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 Archiektur 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 und Webservices wie Newsreader oder Mailbox 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 scaleNao, der die Kommunikation mit dem Nao realisiert.

Gibt es eine zentrale Instanz, die ein Gerät ansteuert, muss keine Kommunikationsmodell vorhanden sein um verschiedenen Anfragen von verschiedenen Stellen verarbeiten zu können und die passenden Antworten an die entsprechende Stelle zurückzuschicken. 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 scaleNao zugeordnet ist. Umgekehrt gibt es keine Einschränkungen.

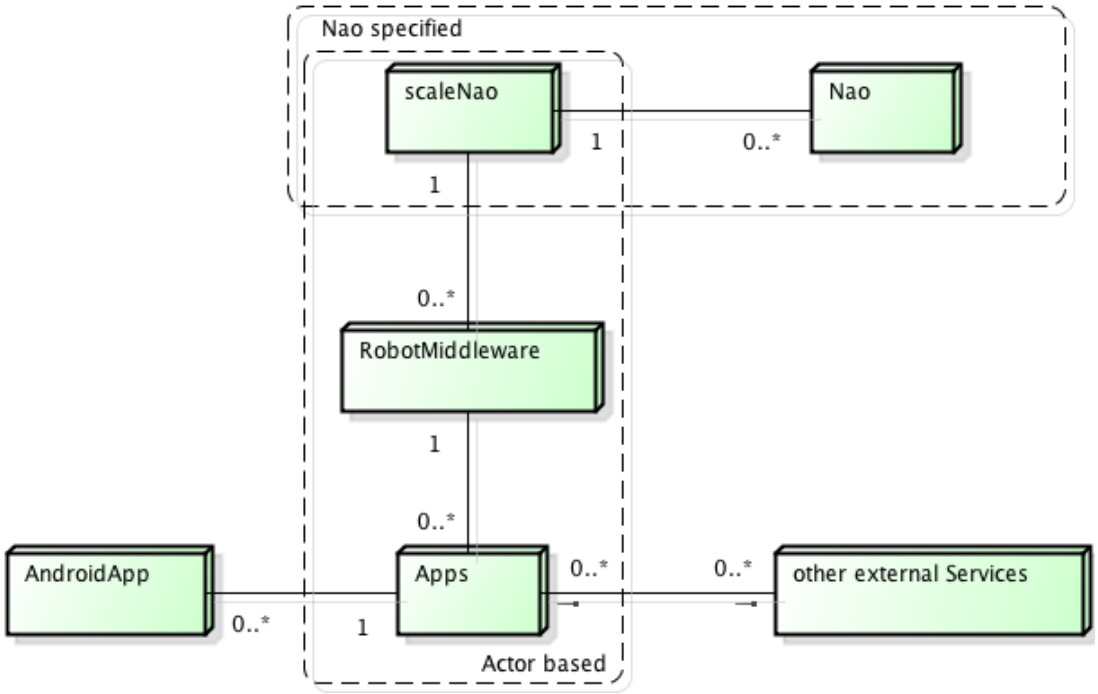

Abbildung 4.1: Verteilungssicht

Im gleichen Zug benötigt die RobotMiddleware genau eine Instanz zu scaleNao 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 auch den Zugang zu externen Diensten schaffen können ohne dabei RobotMiddleware oder scaleNao 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 Proobuf definiert.

Das Aktorensystem scaleNao 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 scaleNao delegiert werden, dort in Naoqi spezifische Aufrufe umgewandelt werden und zu Nao geschickt werden. Die Antwort wird über scaleNao 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.

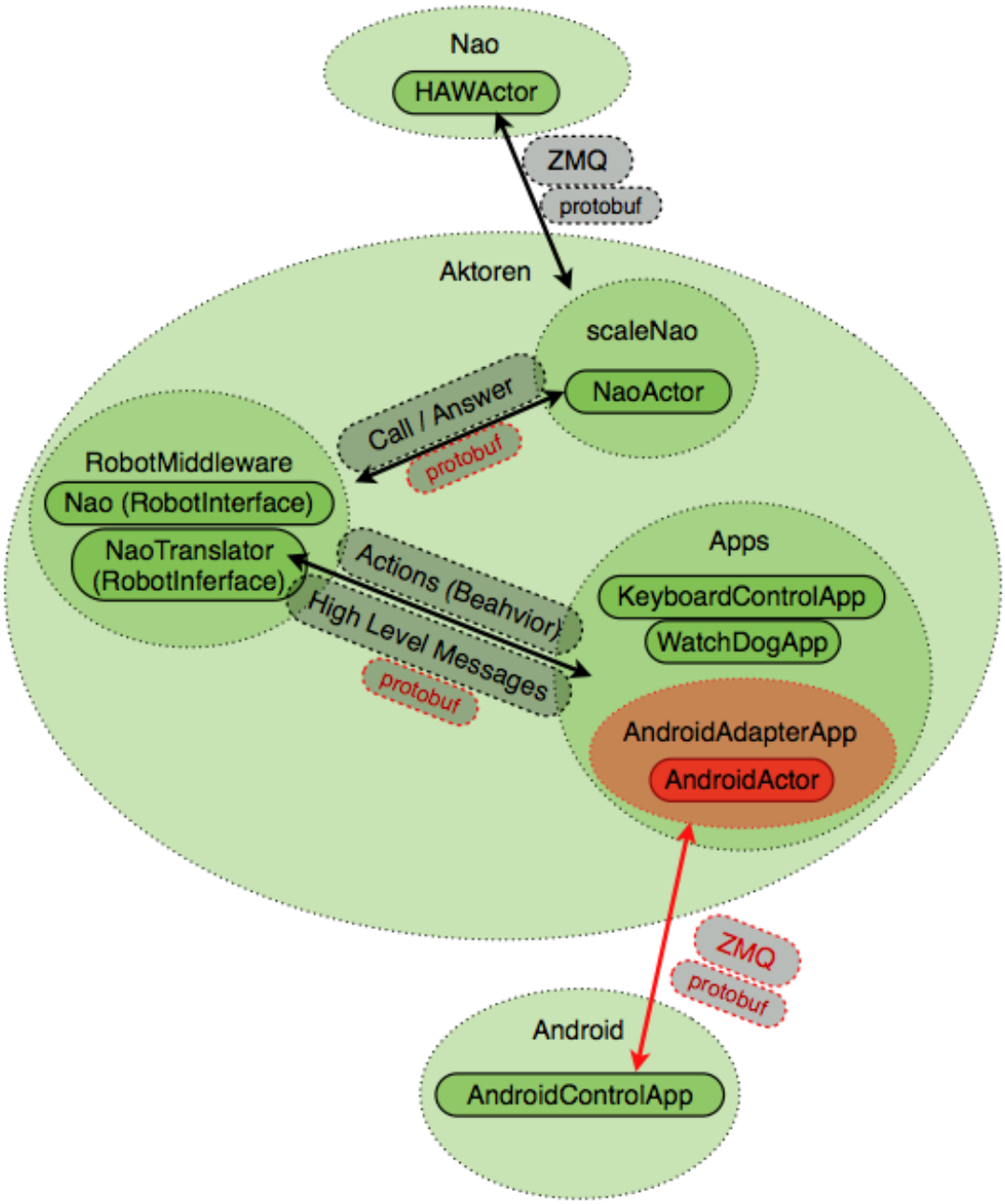

Abbildung 4.2: Kommunikationsschema

Das Aktorensystem scaleNao (Abb. 4.3 Aktorensystem scaleNao) 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). Der ResponseActor 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 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. Für der entfernten

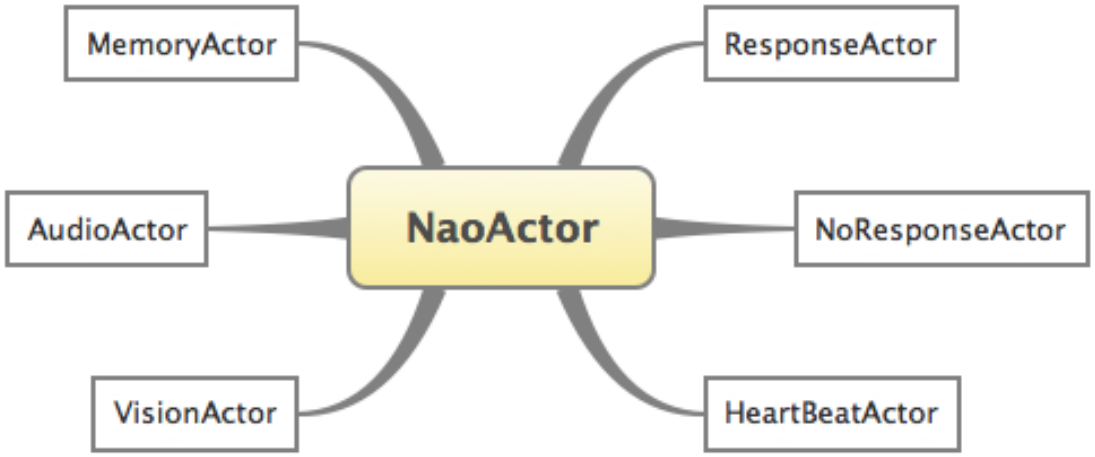

Abbildung 4.3: Aktorensystem scaleNao

Methodenaufruf ist eine Protobuf Nachricht definiert mit genau einem Modulnamen, einem Methodennamen und beliebig vielen Parametern. Dieser HAWActorRPCRequest wird von scaleNao 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 MixeValue 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 scaleNao 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 NaoActor

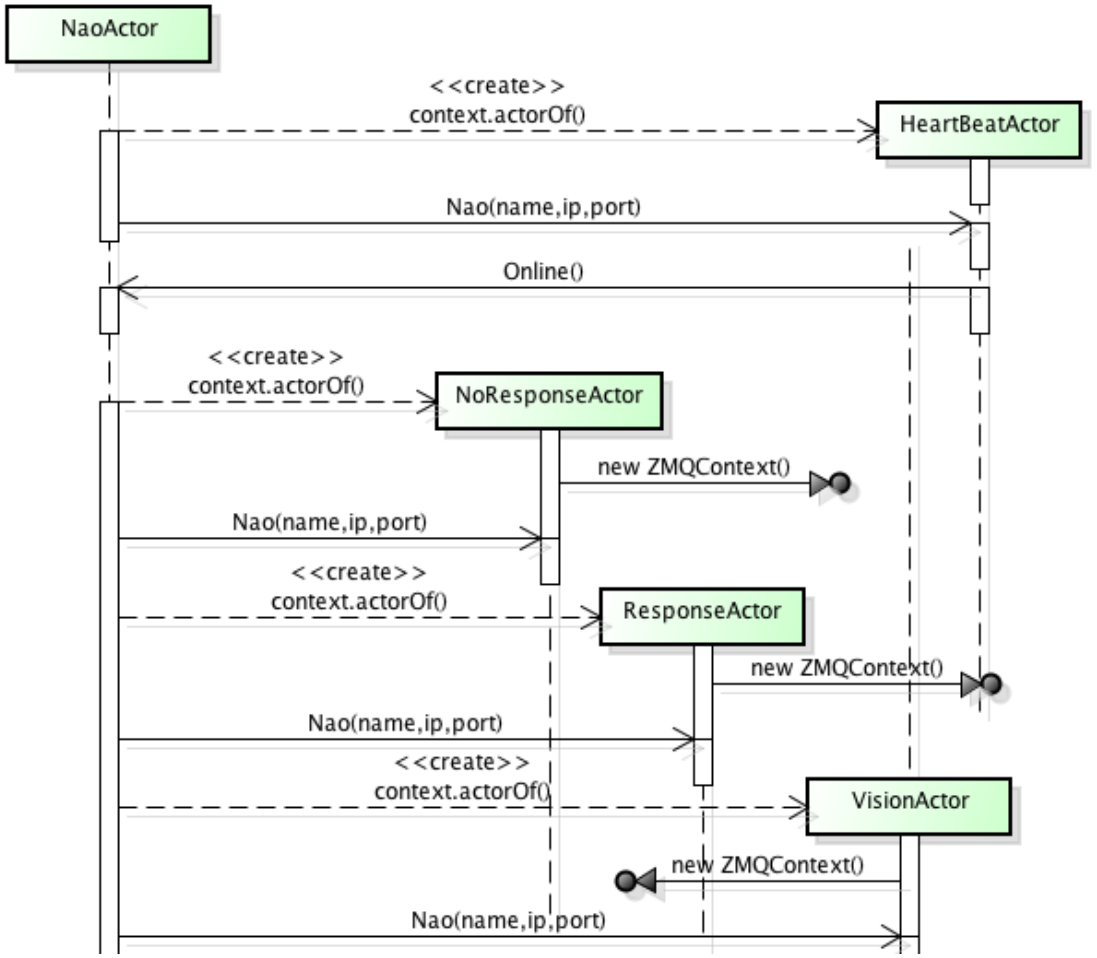

Abbildung 4.4: Initialisierung

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). Ohne dabei zu

viele Aktoren zu erstellen und ohne zu viele ZMQSockets gleichzeitig zu erstellen wird nach dem Senden ein Future erstellt. 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

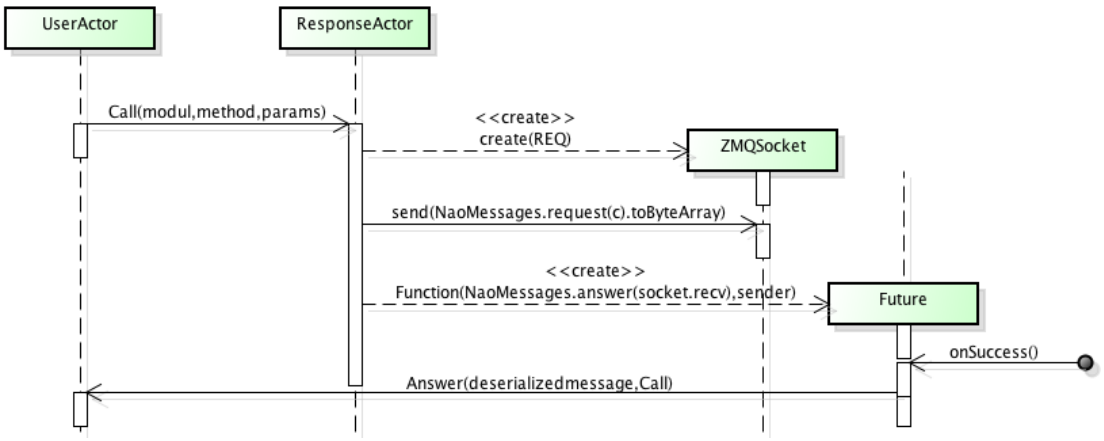

Abbildung 4.5: Reuquest-reply Kommunikation

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. Dazu hat der NoResponseActor zwei Zustände. Im Zustand communicating nimmt der 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.

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

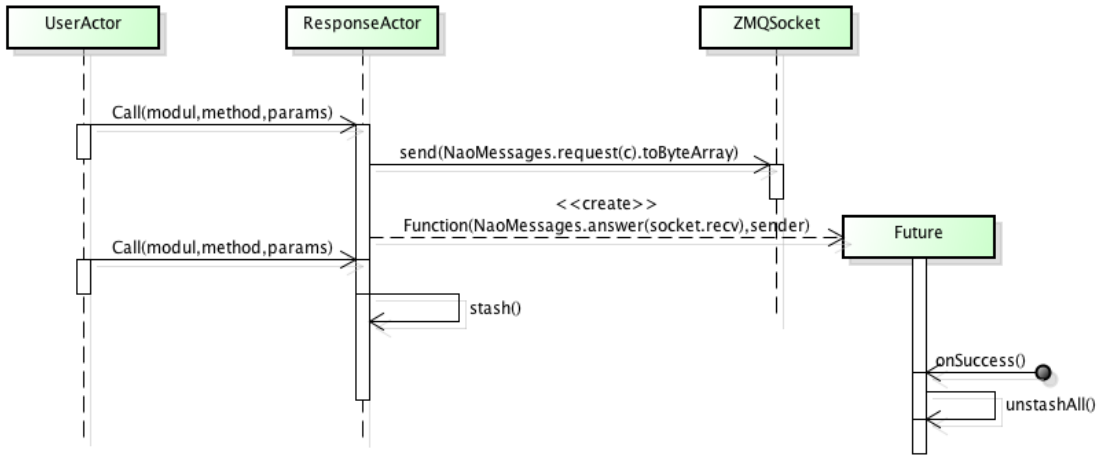

Abbildung 4.6: Vorgetäuschte unidirektionale Kommunikation

Der HeartBeatActor verkörpert die Idee auf Basis von Testnachrichten und einem fest definierten Timeout herauszufinden ob der Nao erreichbar ist. Kommt eine Antwort innerhalb des Timeouts, wird der Nao als erreichbar angenommen, wobei nach eine 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.

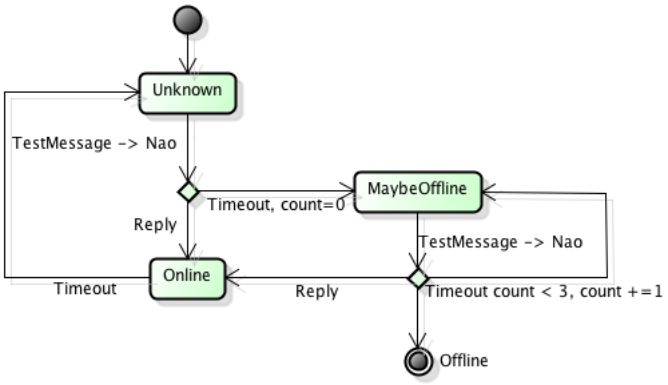

Abbildung 4.7: Heartbeat Zustandsautomat

Literaturverzeichnis

Aldebaran (2013). *Naoqi.* www.aldebaran-robotics.com/documentation/.

Apache (2013). *Thrift.* thrift.apache.org/docs/idl/.

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.

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.

Google (2013). *Protocol Buffers.* developers.google.com/protocol-buffers/docs/overview.

Hartmann, P. (2006). *Mathematik für Informatiker: Ein praxisbezogenes Lehrbuch (German Edition).* Vieweg+Teubner Verlag.

Hintjens, P. (2013). *Code Connected Volume 1: Learning ZeroMQ.* CreateSpace Independent Publishing Platform.

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.

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.

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

Wyatt, D. (2013). *Akka Concurrency*. Artima Inc.