4 Realisierung einer distributiven Steuerung

Im Rahmen des Projektes Humanoider Roboter an der HAW Hamburg im Sommersemester 2012 und dem Wintersemester 2013 wurde eine Steuerung eines humanoiden Roboters am Beispiel des Naos entwickelt. Im Fokus stand dabei eine Architektur, die insbesondere Fehlertoleranz und Sprachenunabhängigkeit in einem distributiven System mit sich bringt. Das Projekt ist in seiner Startphase und bringt noch viel Potential mit sich. Abgebildet werden daher grundlegende Designentscheidungen, die eine greifbare Idee vermitteln sollen, welche Möglichkeiten diese Architektur bietet. Die Funktionsweise der Architektur wird dabei von exemplarischen Apps demonstriert.

4.1 Verteilung

Wie im übiquitären Umgebung üblich, gesellen sich eine Vielzahl von Geräten zueinander. Mehrere Roboter, ein Handy, Webservices wie Newsreader oder Mailbox und Peripherie wie ein Controller für Spielekonsolen werden dazu zur Hand genommen. In der Mitte steht das Aktorensystem Robot Middleware (Abb. 4.1 Verteilungssicht) welches High Level Funktionen unabhängig von der Hardware für Apps zur Verfügung stehen. Speziell auf den Nao zugeschnitten ist der Adapter naogateway, der die Kommunikation mit dem Nao realisiert.

Ist eine zentrale Instanz vorhanden, die ein Roboter ansteuert, muss keine Kommunikationsmodell vorhanden sein um verschiedenen Anfragen von verschiedenen Stellen verarbeiten zu können und die passenden Antworten an den korrekten Absender zu leiten. Eine zentrale Instanz dagegen ist dagegen ein Single Point of Failure und mindert so die Verteilbarkeit. Wünschenswert wäre an dieser Stelle ein Aktor auf dem Nao, der diese Funktionalität mit sich bringt. Hardwarenahe Aktoren wie libcppa sind vorhanden und sind das Ziel für die Zukunft.

Ohne Aktor auf dem Nao muss sichergestellt sein, dass einem Nao genau ein naogateway zugeordnet ist. Umgekehrt gibt es keine Einschränkungen.

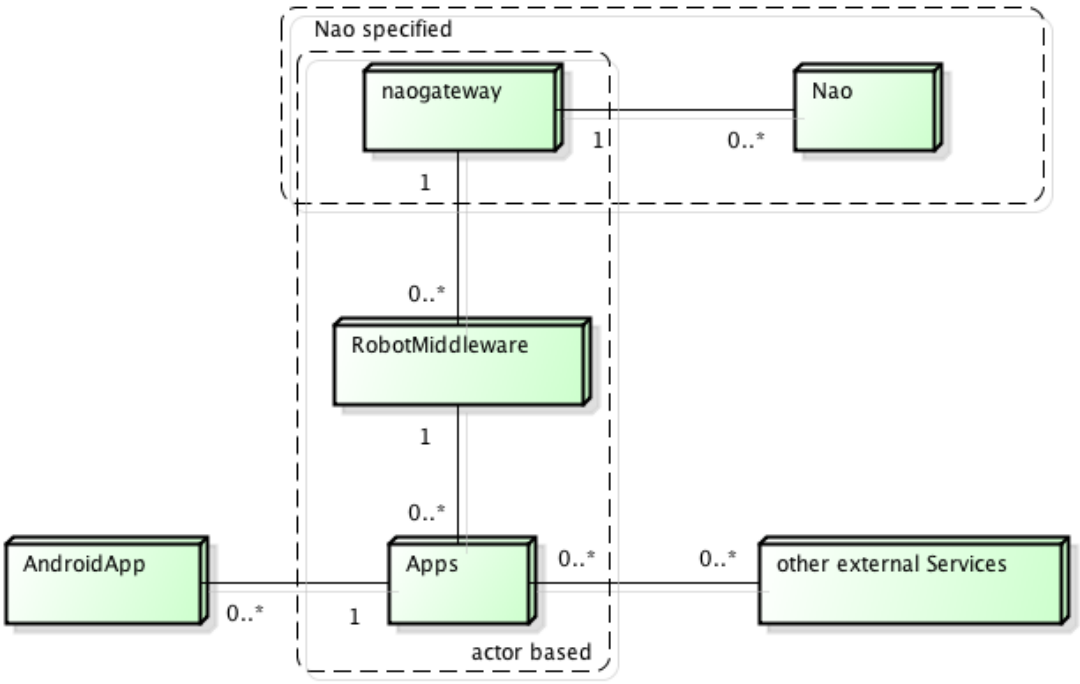

Abbildung 4.1: Verteilungssicht

Im gleichen Zug benötigt die RobotMiddleware, die die hochsprachlichen Funktionen wie beispielsweise Laufen beherbergt, genau eine Instanz zu naogateway um einheitliche Entscheidungen treffen zu können wie den Roboter bei einem Fehler (z.B. leerer Akku) in einen sicheren Zustand (hinsetzen) zu bringen. Apps kommunizieren mit einer RobotMiddleware, sind ansonsten frei von Restriktionen. Architektonisch zu sehen sind die Positionen der externen Dienste. Apps sind unabhängig von einander agierende Softwarepakete, die die Anwendungsfälle abbilden und dabei die Resource RobotMiddleware benutzen. Jedoch können diese auch den Zugang zu externen Diensten erlangen ohne dabei RobotMiddleware oder naogateway zu berühren.

4.2 Kommunikationsmechanismen

Die Kommunikation ist ein Eckpfeiler der Architektur und wird klarer, wenn ein etwas tieferer Blick in die einzelnen Komponenten geworfen wird (Abb. 4.2 Kommunikationsschema). Im Grundsatz gibt es vier Bereiche, die miteinander kommunizieren müssen. Das Mittel der Wahl ist dabei Message Passing um ein nebenläufiges, autonomes und distributives System realisieren zu können.

Der HAWActor ist ein von David Olszowka entwickeltes Naoqi Modul, welches der Naoqi API nachempfunden, entfernte Methodenaufrufe realisiert. Dies wird durchgeführt indem das Modul eine serialisierte Request Nachricht erhält, die aus Modulname, Methodennamen und Parameter besteht und, diese in einen Methodenaufruf umwandelt (Unmarshaling) und den Reply (der auch leer sein kann) serialisiert (Marshalling) und zurücksendet. Die binäre Kommunikation wird über das TCP Socket basierte ZMQ umgesetzt. Für Naoqi wurden dabei spezielle Datenstrukturen durch Protobuf definiert.

Das Aktorensystem naogateway ist speziell für den Nao geschrieben und verbindet sich über ZMQ Sockets mit dem Nao, dabei gibt es die Möglichkeit die Naoqi APi zu benutzen, allerdings auch eine spezielle API für den Zugriff auf komprimierte Kameradaten. In Aussicht steht eine API für Audio und die Naoqi eigene Datenbank ALMemory (ein Key Value Store).

Die RobotMiddleware bietet ein trait RobotInterface, welches hochsprachliche Methoden wie Laufen bereitstellt und beim Aufruf die Methoden in Nachrichten umsetzt, die an naogateway delegiert werden, dort in Naoqi spezifische Aufrufe umgewandelt werden und zu Nao geschickt werden. Die Antwort wird über naogateway wieder zum NaoTranslator zurückgeleitet.

Apps, die in einem eigenen Aktorensystem agieren um völlig unabhängig zu sein, haben die Möglichkeit auf die NaoInterface API zuzugreifen und gleichzeitig externe Dienste zu nutzen um eine vollwertige App zu schaffen, die ein Nao steuern kann. Die Idee ist Nachrichten vollständig über Protobuf abzubilden und dabei von speziellen Roboterschnittstellen wie Naoqi zu abstrahieren. Anzumerken ist, dass durch das frühe Entwicklungsstadium Teile noch nicht umgesetzt sind. In der Abbildung 4.2 Kommunikationsschema sind die noch nicht umgesetzten Teile rot markiert.

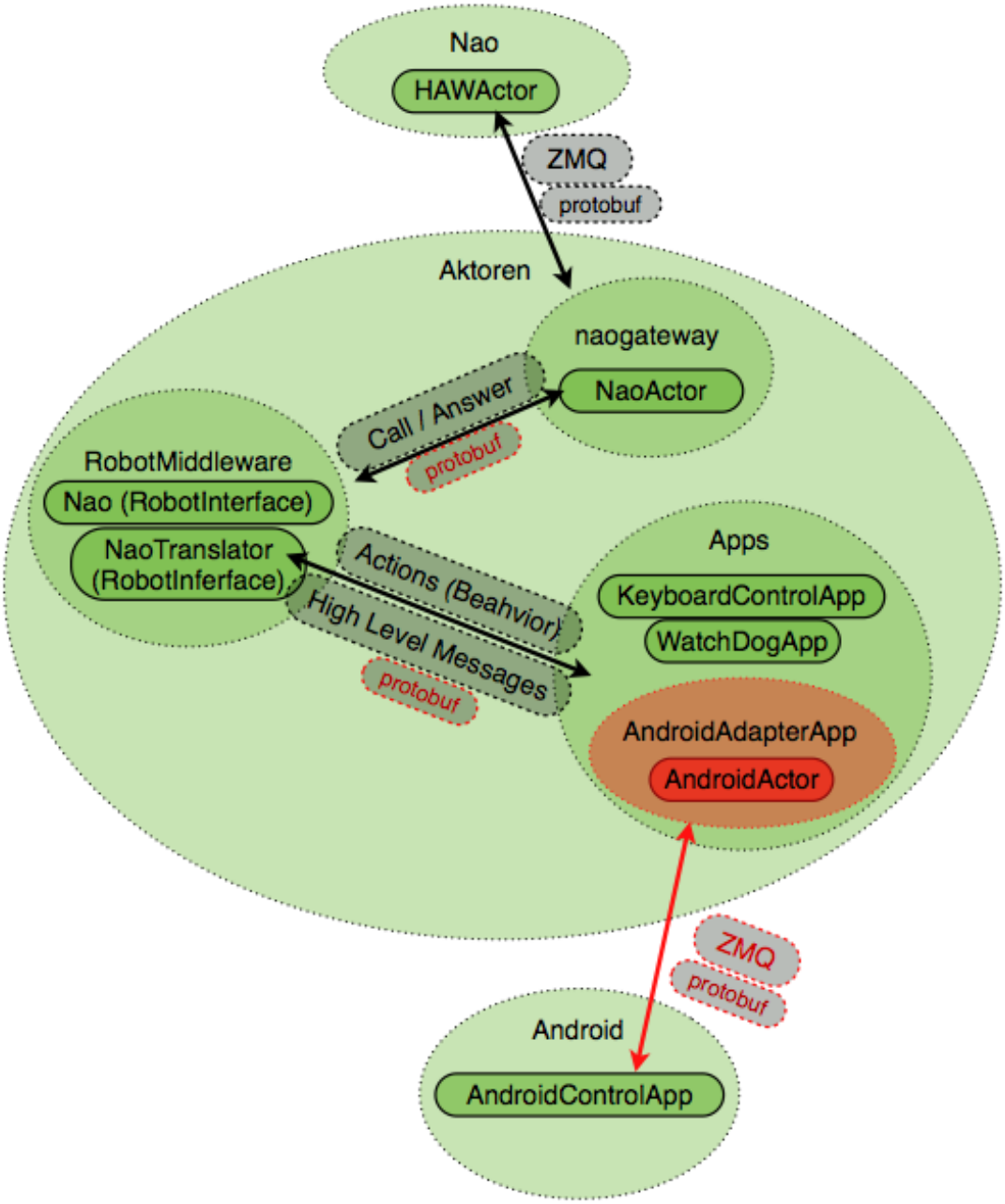

Abbildung 4.2: Kommunikationsschema

Das Aktorensystem naogateway (Abb. 4.3 Aktorensystem naogateway) ist hat verschiedene Dienste die es abbildet. Jeder Dienst ist ein Aktor. Ein standardmäßiger Methodenaufruf in der Naoqi API benötigt einen Aufruf und gibt eine Antwort zurück (die auch im Sinne von

void leer sein kann). Dies wird durch einen ResponseActor umgesetzt. Der NoResponseActor verwirft jegliche Nachricht, sodass der Entwickler beide Varianten zur Wahl hat, je nachdem ob die Antwort interessiert oder nicht. Die Aktoren kümmern sich dabei nicht darum, ob die Funktion einen verwertbaren Rückgabewert hat oder nicht. Der VisionActor ermöglicht die Abfrage von komprimierten Bildern von der Kamera des Nao. Kompression unterstützt Naoqi nicht nativ. Audio und Memory sind derzeit in der Entstehung. Erschafft und überwacht werden diese Aktoren vom NaoActor, der auch als Schnittstelle nach außen benutzt wird. Damit es möglich ist zu wissen ob der Nao zur Zeit erreichbar ist, existiert der HeartBeatActor. In einem fest definierten Intervall werden vom HeartBeatActor Testnachrichten geschickt, die den einzigen Zweck haben eine Antwort zu generieren. Ist eine Antwort gekommen, ist sichergestellt, dass der Nao in diesem Moment erreichbar ist.

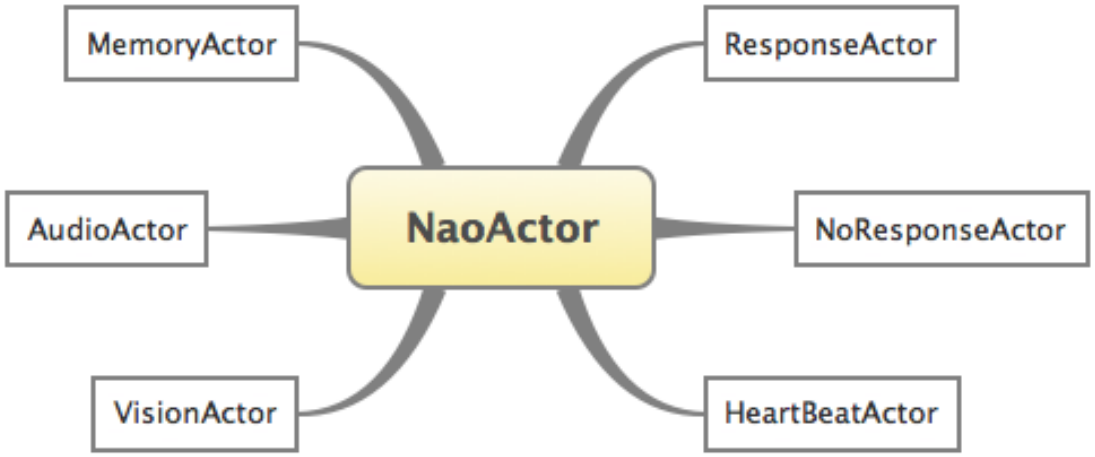

Abbildung 4.3: Aktorensystem naogateway

Für der entfernten Methodenaufruf ist eine Protobuf Nachricht definiert mit genau einem Modulnamen, einem Methodennamen und beliebig vielen Parametern. Dieser HAWActorRPCRequest wird von naogateway an den Nao geschickt. Dieser wird vom ResponseActor und vom NoResponseActor verwendet.

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

Die Parameter sind als MixedValue definiert, der als überladender Typ die Zeichenketten, Integer, Float, Byte, Boolean und Array (beliebige Wiederholung des eigenen Typs) unterstützt. Überladene Typen sind in hardwarenahen Sprachen sehr effektiv, leider aus Sicht der Typsicherheit nicht gut zu handhaben, da durch Abfrage, ob der Typ definiert ist, die Bytes in den entsprechenden Typ überführt werden müssen. Da im Request eine beliebige Anzahl von MixedValue enthalten sein können und MixedValue sich beliebig oft selbst enthalten kann, können verschachtelte Listen von Parametern definiert werden. Egal in welcher Tiefe die Parameterliste vorliegt, es können immer Typen gemischt werden.

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

Die Antwort ist genau ein Rückgabewert MixedValue. Es ist somit keine Verschachtelung möglich. Es ist möglich einen Fehler in Form eines Strings anzugeben. Sollte es eine Methode nicht geben, wird von einer Fehlernachricht gebrauch gemacht. Der Rückgabewert ist dann leer. Dieses Verhalten nutzt der HeartBeat für seine Testnachrichten, da so sehr wenig Last entsteht und das Protokoll nicht verändert werden muss. Die Nachrichten des VisionActors ist vergleichbar aufgebaut.

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

4.3 Kommunikationssequenzen

Das Aktorensystem naogateway übernimmt die entscheidende Kommunikation zum Nao, die in diesem Abschnitt dargestellt werden soll. Initialisiert wird wird der NaoActor (Abb. 4.4 Initialisierung) beim Start des Aktorensystems, definiert durch die Konfigurationsdatei. Damit ist sichergestellt, dass es genau einen NaoActor gibt. In der Konfiguration sind

auch die Zugangsdaten hinterlegt, sodass der HeartBeat direkt genutzt werden kann und zum Prüfen ob der Nao derzeit erreichbar ist. Im Positivfall werden die Kinder gestartet und denen die Zugangsdaten übermittelt. Da ein ZMQContext von verschiedenen Aktoren gleichzeitig angesprochen werden kann, benötigt jedes Kind seinen eigenen. Ist der

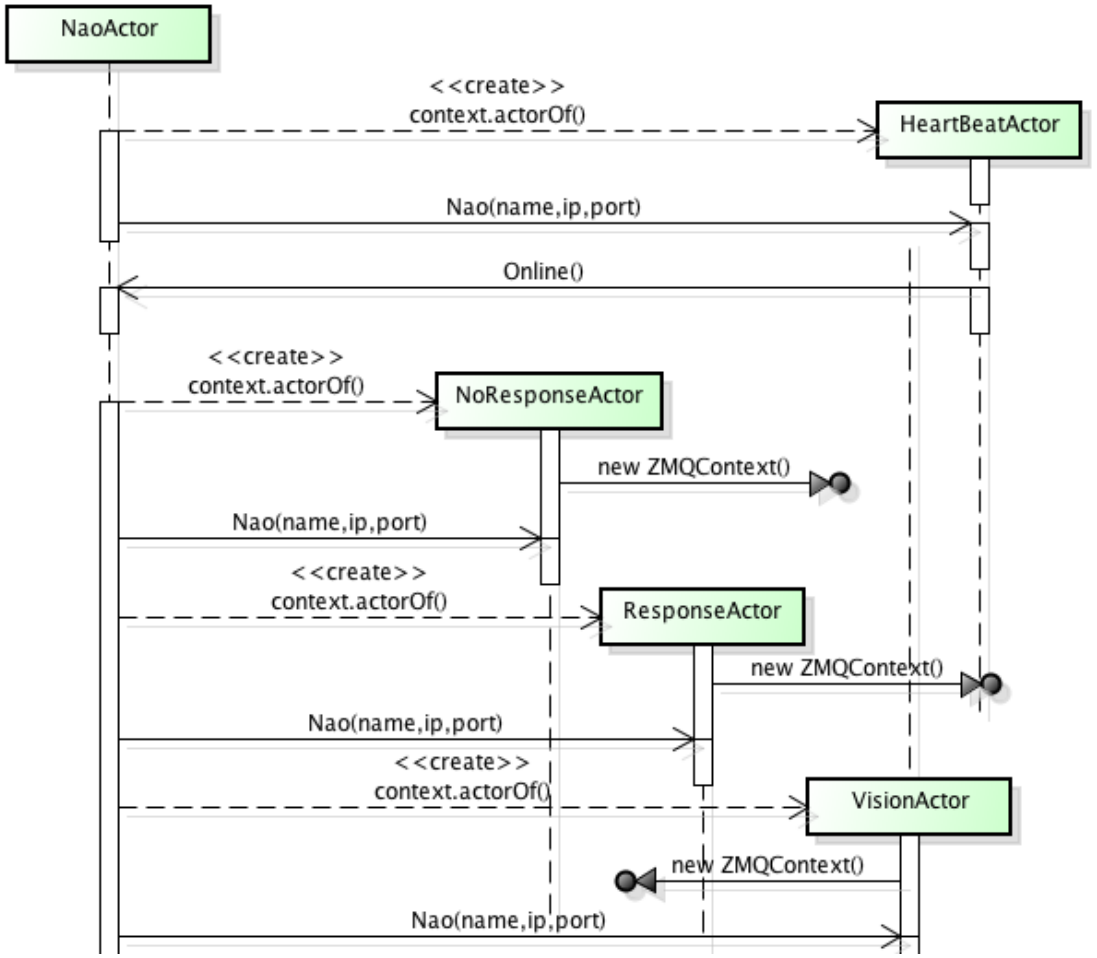

Abbildung 4.4: Initialisierung

NaoActor mit seinen Kindern hochgefahren, kann durch einen Connect Nachricht in einem Tripel $(reponseActorRef, noResponseActorRef, visionActorRef)$ die Aktorreferenzen abgefragt werden um an einen ausgewählten Aktor einen Request zu schicken. Da an einer ActorRef zunächst einmal nicht erkennbar ist, welcher Aktor dahinter steckt, ist hierbei zu überlegen auf TypedActors auszuweichen. Nicht-blockierend zu sein ist eine wichtige Anforderung für einen Aktor um in der Lage zu sein auf neue Nachrichten zu reagieren, unabhängig davon welche Nachrichten davor kamen (Abb. 4.5 Reuquest-reply Kommunikation). Um dabei

nicht zu viele Aktoren und ohne zu viele ZMQSockets gleichzeitig zu erstellen wird nach dem Senden ein Future erzeugt. Das Future erhält als Funktion das Warten auf die Antwort, die Umwandlung in die Protobufdatenstruktur und den Absender um im Erfolgsfall die Antwort zurückzuschicken. Der ZMQ Socket wird danach geschlossen. So wird das Request-reply Protokoll korrekt eingehalten ohne zu blockieren. Das Senden einer Nachricht ohne eine Antwort

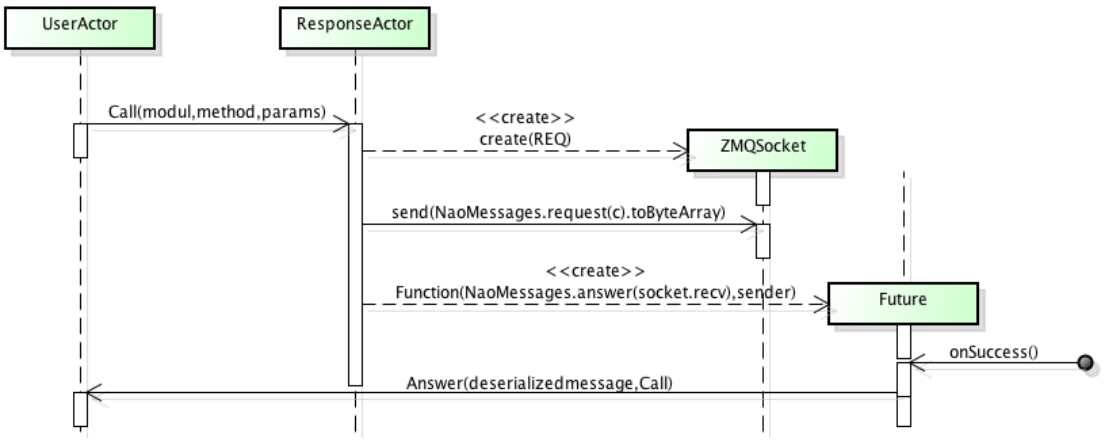

Abbildung 4.5: Reuquest-reply Kommunikation

zu erhalten erfolgt über den NoResponseActor (Abb. 4.6 Vorgetäuschte unidirektionale Kommunikation). Der NoRespionseActor muss auch das Request-reply Protokoll korrekt einhalten ohne zu blockieren. Dieser funktioniert ähnlich zum ResponseActor mit einer Besonderheit. Der Aktor verfügt über eine Mailbox, aus der der Aktor nicht nur Nachricht herausnehmen kann, sondern auch nach dem herausnehmen kurz zur Seite gestellt werden können (Stashing). Die Nachrichten können nicht direkt in die Warteschlange gelegt werden, da sonst eine Endlosschleife für Stashing entstehen würde.

Außerdem besitzt der NoResponseActor zwei Zustände. Im Zustand communicating empfängt dieser einen Call und erstellt einen Future für die Antwort und geht in den Zustand waiting über, indem jeder Call wieder zurückgelegt wird in die Mailbox. Wenn die Antwort gekommen ist, werden alle Nachrichten, die zur Seite gelegt wurden, wieder in die normale Nachrichtenschlange geschoben. Die Antwort wird nicht weitergeleitet.

ZMQ bietet die Möglichkeit anstelle eines Request Sockets einen Dealer Socket einzusetzen, der mit einem Reply Socket kommuniziert. Dies hat die Besonderheit, dass das Request-reply Protokoll nicht benutzt werden muss, sondern beliebig viele Nachrichten gesendet werden können. Dann würde sich der NoResponse auf einen zustandslosen Aktor entwickeln, der ohne

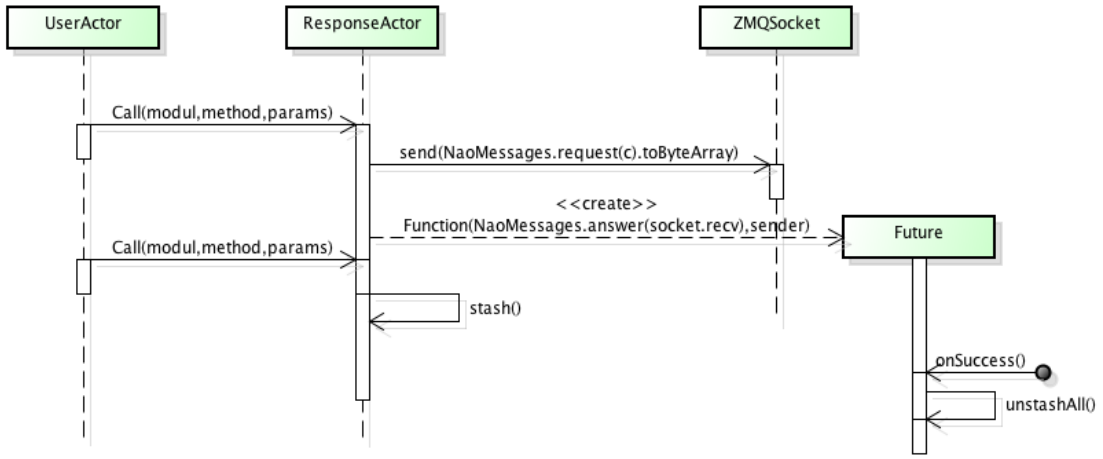

Abbildung 4.6: Vorgetäuschte unidirektionale Kommunikation

weiteres beliebig viele Nachrichen schicken kann. Die eingesetzt ZMQ Version unterstützt dies leider nicht fehlerfrei.

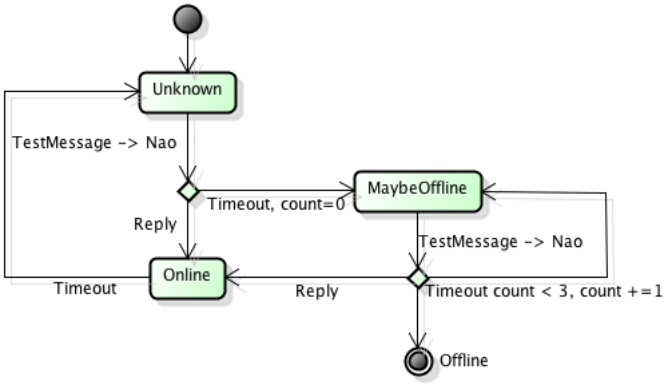

Abbildung 4.7: Heartbeat Zustandsautomat

Der HeartBeatActor verkörpert die Idee auf Basis von Testnachrichten und einem fest definierten Timeout herauszufinden ob der Nao erreichbar ist (Abb. 4.7 Heartbeat Zustandsautomat). Kommt eine Antwort innerhalb des Timeouts, wird der Nao als erreichbar angenommen, wobei nach einem Timeout dieser Zustand wieder überprüft wird. Sollte ein Timeout erfolgen auf Grund einer Testnachricht, wird vermutet, dass der Nao nicht erreichbar ist. Es werden jedoch drei Chancen eingeräumt. Sollte die dritte Chance auch nicht zum Erfolg führen, geht der HeartBeatActor von einem nicht erreichbaren Nao aus und terminiert. Der NaoActor kann nun den HeartBeat neustarten, wenn beispielsweise ein UserActor erneut anfragt.

Die RobotMiddleware ermöglicht hochsprachliche Funktionen, die auf auf Naoqi API implementiert basierend mit dem naogatway implementier sind, jedoch die Naoqi API nach außen vollständig verbergen (Abb. 4.8 Aktoren in der RobotMiddleware). Die Idee ist dabei Funktionalität der Roboters als Aktoren zu modellieren, die untereinander Kommunizieren können. Beispielhaft sind Kopfbewegungen, Armbewegung, Laufen, Sprechen und Aktionen wie Hinsetzen oder Aufstehen realisiert. Entscheidend dabei ist auch, dass der Kommunikationsmechanismus der Naoqi API (Request-reply) nun vollständig gekapselt ist und ein freies Message Passing möglich ist.

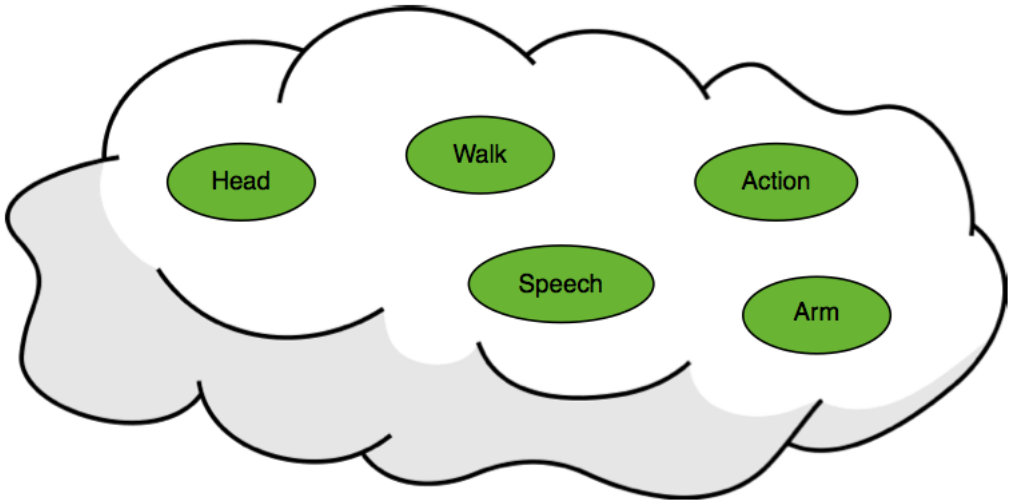

Abbildung 4.8: Aktoren in der RobotMiddleware

Auf Basis der RobotMiddleware wurde eine Steuerung entwickelt, die von verschiedener Peripherie wie Tastatur, Controller einer Spielekonsole realisiert. In Zukunft soll auch die AndroidApp die Dienste der RobotMiddleware verwenden.

5 Fazit und Ausblick

Wenn man in die Zukunft blickt, sind viele Anwendungsszenerien für humanoide Roboter denkbar. Heute sind Smartphones mit vielen individuellen, kleinen Werkzeugen (Apps) ausgestattet. Eine ähnliche Zukunft ist für humanoide Roboter denkbar, die kleine, wichtige Aufgaben erledigen. Beispielhaft ist der Bereich Ambient Assistent Living zu nennen. Mobilität und Informationsaustausch sind die Kernfähigkeiten, sodass der Roboter für Menschen als interaktiven Informationsspender auf einfachste Weise (z.B: Sprachsteuerung) diverse Onlinedienste wie Email, soziale Netzwerke als auch ubiquitäre Elemente wie Mutlimediasysteme oder automatische Fenster nutzbar macht.

Eine moderne Softwarearchitektur ist nötig um Roboter aus den Laboren hinein in das heimische Wohnzimmer zu holen, denn dort steht nicht der Entwickler daneben, der den Roboter auffangen kann, wenn dieser stürzt, dort ist kein Kabel sofort verfügbar, wenn der Akku schwächelt, dort ist keiner da, der nach einem Absturz einen Reboot durchührt. Der Roboter muss stabil sich in seine Umgebung integrieren.

Die Arbeit mit dem Nao gestaltete sich als schwierig, da eine Änderung der Software auf dem Nao an viele Restriktionen mit sich brachte. Die Buildumgebung lässt nur vom Hersteller freigegebene Software zu, eine Nutzung einer Paketverwaltung ist auf dem spezielle angepassten Linux nicht möglich. Wenn außerdem Änderungen an der vom Hersteller gelieferten Software sofort viele Inkompatibilitäten mit sich bringen, zeigt sich, dass auch die Software auf den Robotern an sich auch in die aktuelle Welt integrieren muss.

Der Einsatz einer SOA auf Basis von Aktoren (wie beispielhaft vorgestellt) ist ein wichtiger Schritt, gleichzeitig führt an einem offenen Zugriff auf Daten mit Hilfe einer bekannte IDL vermutlich kein Weg vorbei, denn die Welt ist und bleibt heterogen. Die hochsprachliche Schnittstelle, die Apps ermöglicht mit dem Roboter zu kommunizieren muss dabei flexibel und mächtig sein. Diese muss beim Test verschiedenster humanoider Roboter Bestand haben.

Was eine Architektur alleine nicht leisten kann, ist die Etablierung als Softwarestandard. Zum Softwarestandard entwickelt sich das rein auf Nachrichten spezifizierte ROS. Die hier vorgestellte Architektur ist dagegen mehr richtungsweisend zu sehen und steht noch am Anfang einer Entwicklung.

Blick man in die Umsetzung so muss man feststellen, dass Audio, Video und Memory teilweise bis gar nicht in das System integriert sind. Es stellen sich Fragen, wie verschiedenen Apps gleichzeitig sicher ablaufen, wie sich der Roboter und das Aktorensystem fachlich verhalten, wenn mitten in einer Bewegung des Roboters ein Fehler auftaucht. Welches Wiederaufnahmeszenario muss aktiviert werden? Benötigt man Priorisierungen? Benötigt man Rechte? Insbesondere mit einem nativen Aktor auf dem Nao stellen sich fragen wie verschiedene Aktorensysteme, die mit dem Nao kommunizieren (und dabei die derzeitige 1 zu 1 Beziehung auflösen), sich untereinander absprechen müssen. Wie können Entscheidungen getroffen werden? Sind Regeln für die Kommunikation nötig um einseitige Belastungen zu verhindern. Es stellen sich eine viele Fragen, die in dieser Arbeit bewusst nicht behandelt wurden, da diese den Rahmen sprengen würden, jedoch zu einer vollwertigen Middleware dazugehören.

Die Kombination von einer den zukünftigen Anforderungen gewachsener Architektur und einer etablierten API bleibt eine Herausforderung. Es ist sicher eine grundlegende Untersuchung Wert auf Basis der ROS Spezifikation diese Architektur umzusetzen, denn ROS verkörpert viele ähnliche Konzepte wie Message Passing, Verwendung von einer IDL und die Abstrahierung der Roboter von den Apps. Fehlertolleranz und Sprachenunabhängigkleit bleiben dort derzeit noch ein Nischenthema. Dort kann die hier vorgestellte Architektur anknüpfen.

Literaturverzeichnis

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

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

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

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

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

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

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

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

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

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

Odersky, M. (2011). *Scala Language Specification 2.9*.

Pepper, P. and Hofstedt, P. (2006). *Funktionale Programmierung: Sprachdesign und Programmiertechnik (German Edition)*. Springer. ISBN 978-3-54-0209591.

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

Sarabia, M., Ros, R., and Demiris, Y. (2011). Towards an open-source social middleware for humanoid robots. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 670–675. IEEE.

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

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

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

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

Versicherung der Selbständigkeit

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

Hamburg, 13. Mai 2013 Sigurd Sippel