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. Gezeugt wird die von mir entwickelte Kommunikation mit Nao und grundlegende Designentscheidungen, die eine greifbare Idee vermitteln sollen, welche Möglichkeiten diese Architektur bietet. Die Funktionsweise der Architektur wird dabei von einem Anwendungsbeispiel demonstriert.

4.1 Vorstellung des Roboters

Der Nao von Aldebaran enthält das Open Source Betriebsystem OpenNao (vgl. Aldebaran, 2013, Software in and out of the robot), dass auf die genutzte x86 Architektur zugeschnitten ist. Darauf wird das Metabetriebsystem Naoqi ausgeführt, welches ermöglicht den Nao zu steuern. Naoqi ist dafür in verschiedene C++ oder Python Module aufgeteilt, die spezielle Aufgaben übernehmen wie Bewegung (ALMotion), Sensoren (ALSensors) oder die Bereitstellung einer Datenbank (ALMemory). Mit diesen Modulen kann der Nao gesteuert werden. (vgl. Aldebaran, 2013, NAOqi Framework).

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

Von außen aufrufen bedeutet eine entfernte Methode über das WLAN oder Ethernet Netzwerk von einem anderen Rechner aufzurufen. Dafür stellt der Hersteller in mehreren Sprachen (u.a. Python und C++) implementierte SDKs zur Verfügung (vgl. Aldebaran, 2013, Development / SDKs), die für jedes Modul einen Proxyobjekt bereitstellen. Die Proxyobjekte stellen die API durch einen Remote Procedure Call bereit.

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

Für Java existieren automatisch generierte Bindings, die die API nur zum Teil abbilden und nach eigener Erfahrung grobe Fehler in der Javasyntax enthalten. Es werden keine Callbacks unterstützt, da Java keine Callbacks direkt unterstützt. Die Proxys müssen für jede Sprache und jede API Version geschrieben werden und erhöhen des Entwicklungsaufwand soweit, dass eine vollständige Umsetzung Schwierigkeiten bereitet. Die SDks werden für Windows, Linux und Mac bereitgestellt, allerdings für Windows nur in 32 bit.

Die Idee von Aldebaran ist ausschließlich über die SKDs mit dem Nao zu kommunizieren. Jedoch stößt man nicht nur beim Wahl seines Prozessors und seiner Sprache auf Restriktionen, sondern stößt man auch auf Geschwindigkeitsprobleme. Naoqi komprimiert keine Bilder, welches für eine rückenfreie Darstellung äußerst wichtig ist. Über Ethernet ist dies unkompliziert möglich, jedoch ein mobiler Roboter, der WLAN unterstützt, wird durch ein Kabel unnötig eingeschränkt. Allerdings auch einfache Befehle erscheinen durch die XML basierte Übertragung der Daten sehr schwergewichtig. Dies hat sich beim Entwickeln mit dem SDK gezeigt. Ein RPC ist in den SDKs immer blockierend realisiert, sodass, sodass nicht ohne weiteres nebenläufige Prozesse realisiert werden können.

Zur Entwicklung von eigenen Modulen, die direkt in naoqi integriert werden, bietet der Hersteller ein Cross-platform build system qiBuild (vgl. Aldebaran, 2013, qiBuild documentation). Damit lassen sich auf einem anderen Rechner Module kompilieren, die nativ auf dem Nao ausgeführt werden können. Zusammen mit dem OpenNao, welches ohne Paketverwaltung ausgestattet ist, ist das qibuild ein geschlossenes System. Die Integration neuer Software in einem geschlossenem System ist schwierig, denn es können die die vorgefertigten Werkzeuge nicht verwendet werden, sondern es müssen die Bibliotheken manuell installiert und gewartet werden. Es genügt dabei eine andere Version einer Bibliothek zu benötigen um in massive Schwierigkeiten zu geraten. Hat eine Bibliothek wenige bis keine Abhängigkeiten ist dies im Ausnahmefall noch machbar. Sobald die Abhängigkeiten zu groß werden, wird der Aufwand zu

groß. Da man die Abhängigkeit nicht über eine Paketverwaltung oder über die Buildumgebung nachinstallieren kann, bleibt einem nur das manuelle Installieren und Warten übrig.

Die Entwicklung von nativen Modulen ist auch durch die Rechenkapazität stark beschnitten, sodass rechenintensive Applikationen wie Bilderkennung nicht lokal ausgeführt werden sollten. Naoqi nutzt intern eine Synchronisierung der Daten, die speziell für die vom Hersteller ausgegebenen Module definiert ist. Aldebaran gewährleistet auch nur für diese Konfiguration die Funktionsfähigkeit. Funktionsfähigkeit bedeutet, dass der Nao im Betrieb nicht z.B. das Gleichgewicht verliert oder in einen Deadlock gerät. Da Naoqi im Gegensatz zu OpenNao nicht Open Source ist, kann die Implementierung auch nicht eingesehen werden. Eigene Erfahrungen zeigten, dass der Nao sehr einfach zu Fall gebracht werden kann, wenn man von Arme und Bein gleichzeitig bewegen möchte. Zusätzliche Module können daher nur mit Vorsicht und auf das wesentliche reduziert in den Nao integriert werden.

4.2 Anwendung: Steuerung durch Smartphone und Spielecontroller

Ziel des Projekts war die Realisierung ein konkretes Anwendungszenario. Dazu wurden verschiedene Controller zu Hand genommen. Ein Android Smartphone, ein XBOX Spielecontroller und eine Tastatur sollen benutzt werden können um den Nao zu navigieren.

Die Controller haben verschiedene Stärken und Schwächen und sind daher bewusst ausgewählt um die Steuerung auf Vielseitigkeit zu testen. Die Tastatur besteht nur aus Tasten und eignet sich daher für wenige bestimmte Aktionen wie Rechts, Links, Vorwärts, Rückwärts. Ein Winkel ist nur umständlich abbildbar. Der XBOX Controller besitzt zusätzlich einen Joystick und kann einen Winkel durch daher deutlich einfacher abbilden.

Das Smartphone kann Bilder darstellen und damit auch das Videobild das Naos. Dann ist es denkbar allein mit dem Smartphone ohne direkte Sicht den Nao zu steuern. Allerdings ist dazu auch eine Bewegung des Kopfs nach rechts, links oben und unten nötig. Kann man den Kopf bewegen, sieht man auch wohin der Nao sich bewegt. Zusätzlich können virtuelle Buttons dargestellt werden. Ein ähnliches Verhalten zu einem Joystick ist auch denkbar.

Zusätzlich muss ein Aufstehen und Hinsetzen per Knopfdruck möglich sein. Denn seine sichere Position, in der dieser er wenig Strom benötigt und nicht hinfallen kann, ist auf dem Boden zu sitzen. Dort sitzt er mit dem Armen auf den Knien aufstützend. So kann der Nao auch ohne Strom nicht mehr umfallen. Von dieser Position muss der Nao erst sich aufstellen um Laufen zu können.

Die Anforderung ist, dass alle Controller über eine App in einem Aktorensystem in Echtzeit einen Nao steuern können und dabei ihre (oben beschriebenen) Stärken ausspielen können. Die Steuerung soll losgelöst vom Nao und der Naoqi API auf einem eigenen Rechner ausgeführt werden. So wird zum einem der Naoi nicht unnötig belastet und eine Unterstützung anderer Roboter bleibt möglich.

Naoqi ist in C++ und Python geschrieben und das Aktorensystem Akka in Scala, damit sind mindestens zwei verschiedene Sprachen, die miteinander kommunizieren müssen um eine Kommunikation zu ermöglichen. Die Nachrichten sollen dazu unabhängig von einer Zielsprache mit der Protobuf IDL definiert und mit ØMQ transportiert werden.

4.3 Kommunikation zwischen Nao und dem Aktorensystem

Kommuniziert werden soll ausschließlich über Nachrichten, diese müssen transportiert werden. OpenNao bietet eine Unterstützung für ØMQ in der Version 2.2, die auch in der qibuild direkt verwendet werden kann. ØMQ wird derzeit nicht in Naoqi direkt verwendet, wird trotzdem offiziell angeboten und bietet sich daher an. Die direkte Unterstützung von Protobuf durch ØMQ bestärkt den Beschluss ØMQ für die Kommunikation zwischen Nao und Akka zu verwenden. ØMQ unterstützt Protobuf durch expliziten Hinweis in der Dokumentation (vgl. Hintjens, 2013, S. 16).

ØMQ zu verwenden bedeutet, dass die konventionelle Kommunikation über die Proxyobjekte des SDKs nicht mehr möglich ist, da die Schnittstelle zwischen dem SDK und dem Nao nicht offengelegt ist. Daher wurde eine eigenes Naoqi Modul nötig, dass die Kommunikation über ØMQ und mit Protbuf auf dem Nao ermöglicht.

Die Naoqi API ist nach einer stringenten und sehr einfachen Struktur aufgebaut. Es gibt Modulnamen wie z.B. ALTextToSpeech und Methodennamen wie z.B. getVolume (s. Formel

4.1). Diese beiden Informationen genügen um die Methode über den Broker aufzurufen. Eine Möglichkeit wäre es gewesen eine statische Abbildung $Message \rightarrow Method$ mit einem Dispatcher (switch case) zu realisieren. Es hätten allein für das Modul ALMotion über 80 Methoden mit einem dispatch versehen werden müssen. Da die API so stringent und einfach aufgebaut ist, ist ein reflektier Ansatz naheliegend. Reflektiv bedeutet dass die Methoden in diesem Fall über zwei Strings übertragen werden und zur Laufzeit zu einem Methodenaufruf zusammengesetzt und aufgerufen werden (vgl. Colouris et al., 2012, S. 55). Der HAWActor ist ein von David Olszowka entwickeltes Naoqi Modul HAWActor, welches der Naoqi API nachempfunden, entfernte Methodenaufrufe realisiert.

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

Etwas komplexer sind die möglichen Parameter. Naoqi verwendet generell nur die Basistypen Strings, Integer (32 Bit signed), Float (32 Bit signed), Byte und Boolean. Zusätzlich werden Arrays der Basistypen unterstützt. Die Anzahl der Parameter ist beliebig, ALTextToSpeech.getVolume benötigt keinen Parameter. Insbesondere im Bereich der Bewegung gibt es Methoden mit mehreren Parametern. Am Beispiel der Positionsangabe ALMotion.getCOM (s. 4.2) sieht man mehrere Parameter. Zuerst der Name des Körperteils als String, danach der Koordinatenraum, der durch Ints aus der Menge { FRAMETORSO = 0, FRAMEWORLD = 1, FRAMEROBOT = 2 } angegeben wird, außerdem ein Boolean, dass die Verwendung von Sensoren zur Positionsbestimmung aktivieren kann. Für den reflektiven Ansatz bedeutet dies, dass aus einer überschaubaren und konstanten Menge an Typen eine Liste von Parametern nötig ist um Naoqi Methoden aufzurufen. Naoqi gibt damit keine naoqispezfischen Typen nach außen oder fordert welche durch Parameter in der Naoqi API ein.

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

Der Vorteil des reflektiven Ansatz liegt in einer auf das nötigste reduzierten Implementierung, die unabhängig von Aktualisierungen der Naoqi API funktioniert. Ein Dispatcher müsste bei jeder Aktualisierung angepasst werden.

Noch nicht angesprochen ist der Rückgabewert. Viele Methoden, insbesondere im Bereich Bewegung, enthalten gar keinen Rückgabewert. Wenn jedoch ein Rückgabewert vorhanden ist, ist der Typ dabei, wie bei den Parametern, ein Basistyp. Die Anzahl ist dabei nicht beliebig, es

ist entweder genau einer oder keiner. Für ALMotion.getCOM ist der Rückgabewert ein Array von Floats.

Der HAWActor erhält eine serialisierte Request Nachricht, die aus Modulname, Methodennamen und Parameter besteht und, diese in einen Methodenaufruf umwandelt (Unmarshalling) und den Reply (der auch leer sein kann) serialisiert (Marshalling) und zurücksendet. Die binäre Kommunikation wird über das TCP Socket basierte ØMQ umgesetzt. Für Naoqi wurden dabei spezielle Datenstrukturen durch Protobuf definiert.

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

```
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 unterstützt. Ein Array kann man dabei als beliebige Wiederholungen von MixedValue auffassen. 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 enthält im positiven Fall genau einen Rückgabewert des Typs 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.

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

Wenn der HAWActor mit qibuild kompiliert wurde und vom naoqi Broker gestartet wurde, ist dieser über das Netzwerk ansprechbar. Der HAWActor nutzt für die Netzwerkkommunikation das Request-reply Pattern von ØMQ, indem dieser einen REP (Reply) Socket anbietet und auf diesem mit dem standardmäßigen Port 5555 lauscht.

Um eine Kommunikation zu ermöglichen, wird noch ein weiterer Computer benötigt mit der Clientanwendung, die den passenden ØMQ Port REQ (Request) initialisiert hat. Da später Akka verwendet werden soll, benötigen wir dafür Java oder Scala Bindings, da keine Java oder Scala Implementierung verfügbar ist. Speziell für die Bindings gibt es die native Bibliothek jzmq.

```
  val context = new ZContext
  def socket(url: String) = {
      val sock = context.createSocket(ZMQ.REQ)
      sock.connect(url)
      sock
  }
```

Es wird ein ØMQ Context erstellt, der die Sockels verwaltet und ein ØMQ Socket des Typs REQ angeboten, der sich zu einer bestimmten URL wie z.B. tcp://127.0.0.1:5555 verbindet.

Werden die Protobuf Nachrichten als .proto gespeichert kann der Protobufcompiler diese in der Zielsprache wie beispielsweise Java überführen und in das Projekt eingebunden werden. Der Protobufcompiler erstellt für jede Nachricht einen Builder, mit dem Instanzen der generierten Klassen erzeugt werden können. Dem Builder werden mit set Methoden wie setModule(s:String) die Daten Stück für Stück übergeben. Dabei verändert die set Methode

nicht der Builder sondern es wird ein neuer Builder erzeugt. Zum Abschluss wird mit der Methode build die Datenstruktur erzeugt und zurückgegeben.

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

Parameter werden in der oben beschriebenen Methode request als Liste des Typs MixedValue übergeben und müssen dort nur noch hinzugefügt werden. Erstellt werden MixedTypes aus den Scala Basistypen Int, Float, Boolean, Byte und String. Alles andere führt zu einer Exception. In Scala sind die Basistypen Subtypen des Typs AnyVal. Diese haben u.a. die Besonderheit nicht null sein zu können. Aus Kompatibilitätsgründen zu Java gehören Strings nicht dazu, denn diese können null sein. Daher kann der Übergabeparameter array nicht von Any auf AnyVal eingeschränkt werden. Dazu kommt, dass AnyVal noch weitere Typen wie Unit als Subtypen hat. Eine totale Abbildung $AnyVal \rightarrow MixedValue$ wäre auch ohne String nicht möglich. So muss für nicht unterstützte Typen eine Exception geworfen werden.

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

Da die Methode anyToMixedVal eine implizite Methode ist, muss diese nicht direkt aufgerufen werden, sondern wird vom Compiler aufgerufen, wenn eine Typkonvertierung nötig ist. Für Arrays ist auf die gleiche Weise eine implizite Methode definiert. Damit ist es möglich die Methode request mit ausschließlichen Scalatypen aufzurufen.

$$val\ req = request(\text{"}ALMotion\text{"}, \text{"}getCOM\text{"}, List(\text{"}HeadYaw\text{"}, 1, true))$$

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

Die Nachricht hat eine sehr geringe Größe, daher kann die Nachricht problemlos in einem Frame verschickt werden. Die Endmarkierung der Nachricht bleibt damit die standardmäßige 0. Somit genügt es dem Socket das Bytearray von req zu übergeben und es wird an die angegeben URL versendet.

$$socket.send(req.toByteArray, 0)$$

Die Antwort kann nun über die Methode recv mit Angabe der gleichen Endmarkierung 0 abgefragt werden. Dabei ist zu berücksichtigen, dass im Gegensatz zu der Methode send die Methode recv blockierend ist. Der der Sendevorgang wird in einem Thread in ZContext verarbeitet.

$$socket.recv(0)$$

Kehrt die Methode recv zurück, erhält man wiederum ein Bytearray. Nun kann die automatisch generierte Protobufmethode parseFrom(a:Array[Byte]) verwendet werden um aus dem Bytearray wieder eine Protobufdatenstruktur zu erstellen. Nun muss mit der Methode hasError geprüft werden, ob ein Fehler vorliegt. Diese wird automatisch generiert, da die Nachricht einen optionales Feld error enthält. Ein Fehler liegt vor, wenn eine Methode nicht existiert oder nicht ausgeführt werden konnte. Sollte kein Fehler vorliegen kann auf auf gleiche Weise geprüft werden, ob der optionale Rückgabewert returnval gesetzt ist. Der returnval ist gesetzt, wenn die Methode gefunden wurde, keinen Fehler verursacht hat und als Rückgabewert kein void hat. Ist dies auch nicht der Fall, ist die Nachricht leer. Es kommt eine leere Nachricht zurück, wenn die Naoqi Methode einen Rückgabewert void hat.

```
1  def answer = {
2    val resp = HAWActorRPCResponse.parseFrom(socket.recv(0))
3    if (resp.hasError) {
4      trace("Error: " + resp.getError)
```

```
5     } else if (resp.hasReturnval) {
6       trace("-> " + resp.getReturnval)
7     } else {
8       trace("-> Empty \n");
9     }
10  }
```

Da Naoqi keine Komprimierung für Videodaten unterstützt, entwickelte David Olszowka das Naoqimodul HAWCamServer, dieser ist ähnlich zu HAWAktor aufgebaut. Dieser versteht eine Requestnachricht, die die Konfiguration von Auflösung, Farbbereich und Framerate pro Sekunde zulässt. HAWCamServer verschickt darauf eine Antwort mit genau einem Bild. Es wird genau ein Bild verschickt, da sonst das Request-reply Pattern nicht eingehalten wird. Der VisionActor ist das Gegenstück im Aktorensystem, dass die Requestnachricht CamRequest verschickt und auf die Antwort CamResponse wartet.

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

Ähnlich zu HAWActorRPCResponse sendet der HAWCamServer das konfigurierte Bild vom Form von Bytes zurück und enthält bei Bedarf einen Fehlerstring.

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

Da nicht nur viele Bilder nur sehr schnell hintereinander geschickt werden sollen, sondern die Bilder trotz Kompression relativ groß sind, ist auch die Verarbeitung relativ aufwändig. Das erstellen Einer Protobufdatenstruktur sollte möglichst erst dann erfolgen, wenn dies auch nötig ist. Angezeigt werden soll das Bild am Ende auf einem Smartphone. Auf dem Weg vom Akotorensystem auf das Smartphone muss gegebenenfalls muss die Nachricht wieder serialisiert werden. Im Aktorensystem wird die Nachricht jedoch nicht weiter bearbeitet, d.h. der Inhalt kann verborgen bleiben. Daher bietet der VisionActor zwei Varianten an. Einmal wird die Antwort direkt in eine Protobufdatenstruktur umgewandelt, das andere Mal wird das Bytearray unangetastet weitergesendet.

4.4 Verteilung

Die Kommunikation zwischen dem Nao und dem Aktorensystem ist die Schlüsselstelle zur Steuerung des Naos. Betrachtet man mehrere Roboter, ein Handy, Webservices wie Newsreader oder Mailbox und Peripherie wie ein Controller für Spielekonsolen, stellt man fest, dass für die Kommunikation mit dem Nao ein abgeschlossener Baustein im Gesamtsystem nötig ist, der als Service für das Restsystem zur Verfügung steht. Denn verschiedenste Geräte wollen auf den Nao zugreifen. Dieser Baustein hat den Namen naogateway bekommen. Der Naogateway ist genau ein Aktorensystem, welches unabhängig von den anderen Komponenten auf einem Rechner läuft (Abb. 4.1 Verteilungssicht).

In der Mitte steht das Aktorensystem Robot Middleware, dass High Level Funktionen unabhängig von der Hardware für Apps zur Verfügung stellt. Robot Middleware wurde von mir nicht entwickelt, sondern ist ein weiterer Baustein, der im Projekt entstanden ist.

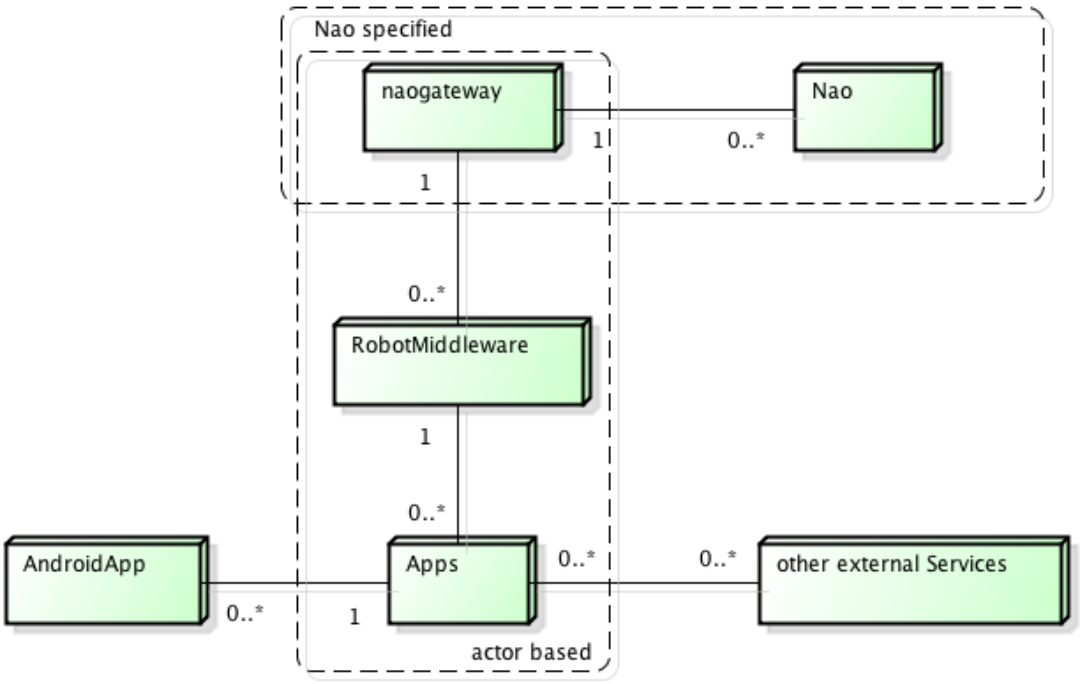

Abbildung 4.1: Verteilungssicht

Im gleichen Zug benötigt die RobotMiddleware, die die hochsprachlichen Funktionen wie beispielsweise Laufen beherbergt, genau eine Instanz zu naogateway um einheitliche Ent-

scheidungen 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 RobotMiddleware benutzen. Jedoch können diese auch den Zugang zu externen Diensten erlangen ohne dabei RobotMiddleware oder naogateway zu berühren.

4.5 Kommunikationsmechanismen

Für die Kommunikationsmechanismen ist ein etwas tieferer Blick in die einzelnen Komponenten nötig (Abb. 4.2 Kommunikationsschema). Im Grundsatz gibt es vier Bereiche, die miteinander kommunizieren müssen.

Das Aktorensystem naogateway ist speziell für den Nao geschrieben und verbindet sich über ØMQ 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 über den eigenen Aktor NaoTranslator an naogateway delegiert werden, dort in Naoqi spezifische Aufrufe umgewandelt werden und zu Nao geschickt werden. Die Antwort wird über naogateway wieder zum NaoTranslator zurückgeleitet.

Apps agieren in einem eigenen Aktorensystem um völlig unabhängig zu sein. Die Apps haben die Möglichkeit auf die NaoInterface API zuzugreifen und gleichzeitig externe Dienste zu nutzen um eine vollwertige App zu schaffen, die ein Nao steuern kann. Die Idee ist Nachrichten vollständig über Protobuf abzubilden und dabei von speziellen Roboterschnittstellen wie Naoqi zu abstrahieren. Akka unterstützt nativ Protobuf als Serialisierer, sodass dies sich gut ins System einbinden lässt. 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.

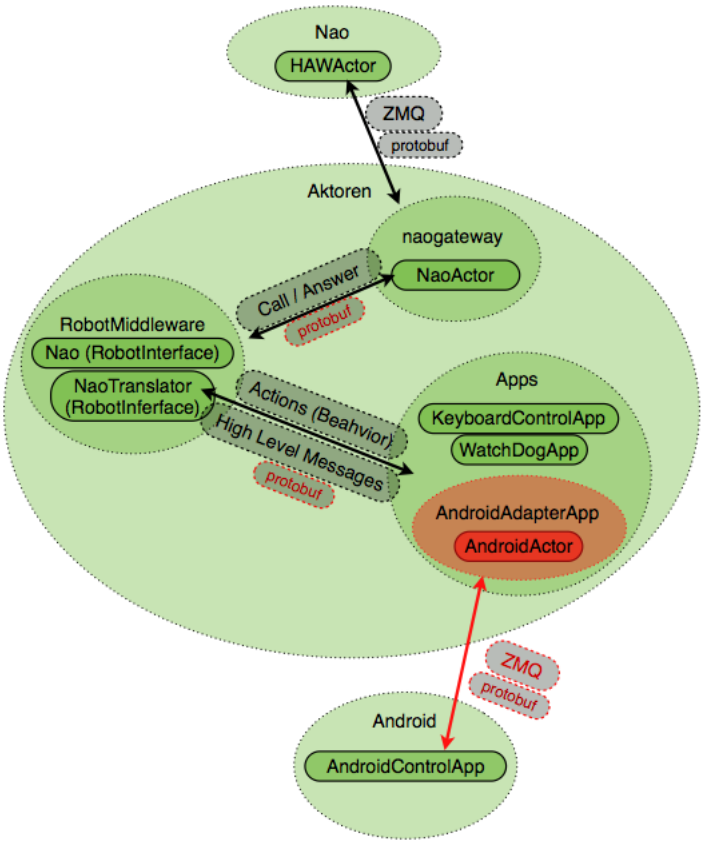

Abbildung 4.2: Kommunikationsschema

Die AndroidControlApp kommuniziert derzeit ähnlich wie der naogateway direkt mit dem Nao. Dies soll nur ein Zwischenschritt sein. Der Fokus stand dort auf der ruckelfreien Übertragung der Kamerabilder des Naos.

Das Aktorensystem naogateway (Abb. 4.3 Aktorensystem naogateway) 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 Antwort kann auch im Sinne von void leer sein. Diese Request-reply Kommunikation wird durch einen ResponseActor umgesetzt. Der NoResponseActor verwirft jegliche Antwort, 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 Naoqi Funktion einen verwertbaren Rückgabewert hat oder nicht. Der VisionActor ermöglicht die Abfrage von komprimierten Bildern von der Kamera des Nao. 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.

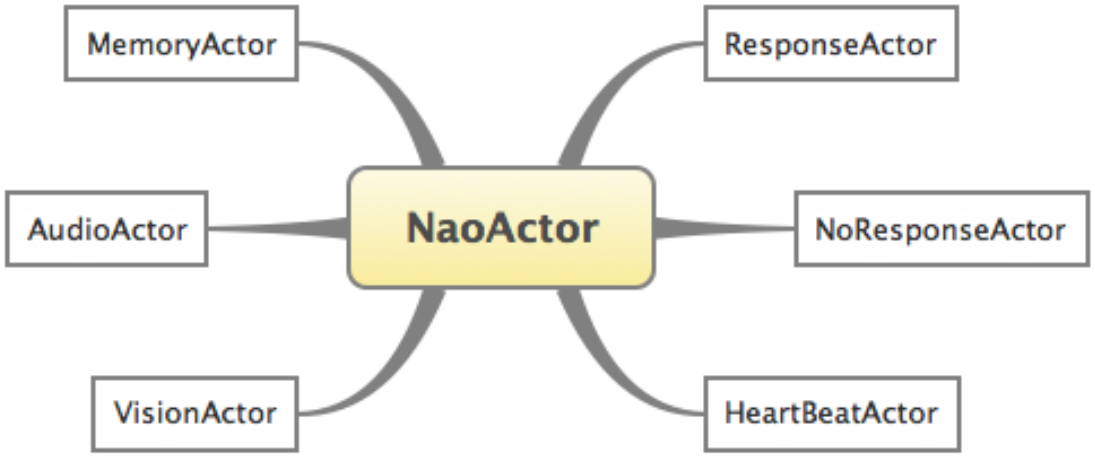

Abbildung 4.3: Aktorensystem naogateway

4.6 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 ØMQ Context von verschiedenen Aktoren nicht gleichzeitig angesprochen werden kann, benötigt jedes Kind seinen eigenen. ØMQ Context hat eine eigene Threadverwaltung, die mit dem Dispatcher des Aktorensystems nicht zusammenarbeitet.

HeartBeatActor, NoResponseActor, ResponseActor und VisionActor benötigen jeweils die Zugangsdaten zum Nao, die diese von dem NaoActor bekommen. Die Zugangsdaten sind

verpackt in einer case class Nao, die den Namen, Host und Port in sich trägt. Der erste Gedanke war die Aktoren durch Message Passing die Zugangsdaten mitzuteilen. Mit dieser Nachricht gehen die Aktoren in einen neuen Zustand über, indem sie ihre eigentliche Aufgabe ausführen können. Da dadurch die Möglichkeit bestand, dass ein Aktor ein Call schickt, bevor die Aktoren bereit sind, könnten Nachrichten nicht verarbeitet werden. Um dieses Problem zu lösen, werden die Daten über den Konstruktur übergeben. Alle vier Aktoren benötigen damit nur noch genau einen Zustand, in dem sie ihre Aufgabe erfüllen können.

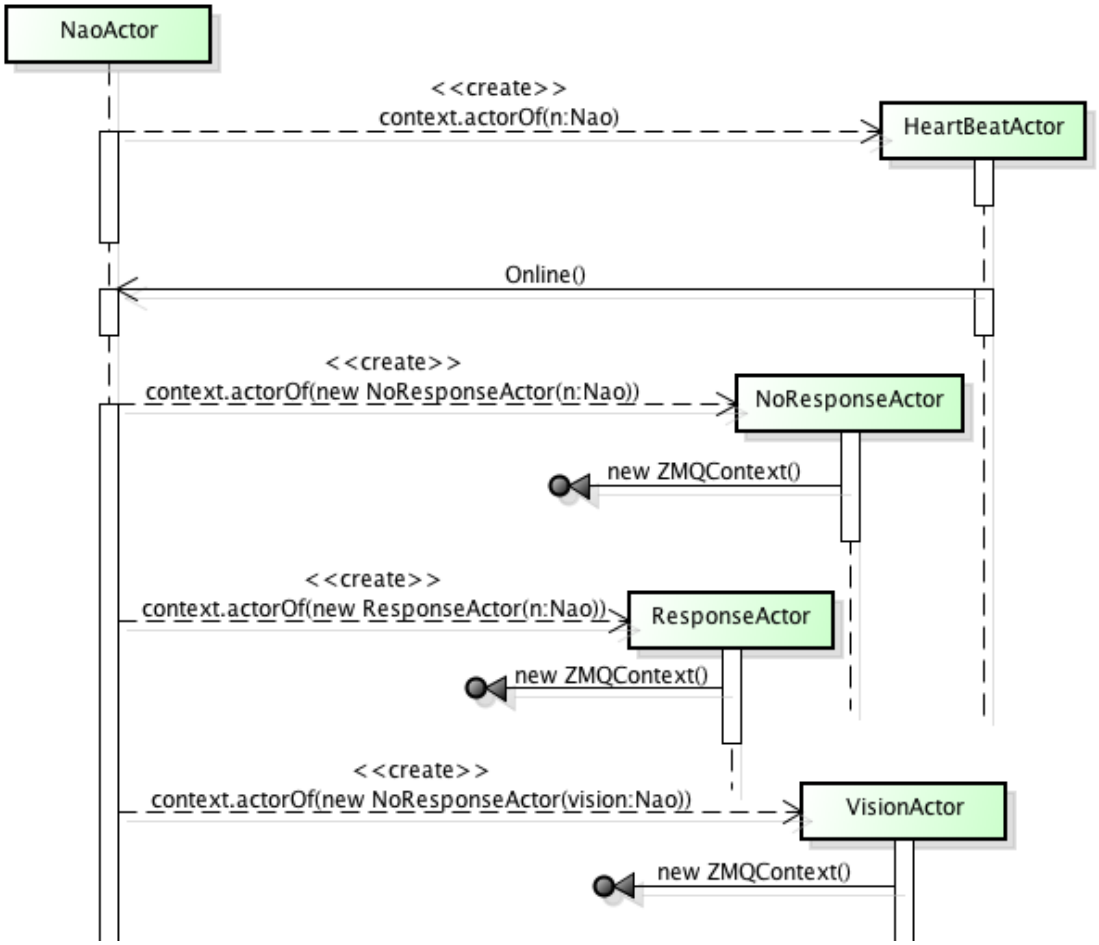

Abbildung 4.4: Initialisierung

Ist der NaoActor mit seinen Kindern hochgefahren, kann durch einen Connect Nachricht Aktorreferenzen abgefragt werden. In einem Tripel (reponseActorRef, noResponseActorRef, visionActorRef) werden die Aktorreferenzen versandt, diese können verwendet werden um

an einen ausgewählten Aktor einen Request zu schicken. 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 Request-reply Kommunikation). Die drei Aktoren kann man als Worker auffassen. Diese realisieren auf Anfrage die Kommunikation zwischen Aktorensystem und dem Nao.

Die Anfrage an den ResponseActor und den NoResponseActor ist die Nachricht Call (s. Formel 4.3). Call kapselt die Protobufnachricht HAWActorRPCRequest indem es einen Modulnamen, einen Methodennamen und eine Liste von Parametern als MixedValue beherbergt. Dazu gibt es die Nachricht Answer, die den Rückgabewert und den Call enthält. Die Nachricht Answer enthält den Call um eine Identifizierung zu ermöglichen. Wenn ein beliebiger Aktor einem der drei Worker mehr als einen Call schickt, ist nicht geregelt, wann die jeweilige Antwort zurückkommt. Es könnte auch eine Antwort vollständig ausfallen. Da Aktoren nebenläufig arbeiten, kann keine Aussage über das Eintreffen der Nachricht getroffen werden. Sollte ein Fehler in der Protobufnachricht sein, da z.B. die Methode nicht gefunden wurde, wird eine InvalidAnswer zurückgeschickt.

$$\begin{aligned} val\ call = Call('ALTextToSpeech,'\ getVolume) \\ responseActor\ !\ call \end{aligned} \tag{4.3}$$

Durch den Call ist die Antwort eindeutig, sofern verschiedene Calls versendet werden. Es ist dafür nötig sich die Referenz auf den verwendeten Call zu merken, dann kann mit Patternmatching (s. Formel 4.4) auf das innere der case class gematched werden.

$$case\ Answer(call, value) \tag{4.4}$$

Der HAWAktor nutzt strikt das Request-reply Pattern, d.h. wenn ein Aktor ein ØMQ Socket nutzt um den Call zu schicken, darf der Socket nicht direkt genutzt werden um den nächsten Call zu schicken. Es können zwar ohne weiteres viele Sockets benutzt werden, es muss jedoch auch sichergestellt sein, dass nach einem Call auf einen Socket die Antwort zunächst verarbeitet wird und danach der Socket entweder weiterverwendet oder explizit geschlossen wird. ØMQ nutzt keine Garbagecollection. Gleichzeitig darf der Aktor nicht blockieren, da dieser sonst nicht mehr erreichbar wäre. Blockieren kann der Aktor beispielsweise durch die Methode recv. Wenn eine Nachricht nicht ankommt, würde der Aktor nicht mehr reagieren.

Die erste Idee war einen eigenen Aktor zu verwenden, der nur Nachrichten schickt und empfängt. Dieser Aktor wird für jeden Call instanziiert und nach dem die Antwort weitergeleitet wurde wieder geschlossen. Man hätte dazu einen Pool von Aktoren anlegen müssen um die Zahl der Aktoren unter Kontrolle zu halten.

Die Lösung war unnötig komplex, denn für solche überschaubaren Aufgaben sind Futures absolut ausreichend. 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 ØMQ Socket wird danach geschlossen. So wird das Request-reply Protokoll korrekt eingehalten ohne zu blockieren. Das Senden einer Nachricht ohne eine Antwort

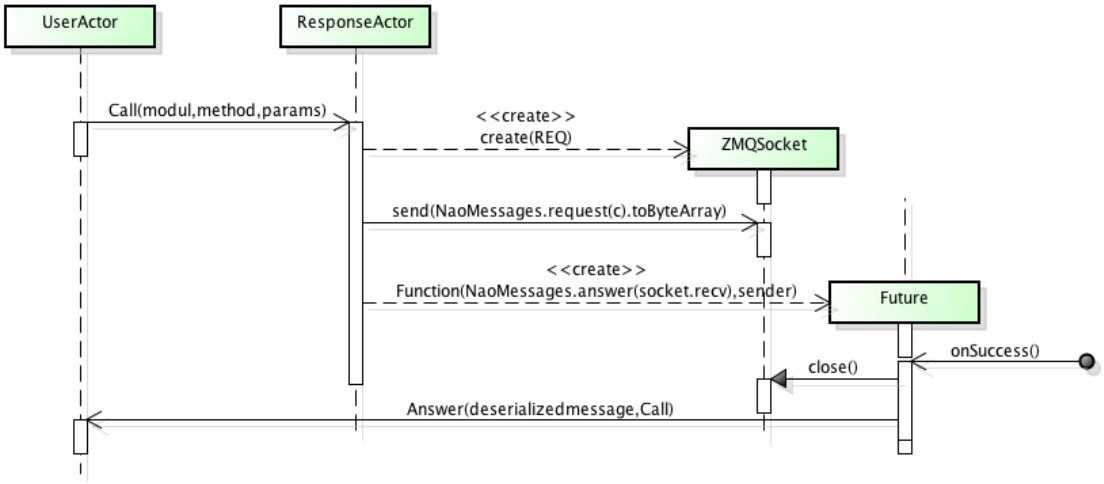

Abbildung 4.5: Request-reply Kommunikation

zu erhalten erfolgt über den NoResponseActor (Abb. 4.6 Vorgetäuschte unidirektionale Kommunikation). Der NoResponseActor muss auch das Request-reply Protokoll korrekt einhalten ohne zu blockieren, denn der HAWAktor erfordert dieses strikte Kommunikationsmodell.

Dieser funktioniert ähnlich zum ResponseActor. Es stellt sich an dieser Stelle die Frage, wie man mit Calls verfährt, während eine Antwort noch nicht vorhanden ist. Eine erste Lösung war folgende. 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. Dies wird Stashing genannt. 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 in die Mailbox zurückgelegt wird. Wenn die Antwort gekommen ist, werden alle Nachrichten, die zur Seite gelegt wurden, wieder in die normale Nachrichtenschlange geschoben. Die Antwort wird nicht weitergeleitet. Verwendet wurde dazu immer nur ein ØMQ Socket.

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

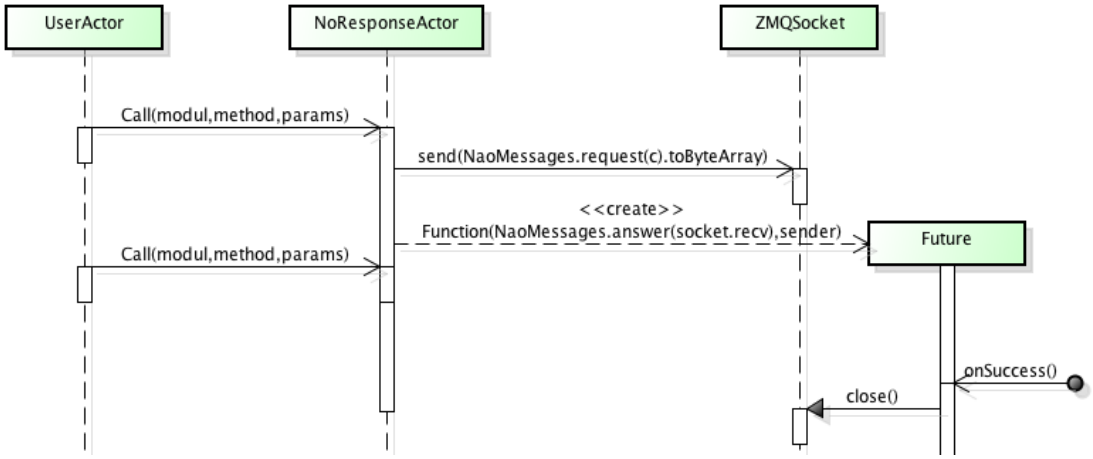

Abbildung 4.6: Vorgetäuschte unidirektionale Kommunikation

Der VisionActor ist im im Grunde aufgebaut wie ein ResponseActor, jedoch basierend auf eigenen Protobuf Nachrichten. Der VisionActor kann auf Anfrage einzelne Bilder liefern. Analog zum Call gibt für Bilder den VisionCall und den RawVisionCall, die beide das trait VisionCalling implementieren. VisionCalling kapselt den CamRequest, indem es Auflösung, Farbbereich und Framerate pro Sekunde speichert. Diese sind als Enumeration implementiert, da jeweils nur eine sehr geringe Anzahl an Wert möglich ist. Es werden vier Auflösungen, sechs Farbräume und 1-30 Frames pro Sekunde unterstützt. Schickt man einen VisionCall an den VisionActor, erhält man ein CamResponse, schickt man ein RawVisionCall erhält man das unangetastete Bytearray.

```
trait VisionCalling{
  val resolution:Resolutions.Value
  val colorSpaces:ColorSpaces.Value
  val fps:Frames.Value
}
```

Der HeartBeatActor soll auch in einem bestimmten Intervall erneute Testnachrichten schicken um den Nao nicht unnötig zu belasten. Dafür wird auch ein Timer benötigt. Abschließend soll auch der (No)ResponseActor in der Lage sein einer Verzögerung einzuleiten für bestimmte Nachrichten. Beispielsweise macht es keinerlei Sinn innerhalb von wenigen Millisekunden dem Arm nach oben und wieder nach unten zu bewegen. Auch hierfür ist ein Timer nötig. Man hätte diesen Timer auch in den Actor verlagern können, der den Response oder NoResponseActor benutzt, allerdings hätte dies viel doppelten Code erzeugt und wen der Nao dies später doch selber unterstützt, wäre die Anpassung sehr aufwändigt geworden.

Ein nicht blockierender Timer kann einem Future realisiert werden. Akka unterstützt dies direkt durch das after Pattern. Der Funktion after wird eine Zeitangabe (Duration) und ein ExecutionContext (z.B. der Akka Schedular) übergeben. Als Aufgabe wird eine Nachricht Timeout an sich selbst zu schicken definiert. Der ExcutionContext führt nach Ablauf der Zeitangabe das übergeben Future aus.

```
import akka.pattern.after
class TimerActor extends Actor {
      def receive = {
        case TimeOut =>
        case _ => after(2000 millis,
               using = context.system.scheduler){
             Future{
               self ! TimeOut
             }
          }
      }
}
```

Die Verzögerung für den ResponseActor und den NoResponseactor wird das trait Delay gelöst. Darin wird die die Konfiguration des Namespaces responseactor.delay geladen. In diesem Namespace werden die Verzögerungszeiten in Millisekunden angegeben. Da die Konfiguration nur Kleinbuchstaben zulässt, werden Modulnamen und Methodennamen klein geschrieben.

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

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

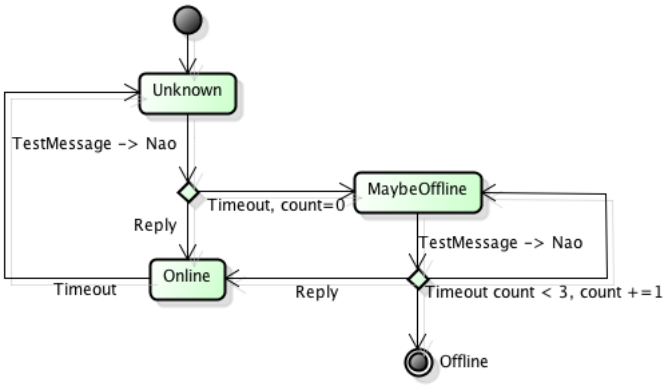

Abbildung 4.7: Heartbeat Zustandsautomat

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). Als Testnachricht wird Call('test, 'test) verwendet. Modul und Methodenname sind fast beliebig. Wichtig ist, dass ein Modulname und ein Methodenname angegeben ist. 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. Starten und neustarten erfolgt indem die Zugangsdaten des Nass an den HeartBeat gesendet werden. Zum verschicken der Testnachrichten verwendet der HeartBeatActor einen eigenen ResponseActor. Es ist ein eigener Responseactor um die Last auf dem normalen ResponseActor zu reduzieren.

Prinzipiell kann der HeartBeat auch direkt einen ØMQ Socket nutzen. Dies bedeutet, dass das after Pattern direkt auf ein ØMQ Socket angewendet wird. Ein ØMQ Socket wird von ØMQ auf deren native Bibliothek abgebildet. Im Test gab es allerdings Fehler aus der nativen Bibliothek heraus, sobald das Timeout erreicht wurde. Kapselt man die Kommunikation in einem Aktor, tritt der Fehler nicht auf. Daher wird auf einen eignenen ResponseActor gesetzt.

Der Zustandsübergang des HeartBeatActors erfolgt über eine einzelne Methode step. Der Methode step wird für sowohl für den positiven als auch für den negativen Fall der Nachfolgezustand (als partielle Funktion) und eine dazu passende Nachricht übergeben. Die Methode step schickt daraufhin mit der Methode ask die Testnachricht an den ResponseActor. Dabei wird implizit ein Timeout angegeben.

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

Wird eine Nachricht innerhalb des Timeouts vom ResponseActor zurückgeschickt, wird onSuccess ausgeführt. Darin auf eine beliebige Antwort gematched, denn die Antwort ist irrelevant. Als erstes wird die positive Nachricht an den caller geschickt. Der caller ist standardmäßig der NaoActor. Danach wird der Timer aktiviert mit einem Timeout, dass in der Konfiguration für den positiven Fall (on) definiert ist. Das Timeout reduziert die Anzahl an Testnachrichten, da sonst eine unnötige Last erzeugt wird. Am Ende wird mit become in den neuen Zustand übergegangen. Der Nagativfall onFailure verläuft analog zum Positivfall. Nur wird die nagative Nachricht an den NaoActor geschickt wird, die Verzögerung off dem Timer übergeben wird und in den Nachfolgezustand für den negativen Fall übergegangen.

```
1  answering onSuccess {
2      case _ => {
3        caller ! succMessage
4        delay(on)
5        become(suc)
6      }
7  }
8  answering onFailure {
9      case _ => {
10       caller ! failMessage
11       delay(off)
12       become(fail)
13     }
14   }
```

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

Durch den Neustart geht der NaoActor in den Startzustand über. In diesem Zustand wartet der NaoActor auf HeartBeatActor. HeartBeatActor hat vom NaoActor die Zugangsdaten neu erhalten und versucht wiederum Testnachrichten zu schicken. Ist dies möglich, bekommt der NaoActor die online Nachricht und kann nun wieder in den Zustand communicating übergehen. Zustände die Maybeoffline sind für diesen Ablauf unerheblich und können daher gespart werden. Damit wird auch die Last für den NaoActor reduziert.

4.7 Strukturierung auf Dateisystemebene

Der naogateway ist in seiner Ordnerstruktur nach folgendem möglichst einfachem Schema aufgebaut:

- README.md - Anleitung zum Installieren in einer Ubuntu VM, Erklärung der Kompilierung, Startparameter und Konfigurationsbeispiele
- naogateway.pdf - Dokumentation: Realisierung einer distributiven Steuerung
- communication - Ordner mit Sequenzdiagrammen für die Kommunikation und Heartbeat Zustandsmaschine
- scaladoc - Generiertes Scaladoc, dass ähnlich dem Javadoc Klassen- und Methoden dokumentiert

- hosted.conf - Konfigurationsbeispiel um mit naogateway, der in der Virtual Box VM läuft vom Host aus zu kommunizieren
- build.sbt - Konfiguration des Simple Build Tools für die Kompilierung des naogateway
- src/main/ressources/application.conf - naogateway Standardkonfiguration
- src/main/scala - Quellcode
  - naogateway - Aktoren und die Mainclass des Naogateways
    - traits - Scala traits für Aktoren (z.B. Anbindung von zmq)
  - naogatewayValue - Alle nötigen Nachrichten
  - zeromq - jzmqadapter zum Ansprechen der Bindung
  - test - Beispiele um einzelne Aktoren zu testen
    - simple - lokale Tests für einzelne Aktoren im naogatway
    - remote - Test for die Kommunikation zwischen zwei JVMs
    - timer - Test für ein Timeout
    - directZMQ - Tests für die Kommunikation mit dem Nao ohne Aktoren

Da SBT (vgl. Typesafe Simple Build Tool, 2013) zum kompilieren verwendet wird, ist ein Teil der Ordnerstruktur vorgegeben. Der Quellcode muss sich in src/main/scala oder src/main/java befinden. Der Ordner src/test ist für Testcode gedacht. Allerdings hatte ich beim Testen massive Schwierigkeiten mit dem Auflösen der Imports aus dem src/main Ordner, sodass ich den Testcode in src/main/scala gespeichert habe. SBT ist das Standardtool für Scalaentwickler, da es eine sehr umfangreiche Konfigurationsmöglichkeit liefert um Java und Scala zu kompilieren, in jars zu packen oder ein Scaladoc zu generieren. Außerdem integriert es auf einfachste Weise Maven repositories, die es einem erlauben Abhängigkeiten dynamisch nachzuladen. Wichtig ist dafür die Konfigurationsdatei sbt.build. In dieser muss der Projektename, eine Version, der Scalacompiler und die mainClass definiert werden. Außerdem können die Abhängigkeiten,

sofern vorhanden, definiert werden. Die Abhängigkeiten der Abhängigkeiten werden durch Maven automatisch aufgelöst.

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

Listing 4.1: build.sbt - SBT Konfigurationsdatei

Die Syntax besteht zunächst aus Konstantendefinitonen nach dem Schema $name := value$, Verarbeitet wird diese Datei von einem Scalacompiler, sodass man den gesamten Sprachumfang von Scala zur Verfügung hat. Die Konstantendefinitonen sind in einer einfachen DSL definiert und werden umgeformt und validem Scalacode. Wenn SBT auf einem Rechner installiert ist, genügt im Projektverzeichnis ein einziger Befehl (s. Formel 4.5) zum Kompilieren und Starten. Statt dem Befehl run kann auch compile zum reinen compilieren und doc zum generieren des Scaladocs verwendet werden.

$$sbt\ run \tag{4.5}$$

Zur Kommunikation mit dem naogateway ist nur naogatewayValue als Abhängigkeit nötig, denn dort sind alle Nachrichten zur Kommunikation mit dem naogateway gespeichert. Ohne gleich den gesamten naogateway einbinden zu müssen, sind die Nachrichten in einem anderen Ordner. D.h. es gibt mehrere Ordner für den Quellcode. Erst darin ist die Packagehierachie abgebildet. In der Packagehierachie ist naogatewayValue das Subpackage value von des Packages naogateway.

5 Fazit und Ausblick

Im Nachhinein sind mir folgende Verbesserungsmöglichkeiten aufgefallen. Ein wichtiger Punkt sind die verwendeten Bibliotheken. Eingesetzt werden die Java Bindings für jzmq, da diese schon länger verfügbar sind. Die ersten Comitts der Scala Bindungs sind weniger als ein Jahr alt. Langfristig sind die Scala Bindings die klügere Wahl, da die restliche Applikation in der gleichen Sprache geschrieben ist und zusätzlich die Bindings sehr sehr kompakt gehalten sind. Es gibt, ähnlich wie bei jzmq, auch eine Scalavariante für Protobuf, die noch sehr jung ist und in unserem Test nicht funktioniert hat, daher sind wir bei der Javavariante geblieben. Auch hier macht es Sinn weiter zu testen, denn die Scalavariante ist nicht nur sehr kompakt, sondern auch in der Lage Besonderheiten von Scala zu verwenden, wie beispielsweise implizite Methoden.

Der HAWAktor und auch der HAWCamServer entstanden unabhängig von mir, damit musste ich so wie auf dem Nao installiert sind, umgehen. Das Request-reply Pattern, dass der HAWCamServer und des HAWCamServer erfordert, macht die Kommunikation an manchen Stellen umständlich, denn es gibt nur einzelne Bilder, die einzeln angefordert werden müssen. Das einzelne Anfordern von Bildern, die 20-30 mal pro Sekunde über das Netzwerk geschickt werden sollen, ist sehr teuer. An dieser Stelle könnte optimiert werden, indem auf man andere Socket-types wie Router in ZMQ verwendet. Damit kann das Request-reply Pattern aufgebrochen werden.

Ist eine zentrale Instanz des Naogateway 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 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 (vgl. Charousset, Dominik, 2013) sind vorhanden und sind das Ziel für die Zukunft. Der Einsatz von libcppa scheitert derzeit am C++ 11 Standard,

für den die Compiler in qibuild nicht verfügbar sind. Die Bibliothek libcppa kann ohne C++ 11 Standard nicht kompiliert werden. Ohne Aktor auf dem Nao muss sichergestellt sein, dass einem Nao genau ein naogateway zugeordnet ist. Umgekehrt gibt es keine Einschränkungen.

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

Die Identifizierung des Calls erwies sich als funktionsfähig, jedoch relativ unpraktikabel, da der Code unübersichtlich und unnötig lang wird. Jede Referenz des Call muss man sich merken und bei vielen Calls muss man diese Calls gegebenenfalls über eine Map $String \rightarrow Call$ speichern. Außerdem muss diese bei jedem Zustandswechsel mitgeführt werden. Die Identifizierung ließe sich vereinfachen, indem man dem Call ein beliebiges, serialisierbares Token übergeben würde. Das Token wird in der Answer statt dem Call gespeichert. Dann ist es möglich im Patternmatching das Token anzugeben (s. Formel 4.4). Da Answer in Scala als case class realisiert ist, kann auch auf das innere das Patternmachting angewendet werden. Zusätzlich könnten mehrere Calls mit gleichem Inhalt unterschieden werden. Wenn man ein kleines Token wählt ist das Objekt deutlich kleiner und lässt sich schneller serialisieren, deserialisieren und verschicken.

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 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 noch viele Fragen offen 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. Wie müssen sich die untereinander Aktoren untereinander absprechen? 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.

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

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.

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

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

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

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

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

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

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

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

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

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