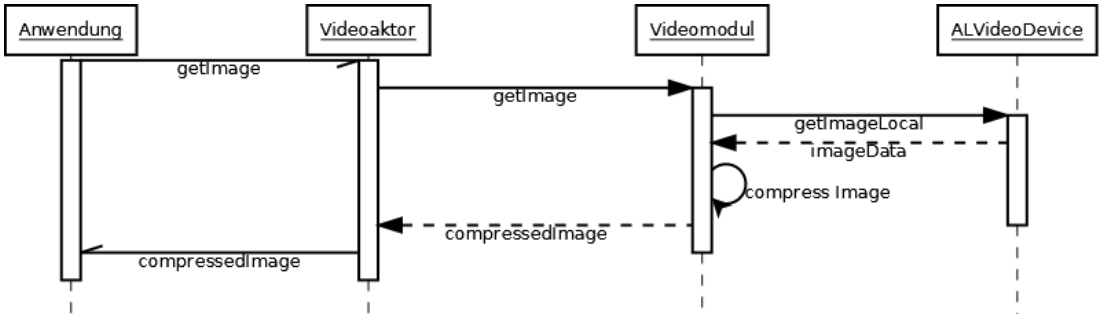

Abb. 11 Bildabfrage nach traditionellem RPC-Schema

Bei diesem Diagramm ist zu beachten, dass die letzten beiden Antwortnachrichten einen Transfer von Bilddaten über eine Netzwerkschnittstelle darstellen, was zu einer zusätzlichen Verzögerung führt. Insgesamt hat dieses Design zur Folge, dass eine Anwendung sehr lange auf die angeforderten Bilddaten warten muss, da sie zur Zeit der Anfrage noch nicht existieren. Gibt es nun mehrere Anwendungen, die vom Videoaktor Bilddaten erwarten und würde der Videoaktor für jede Anfrage eine Anfrage an das Videomodul stellen, dann würde die Übertragungsdauer der Bilder bei zu vielen Anfragen dafür sorgen, dass sich die Anfragen ansammeln, weil der Videoaktor sie nicht rechtzeitig bearbeiten kann.

Um diese Probleme zu vermeiden, werden die Bilddaten zeitgesteuert in regelmäßigen Intervallen vom NAO abgefragt. Das Intervall ist an die aktuelle Bildrate angepasst. Die abgefragten Bilddaten werden dann solange im Videoaktor vorgehalten, bis neue Bilddaten abgefragt werden. Fragt eine Anwendung nun Bilddaten ab, erhält sie die zwischengespeicherten Bilddaten. Der Videoaktor erhält dazu einen Unteraktor, den *ImageFetcher*. Dieser ist ausschließlich dazu da, auf Timeout-Nachrichten zu reagieren, indem er Bilddaten vom Videomodul abfragt. Wenn die Bilddaten erfolgreich übertragen wurden, sendet er die aktuellen Bilddaten an den Videoaktor, welcher seine zwischengespeicherten Bilddaten durch die neuen Bilddaten ersetzt. So kann der Videoaktor weiterhin Bildanfragen bearbeiten während der *ImageFetcher* auf die Übertragung der Bilddaten vom NAO wartet.

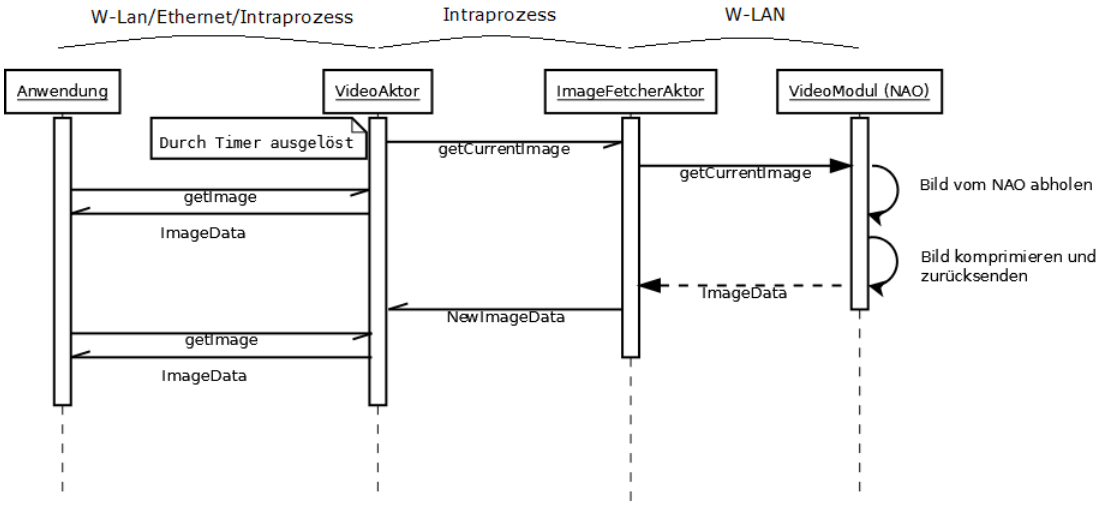

Abb. 12 Kommunikationsschema des Videoaktors

Im Diagramm sind zusätzlich die Verbindungsmöglichkeiten zwischen den Knoten eingetragen. Lediglich die Kommunikation zwischen dem *ImageFetcher* und dem Videomodul muss über eine bandbreitenschwache W-LAN-Verbindung übertragen werden. Die Anwendung, welche die Bilddaten benötigt, kann auf dem gleichen Rechner laufen, wie der Videoaktor, oder auf einem entfernten Rechner im Ethernet-Netzwerk. Dadurch kann der Videoaktor Bilddaten schneller zur Anwendung übertragen, als das Videomodul die Bilddaten zum *ImageFetcher* übertragen kann. So kann der Videoaktor mehrere Anwendungen gleichzeitig mit aktuellen Bilddaten versorgen, ohne dass sich die Bildanfragen beim Videoaktor aufstauen.

Die Timer-basierte Lösung hat ohne weitere Änderungen das Problem, dass kontinuierlich Bilddaten vom NAO zum Aktor übertragen werden, auch wenn die Anwendung keine Bilddaten benötigt. Dadurch wird der NAO ständig durch die Kompression und das Versenden der Bilddaten ausgelastet. Außerdem wird das W-LAN durch die Bildübertragungen ausgelastet. Um dem entgegen zu wirken, deaktiviert sich der *ImageFetcher*, wenn über eine bestimmte Anzahl an Frames keine Bildanfrage beim Videoaktor eingegangen ist.

Zusätzlich zum Abfragen von Bilddaten unterstützt der Videoaktor auch Methoden, um die Bildrate, den Farbraum und die Auflösung zu ändern und abzufragen. Will eine Anwendung die aktuellen Kameraparameter abfragen, muss der Videoaktor entweder jedes Mal die Anfrage an den RPC-Aktor weiterleiten, damit das RPC-Modul im NAO die Daten abfragt und an die Anwendung zurücksendet. Alternativ kann das Videomodul diese Kameraparameter speichern. Damit kann das Videomodul schneller auf solche Anfragen antworten, da kein RPC-Aufruf im NAO für jede Anfrage notwendig ist. Für die Umsetzung des Videoaktors wird der letztere Ansatz gewählt, weil sich zum einen die

Kameraparameter nicht ständig ändern sollten und zum anderen der Videoaktor zumindest die aktuelle Bildrate kennen muss, um den Timer für den *ImageFetcher* korrekt einzustellen. Deshalb sendet der Videoaktor bei der Initialisierung 3 Anfragen an den RPC-Aktor, und die entsprechenden Parameter zu erhalten. Anhand dieser wird dann der Timer für den *ImageFetcher* konfiguriert und gestartet. Die Werte werden im Zustand des Aktors gespeichert.
Bei jeder Änderung der Kameraparameter werden diese erneut vom RPC-Aktor abgefragt.

Das Videomodul unterstützt nur einen Teil der Funktionalität des NAOqi *ALVideoDevice*-Moduls. Alle Anfragen, die nicht direkt vom Videomodul unterstützt werden, werden deshalb an den RPC-Aktor weitergegeben, damit das *ALVideoDevice*-Modul trotzdem vollständig unterstützt wird.

3.3.3 Info-Aktor

Der Info-Aktor kommuniziert nicht mit dem NAO und stellt lediglich generelle Informationen über den NAO zur Verfügung. Das NAOqi-API enthält kein Info-Modul. Dieses ist jedoch notwendig, wenn Anwendungen für verschiedene Robotertypen geschrieben werden. Die Anwendung muss den Roboter abfragen können, um was für eine Art von Roboter es sich handelt und welche Module unterstützt werden. Nur so kann die Anwendung entscheiden, ob der verfügbare Roboter für die Anwendung genutzt werden kann.

```
1.  val result = (nao ? RobotRequest("Info", "getSupportedModules")).
2.    mapTo[Try[Option[List[String]]]]
```

Codebeispiel: Modulliste abfragen

Mit dem eben genannten Aufruf kann die Liste aller unterstützten Module abgefragt werden. Diese Liste kann für jeden Roboter fest im Info-Aktor kodiert sein, sodass für das Abfragen der Module keine Kommunikation mit dem Roboter notwendig ist.
Um zu entscheiden, welche Aufrufe dieses Info-Modul unbedingt unterstützen muss, wäre eine genauere Untersuchung anhand vorhandener Anwendungsfälle notwendig. Diese wurde im Rahmen dieser Arbeit nicht durchgeführt. Stellvertretend für das vollständige Info-Modul, wird deswegen eine minimale Implementierung verwendet. Diese unterstützt nur das Abfragen der verfügbaren Module.

3.4 Anwendungsbeispiel

Im Folgenden wird ein an einem Beispiel erklärt, wie eine Anwendung das API verwenden kann.

```
1. import robots.nao._
2. import akka.actors._
3.
4. Configuration.nao = NaoConfiguration("192.168.1.10")
5. ActorSystem("robotapp").actorOf(Props[RobotActor], "nao")
```

Codebeispiel: Front-End-Aktor konfigurieren und starten

Bevor eine Anwendung den NAO ansteuern kann, muss dessen Front-End-Aktor gestartet werden. Dieser kann auf einer anderen Maschine gestartet werden, als die eigentliche Anwendung. Mit der *actorOf*-Anweisung wird der Front-End-Aktor mit seinen zugehörigen Unteraktoren erstellt. Die erstellte Aktorenhierarchie sieht dann wie folgt aus.

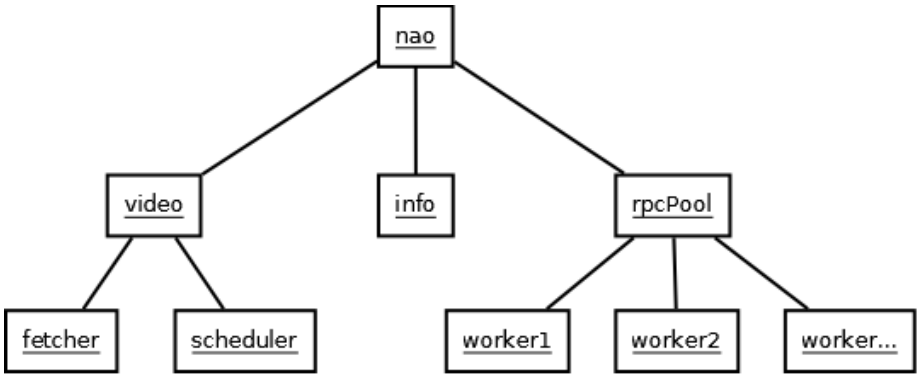

Abb. 13 Aktorenhierarchie

Diese Hierarchie muss genau ein Mal für jeden Roboter aufgebaut werden. Um Anfragen an den Roboter zu stellen, müssen die Anwendungen nur den Front-End-Aktor(hier: nao) im Aktorensystem erreichen können. Dazu muss die Maschine, auf der das Aktorensystem gestartet wurde, nur über das Netzwerk erreichbar sein.
Alle Aktoren implementieren für ihre Kind-Knoten die One-For-One Ausfallstrategie[1]. Sollte also beispielsweise ein Worker-Aktor durch eine unbehandelte Exception sterben, dann startet sein Eltern-Knoten einen neuen Worker-Aktor, der dessen Aufgaben weiter abarbeitet.

Das Anwendungsbeispiel soll das aktuelle Kamerabild in einer GUI darstellen und den Roboter durch Buttons bewegen können. Dazu wird die eben erwähnte Initialisierung im Package-Object der Anwendung durchgeführt.

[1] vgl. [Akka2014]

```
1.  import robots.nao._
2.  package object robotdemo {
3.    lazy val ACTOR_SYSTEM = initActorSystem("robotdemo", "nao")
4.    val ROBOT_IP = "127.0.0.1"
5.
6.    private def initActorSystem(systemName:String, actorPath:String):ActorSystem = {
7.      val sys = ActorSystem(systemName)
8.      Configuration.nao = NaoConfiguration(ROBOT_IP)
9.      sys.actorOf(Props[RobotActor], actorPath)
10.     sys
11.   }
12. }
```

Codeausschnitt: Aktorensystem für die Nutzung der Roboter-Schnittstelle einrichten

Im Codeausschnitt wird das Aktorensystem durch die Anweisung **lazy val** bei Bedarf erstellt. Dies führt dazu, dass das Aktorensystem und der gestartete Front-End-Aktor an den Thread gebunden werden, welcher das System als erstes benutzt. Wird dieser Thread beendet, dann beendet sich auch das Aktorensystem und der Aktor für den Roboter ist nicht mehr ansprechbar. Bei dieser Art von Initialisierung muss sichergestellt werden, dass der initialisierende Thread nicht beendet wird, solange die Aktoren noch benötigt werden.
Das hier besprochene Anwendungsbeispiel ist eine einfache Swing-Anwendung, deren Threads nur beendet werden, wenn die GUI geschlossen wird. Somit spielt es keine Rolle, welcher Thread die Initialisierung des Aktorensystems durchführt.

Zur Darstellung des Kamerabilds wird eine abgeänderte Form des *ImagePanels*[1] genutzt.

```
1.  class ImagePanel extends Panel {
2.    var bufferedImage:BufferedImage = null
3.
4.    override def paintComponent(g:Graphics2D) = {
5.      val image = bufferedImage
6.      if (image != null) g.drawImage(image, null, 0, 0)
7.    }
8.  }
```

Codeausschnitt: GUI-Element zur Darstellung der übertragenen Bilder

Im Gegensatz zum originalen *ImagePanel* wird kein Pfad zur Bilddatei angegeben, sondern der Bildpuffer des im Speicher liegenden Bildes. Über den Setter *bufferedImage* werden die Bilder in der Anwendung periodisch durch neue ersetzt. Das periodische Abfragen der Bilddaten wird in einen eigenen Thread ausgelagert. Im Folgenden ist die *run*-Methode des Threads abgebildet.

[1] vgl. [SO2011]

```
override def run {
  implicit val actorSystem = ACTOR_SYSTEM
  val nao = actorSystem.actorFor("/user/nao");
  if (robotConnected(nao)) {
    nao ! RobotRequest("VideoDevice", "setResolution", Resolution.VGA)
    nao ! RobotRequest("VideoDevice", "setFrameRate", 5)
    actor("videoPanel")(new Act{
      import context._
      system.scheduler.schedule(200 millis, 200 millis, self, Timeout)
      become {
        case Success(bytes:Array[Byte]) => updateImage(bytes)
        case Timeout => nao ! RobotRequest("VideoDevice", "getImageLocal")
      }
    })
  }
}
```

Codeausschnitt: Periodisches Abfragen von Bilddaten

Mit der ersten Zuweisung wird das Aktorensystem zusammen mit dem Front-End-Aktor gestartet. Anschließend wird mit der Hilfsmethode *robotConnected* geprüft, ob der Roboter erreichbar ist. Dazu wird in dieser Methode ein kurzer Aufruf an den Aktor gesendet und geprüft, ob innerhalb eines vorgegebenen Timeouts eine Antwort empfangen wurde. Sollte der Roboter erreichbar sein, werden die Kameraauflösung und die Bildrate gesetzt. Anschließend wird mithilfe der ActorDSL von Akka ein neuer Aktor definiert und gestartet. Dieser Aktor wird von einem Scheduler alle 200 Millisekunden geweckt und fragt die aktuellen Bilddaten vom Front-End-Aktor ab. Sobald die Bilddaten eingehen, wird die Hilfsmethode *updateImage* aufgerufen. Diese Methode wandelt das Byte-Array in ein *BufferedImage* um und aktualisiert das *ImagePanel*, welches das Bild darstellen soll.

Bevor der Roboter durch den Raum navigiert werden kann, muss er aufstehen und auf das Gehen vorbereitet werden. Dazu wird ein Button verwendet, der den Roboter aufstehen, oder wieder hinsetzen lässt. Aufstehen und Hinsetzen bestehen im NAOqi-API aus zwei Befehlen, die sequentiell ausgeführt werden müssen. Grundsätzlich bietet die in dieser Arbeit entworfene Schnittstelle nur parallele Anfragen an. Weil aber eine sequentielle Ausführung von Anfragen in manchen Anwendungsfällen notwendig ist, gibt es die Hilfsklasse *RequestSequence*. Diese Hilfsklasse ist unabhängig vom genutzten Robotertyp und deshalb für alle Roboter verfügbar. *RequestSequence* kann implizit aus einer Liste von Anfragen erstellt werden. Mit der Methode *sendTo* werden dann alle Anfragen sequentiell zu dem angegebenen Zielaktor gesendet. Sollte eine Anfrage fehlschlagen, dann wird die Sequenz abgebrochen und der Fehler zurückgegeben. Schlägt keine Anfrage fehl, dann gibt die Methode den letzten Rückgabewert zurück.

```
1.  action = new Action("toggleStand") {
2.    def apply = {
3.      enabled = false
4.      requestList.sendTo(system.actorFor("/user/nao")) onComplete {
5.        case Success(_) => {
6.          standing = !standing
7.          text = updatedText
8.          enabled = true
9.        }
10.       case Failure(error) => enabled = true; println("An error occurred: " + error)
11.     }
12.   }
13. }
```

Codeausschnitt: Button-Action zum Aufstehen und Hinsetzen

In diesem Anwendungsbeispiel wird der Button zum Aufstehen und Hinsetzten von einem Scala-Swing-Button abgeleitet. In *action* wird das Verhalten des Buttons definiert. In diesem Fall wird der Button beim Draufklicken deaktiviert und aktiviert, sobald die Anfrage abgeschlossen ist. Die Anfrage besteht aus einer Liste von Anfragen.

```
1.  def requestList = {
2.    if (!standing)
3.      List(RobotRequest("RobotPosture", "goToPosture", "Stand", 1.0),
4.           RobotRequest("Motion", "moveInit"))
5.    else
6.      List(RobotRequest("Motion", "stopMove"),
7.           RobotRequest("RobotPosture", "goToPosture", "Sit", 1.0))
8.  }
```

Codeausschnitt: Anfragen zum Aufstehen und Hinsetzen

Abhängig davon, ob der Roboter bereits aufgestanden ist oder nicht, müssen andere Anfragen ausgeführt werden. Wenn die Anfragensequenz erfolgreich ausgeführt wurde, dann wird der Zustand des Buttons geändert, der Text aktualisiert und der Button wieder aktiviert. Sollten die Anfragen fehlschlagen, dann wird der Button wieder aktiviert und ändert seinen Zustand nicht. Analog dazu werden die Buttons für das Gehen und den Richtungswechsel implementiert. Der vollständige und lauffähige Code für dieses Anwendungsbeispiel befindet sich im Anhang.

4 Fazit und Ausblick

Die Implementierung eines erweiterbaren APIs für alle Robotertypen ist realisierbar und ermöglicht die Wiederverwendung von Anwendungen für Roboter. Die Schwierigkeit liegt darin, dass jeder Robotertyp einzeln an dieses API angebunden werden muss und sich einige Schnittstellen unter Umständen nicht auf das gewählte NAOqi-API abbilden lassen. Dies wurde in dieser Arbeit nicht weiter untersucht und als gegeben vorausgesetzt.

Eine einheitliche Roboterschnittstelle könnte in Zukunft gerade im Smart-Home-Bereich unerlässlich werden. Wenn das Smart-Home beliebige Roboter zur Erledigung von Aufgaben nutzen soll, müssen alle Roboter ein einheitliches API haben. Durch das einheitliche API für alle Robotertypen lässt sich jedoch nicht vermeiden, dass Software für jeden neuen Robotertypen angepasst werden muss. Die Anwendungen selbst müssen nicht angepasst werden. Das System, welches die die einzelnen Roboter in das Aktorensystem einbindet, muss jedoch für jeden neuen Robotertypen angepasst werden. Schließlich muss jeder Roboter anders in die Schnittstelle eingebunden werden. Die einzige Möglichkeit dies zu vermeiden wäre, die Anbindung des Roboters zu standardisieren, sodass alle Roboter über die gleiche Schnittstelle ansprechbar sind, wie das RPC-Modul im NAO. Dies wäre jedoch eine gewaltige Einschränkung für die Funktionalität der Roboter, denn bereits für die performante Übertragung von Bilddaten im NAO war eine speziell angepasste Schnittstelle notwendig.

Die Probleme mit dem Videomodul haben auch die technischen Grenzen von Roboteranwendungen gezeigt. Um die Videos mehrerer Roboter in Echtzeit über ein Funknetzwerk zu übertragen, reicht die Bandbreite für gewöhnlich nicht aus. Wenn mehrere Roboter zur Unterstützung des Menschen von einem Smart-Home koordiniert werden sollen, muss die Umsetzung diese technischen Einschränkungen berücksichtigen.

In dieser Arbeit wurde vereinfachend das NAOqi-API als Referenz-API verwendet. Es wurden keine weitern Roboter-APIs evaluiert und nicht versucht, ein eigenes API zu entwerfen. Diese Vereinfachung wurde vorgenommen, um den NAO mit möglichst wenig Aufwand in das API integrieren zu können und im zeitlichen Rahmen einer Bachelor-Arbeit eine funktionsfähige Implementierung abzuliefern. Das NAOqi-API ist keine optimale Schnittstelle für Roboteranwendungen. Zum Beispiel unterstützt es kein Videostreaming und jedes Bild muss einzeln abgefragt werden. Dennoch bietet das NAOqi-API eine Menge an Modulen und Methoden, die für viele Aufgaben ausreichen. Durch das leicht

erweiterbare Nachrichteninterface von Akka lässt sich das API schnell um neue Methoden erweitern.
Das NAOqi-API ist zudem für das präzise Ansteuern der einzelnen Elemente des Roboters entworfen worden und abstraktere Befehle sind manchmal nur durch Verkettung mehrerer einzelner Befehle realisierbar.

```
1. val joints = List("HeadYaw")
2. val angles = List(Math.toRadians(90))
3. val speed = 0.5
4.
5. for {
6.   x <- nao ? RobotRequest("Motion", "setStiffnesses", joints, List(1.0))
7.   y <- nao ? RobotRequest("Motion", "setAngles", joints, angles, speed)
8. } nao ! RobotRequest("Motion", "setStiffnesses", joints, List(0.0))
```

Codebeispiel: Kopf des Roboters um 90° drehen

Das eben gezeigte Codebeispiel zeigt, dass für eine einfache Kopfdrehung des Roboters bereits drei einzelne Befehle notwendig sind. Diese Befehle müssen zudem sequentiell ausgeführt werden, was der for-Ausdruck sicherstellt.
In der Form ist das API zu kompliziert in der Verwendung, um von Anwendungsentwicklern akzeptiert zu werden. Deshalb muss im nächsten Schritt das API um Aufrufe wie „turnHead“ erweitert werden. Diese Aufrufe würden Befehlssequenzen, wie die obige zu einem Befehl zusammenfassen und der Anwendungsentwickler würde somit auf einer höheren Abstraktionsschicht entwickeln. Aufgrund der losen Kopplung der Aktoren-Schnittstelle, kann das Modul, welches diese abstrakten Methoden zur Verfügung stellt, für alle Roboter ohne Anpassung verwendet werden. Zudem kann es jederzeit um neue Funktionalität erweitert werden.

Kritisch ist zudem die Verwendung von ZeroMQ zur Kommunikation zwischen dem NAO und dem Aktorensystem. ZeroMQ ist zwar sehr performant und für diesen Aspekt die richtige Wahl, es bietet durch seine hohe Abstraktion dem Anwender jedoch keine Möglichkeit, auf Verbindungsabbrüche zu reagieren. Einerseits ist das API dadurch für den Benutzer einfacher zu verwenden, weil er davon ausgehen kann, dass eine Anfrage trotz Verbindungsabbruch irgendwann übermittelt wird. Andererseits ist es für den Benutzer so aufwendiger, die Anwendung auf Verbindungsabbrüche reagieren zu lassen. Die Anwendung kann die Erreichbarkeit des Roboters nur durch Timeouts oder durch regelmäßiges Senden von kurzen Anfragen(Heartbeat) prüfen.
Eine mögliche Verbesserung wäre es, diesen Heartbeat in das API zu integrieren, damit nicht jede Anwendung ihren eigenen Heartbeat implementieren muss.
Die Verwendung von ZeroMQ bringt noch ein weiteres Risiko mit sich. Das Request-Reply-Pattern in ZeroMQ kann unter bestimmten Umständen dazu führen, dass der Socket nicht mehr ansprechbar ist. Auf die Implementierung des APIs bezogen hat dies zur Folge, dass ein Worker-Aktor des RPC-Aktors durchgehend auf eine Antwort vom NAO wartet, diese aber nie erhält. Das Problem wird im folgenden Diagramm kurz verdeutlicht.

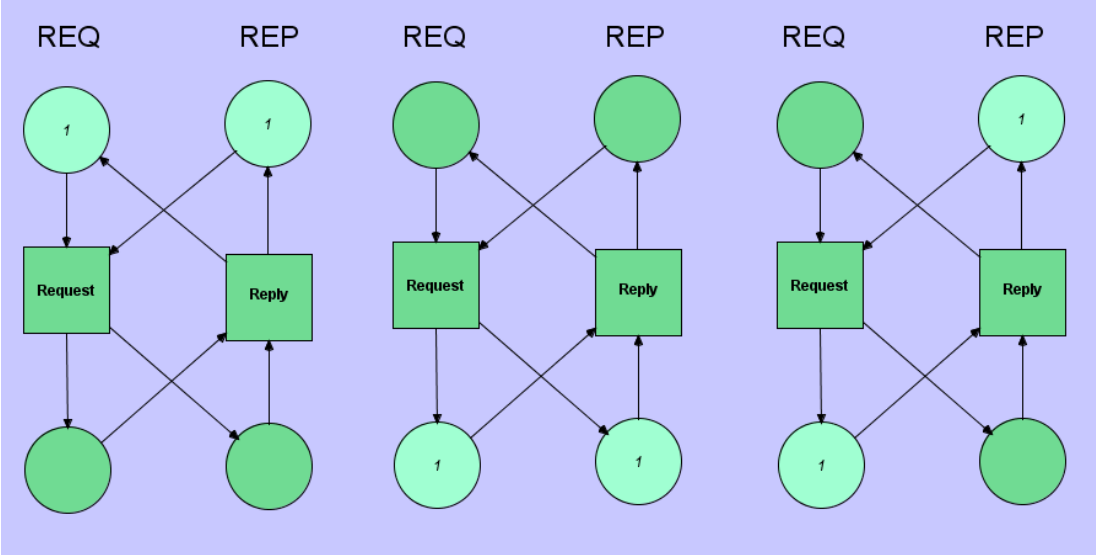

Abb. 14 Stürzt nach dem 2. Schritt der REP-Socket ab, befindet sich die Verbindung in einem blockierten Zustand

Die Request-Reply Sockets besitzen intern zwei Zustände. In dem einem kann nur empfangen und in dem anderen kann nur gesendet werden. Der Hauptunterschied zwischen Request-Socket und Reply-Socket besteht darin, in welchem Zustand sie sich zu Anfang befinden. Kommt es nun durch einen Verbindungsabbruch zu einer Konstellation, in welcher sich beide im Empfangszustand befinden, kann die Kommunikation erst fortgesetzt werden, wenn der Request-Socket geschlossen und erneut geöffnet wird. Das Problem an dieser Situation ist, dass der Request-Socket keine Möglichkeit hat, zwischen einem lang laufenden Aufruf und einer solchen Blockade zu unterscheiden.[1]

Zudem liefert das Info-Modul in der jetzigen Umsetzung Informationen, die nicht ausreichend genau sind. In der Umsetzung in dieser Arbeit gibt das Info-Modul nur an, welche Module implementiert werden. Dies ist nicht genau genug, da einzelne Module sehr umfangreich sind und sich in einigen Robotern möglicherweise nur teilweise implementieren lassen. Dann muss der Info-Aktor zu jedem Modul zusätzlich die implementierten Methoden liefern können.

Trotz der vielen vereinfachenden Annahmen konnte am Beispiel des NAOs in dieser Arbeit gezeigt werden, dass ein erweiterbares und generisches API für humanoide Roboter realisierbar ist. Für die Praxistauglichkeit müssen die vorgenommen Vereinfachungen durch robuste und performante Lösungen ersetzt werden.

[1] vgl. [Ron2012]

5 Anhang

5.1 Literaturverzeichnis

[Akka2014] Akka Documentation: *Supervision and Monitoring*
Online unter:
http://doc.akka.io/docs/akka/snapshot/general/supervision.html
Abruf: 01.04.2014

[Alde2013a] *NAO Datenblatt.*
Online unter: http://www.aldebaran-robotics.com/en/Discover-NAO/datasheet.html
Abruf: 24.10.2013

[Alde2013b] *OpenNAO.*
Online unter: http://www.aldebaran-robotics.com/documentation/dev/tools/opennao.html
Abruf: 24.10.2013

[Alde2013c] *NAOqi modules APIs.*
Online unter: https://community.aldebaran-robotics.com/doc/1-14/naoqi/index.html
Abruf: 14.02.2014

[Alde2014d] *ALVideoDevice – NAO Software Documentation.*
Online unter: https://community.aldebaran-robotics.com/doc/1-14/naoqi/vision/alvideodevice.html#performances-and-limitations
Abruf: 21.02.2014

[Ayedo2013] *What is Netty?*
Online unter: http://ayedo.github.io/netty/2013/06/19/what-is-netty.html
Abruf: 01.11.2013

[Cooper2010] Peter Cooper (2010). *MessagePack: Efficient, Cross Language Binary Object Serialization.*
Online unter: http://www.rubyinside.com/messagepack-binary-object-serialization-3150.html
Abruf: 28.10.2013

[MsgB2013] *FAQ MessagePack.*
Online unter: http://wiki.msgpack.org/ display/MSGPACK/FAQ
Abruf: 28.10.2013

[MuGra2009] G. Mulligan, D. Gračanin (2009). *A comparison of SOAP and REST implementations of a service based interaction independence middleware framework.* Aus *WCS '09 Winter Simulation Conference* Seiten 1423-1432. ISBN: 978-1-4244-5771-7

[OOPSLA2013] H. Miller, P. Haller, E. Burmako, M. Odersky (2013). *Instant Pickles: Generating Object-Oriented Pickler Combinators for Fast and Extensible Serialization.* Aus OOPSLA'13, Indianapolis

[Pickl2013] *Scala Pickling.*
Online unter: http://lampwww.epfl.ch/~hmiller/pickling/
Abruf: 28.10.2013

[Proto2013] *Language Guide Protocol Buffer.*
Online unter: https://developers.google.com/protocol-buffers/docs/proto
Abruf: 28.10.2013

[Ron2012] Armin Ronacher(2012). *ZeroMQ: Disconnects are good for you*
Online unter: https://lucumr.pocoo.org/2012/6/26/disconnects-are-good-for-you/
Abruf: 31.03.2014

[SO2011] Stack Overflow: *Scala Swing image*
Online unter: http://stackoverflow.com/questions/5752330/scala-swing-image#5752509
Abruf: 07.04.2014

[Wyatt2009] Derek Wyatt (2009). *Balancing Workload Across Nodes with Akka 2*
Online unter: http://letitcrash.com/post/29044669086/balancing-workload-across-nodes-with-akka-2
Abruf: 11.03.2014

[Zero2013] *ZeroMQ Guide.*
Online unter: http://zguide.zeromq.org/page:all
Abruf: 25.10.2013

[Zero2013b] *Multithreaded service in C++.*
Online unter: http://zguide.zeromq.org/cpp:mtserver
Abruf: 14.02.2014

5.2 Abbildungsverzeichnis

5.3 Tabellenverzeichnis

5.4 Quellenverzeichnis

Abb. 1 - http://www.aldebaran-robotics.com/images/site/nao_schema_zoom.jpg
Abb. 5 - https://raw.githubusercontent.com/imatix/zguide/master/images/fig16.png
Tbl. 1 - http://zguide.zeromq.org/page:all#Messaging-Patterns

Versicherung über Selbstständigkeit

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

Hamburg, den _____ _____