# 4 Realisierung einer distributiven Steuerung

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

## 4.1 Vorstellung des Naoqi SDK

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

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

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

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

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

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

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

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

Auch einfache Befehle 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. Um nebenläufige Prozesse mit den RPCs abzubilden, wird eine Auslagerung in z.B. Futures benötigt.

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

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

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

## 4.2 Anwendung: Steuerung durch Smartphone und Spielecontroller

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

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

Das Smartphone kann Videos darstellen und damit auch das Videobild des Naos. Mit dem Videobild kann der Nao gesteuert werden, ohne das ein direkter Sichtkontakt besteht. Allerdings

ist dazu auch eine Bewegung des Kopfs nach rechts, links, oben und unten nötig. Wird der Kopf bewegt, verändert sich auf der Blickwinkel der Kamera im Kopf des Nao. Damit kann man sich in der Umgebung des Nao besser orientieren.

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

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

## 4.3 Kommunikation mit dem Nao

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

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

Die Naoqi API ist nach folgender Struktur aufgebaut: Es gibt Modulnamen wie z.B. ALTextToSpeech und Methodennamen wie z.B. getVolume (s. Formel 4.1). Diese beiden Informationen genügen, um die Methode über den Broker aufzurufen. Daraus könnte abgeleitet werden, eine statische Abbildung $Message \rightarrow Method$ mit einem Dispatcher (switch case) zu realisieren. Es hätten allein für das Modul ALMotion über 80 Methoden mit einem Dispatch versehen

werden müssen. Da die API nach einem einheitlichen Schema aufgebaut ist, ist ein reflektiver Ansatz naheliegend. Reflektiv bedeutet, dass die Methoden in diesem Fall über zwei Strings übertragen werden und zur Laufzeit zu einem Methodenaufruf zusammengesetzt und aufgerufen werden (vgl. Colouris et al., 2012, S. 55).

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

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

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

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

Der Rückgabewert wurde bisher noch nicht angesprochen. Viele Methoden, insbesondere im Bereich Bewegung, enthalten gar keinen Rückgabewert. Wenn jedoch ein Rückgabewert vorhanden ist, dann ist der Typ, wie bei den Parametern, ein Basistyp. Für z.B. ALMotion.getCOM ist der Rückgabewert ein Array von Floats.

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

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

Codebeispiel 4.2: Request-Nachricht für den HAWAktor

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

```
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;
}
```

Codebeispiel 4.3: „überladener Datentyp“ für Parameter und Rückgabewerte

Die Antwort enthält im positiven Fall genau einen Rückgabewert des Typs MixedValue (s. Codebeispiel 4.4). Ein Fehler kann in Form eines Strings angegeben werden. Sollte es eine Methode nicht geben, wird von einer Fehlernachricht Gebrauch gemacht. Der Rückgabewert ist dann leer.

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

Codebeispiel 4.4: Response-Nachricht für den HAWAktor

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

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

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

Codebeispiel 4.5: Zugriff auf ØMQ Sockets

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

```
  def request(module: String, method: String,
        params: List[MixedValue] = Nil) {
    val param = HAWActorRPCRequest.newBuilder
        .setModule(module).setMethod(method)
```

```
5     for (mixed <- params) {
6       param.addParams(mixed)
7     }
8     param.build
9   }
```

Codebeispiel 4.6: Erstellung eines Requests für den HAWAktor

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

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

Codebeispiel 4.7: Implizite Konvertierung in MixedValue

Da die Methode anyToMixedVal (s. Codebeispiel 4.7) eine implizite Methode ist, muss diese nicht direkt aufgerufen werden, sondern wird vom Compiler aufgerufen, wenn eine Typkonvertierung nötig ist. Für Arrays ist auf die gleiche Weise eine implizite Methode definiert. Damit

ist es möglich die Methode request mit ausschließlichen Scalatypen aufzurufen (s. Formel 4.3). Die Parameterliste wird dabei implizit von List[Any] zu List[MixedValue] konvertiert.

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

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

Die Nachricht, wie HAWActorRPCRequest, hat eine sehr geringe Größe, daher kann sie in einem Frame verschickt werden (s. Formel 4.4). Als Endmarkierung der Nachricht wird daher die 0 benötigt. Somit genügt es dem Socket das Bytearray von Nachricht req zu übergeben und es wird an die angegeben URL versendet.

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

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

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

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

```
1  def answer = {
2    val resp = HAWActorRPCResponse.parseFrom(socket.recv(0))
3    if (resp.hasError) {
```

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

Codebeispiel 4.8: Überführung eines Bytearrays in eine Response Nachricht

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

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

Codebeispiel 4.9: Request-Nachricht für den CamServer

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

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

Codebeispiel 4.10: Response-Nachricht für den CamServer

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

## 4.4 Kommunikationssequenzen im Aktorensystem

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

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

Erschafft und überwacht werden diese Aktoren vom NaoActor. Es existiert der HeartBeatActor, um zu überprüfen, ob der Nao zur Zeit erreichbar ist. In einem fest definierten Intervall werden vom HeartBeatActor Testnachrichten geschickt, die den einzigen Zweck haben eine Antwort zu generieren. Ist eine Antwort gekommen, ist sichergestellt, dass der Nao in diesem Moment erreichbar ist. Der NaoActor erhält vom HeartBeatActor Nachrichten, wenn sich der Verbindungsstatus ändert.

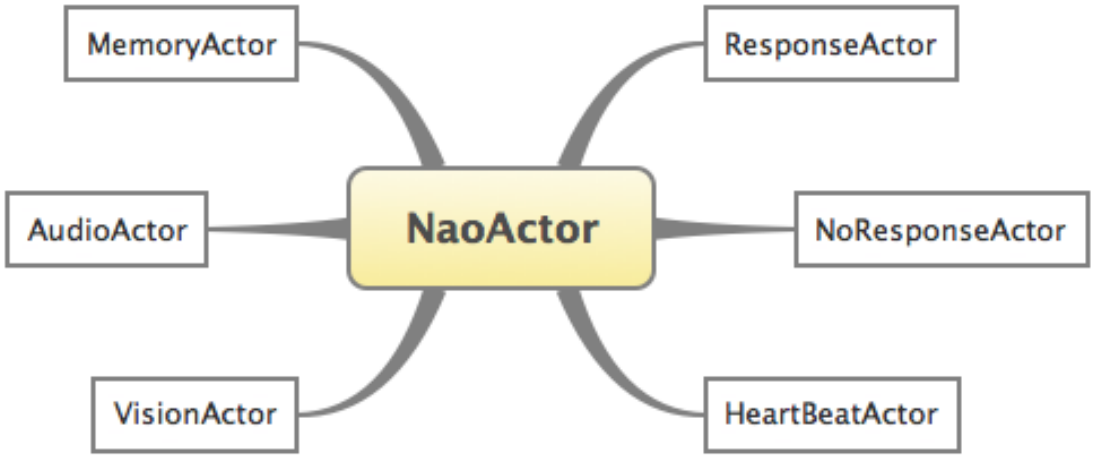


Abbildung 4.1: Aktorensystem naogateway

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

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

Codebeispiel 4.11: Zugangsdaten des Naos

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

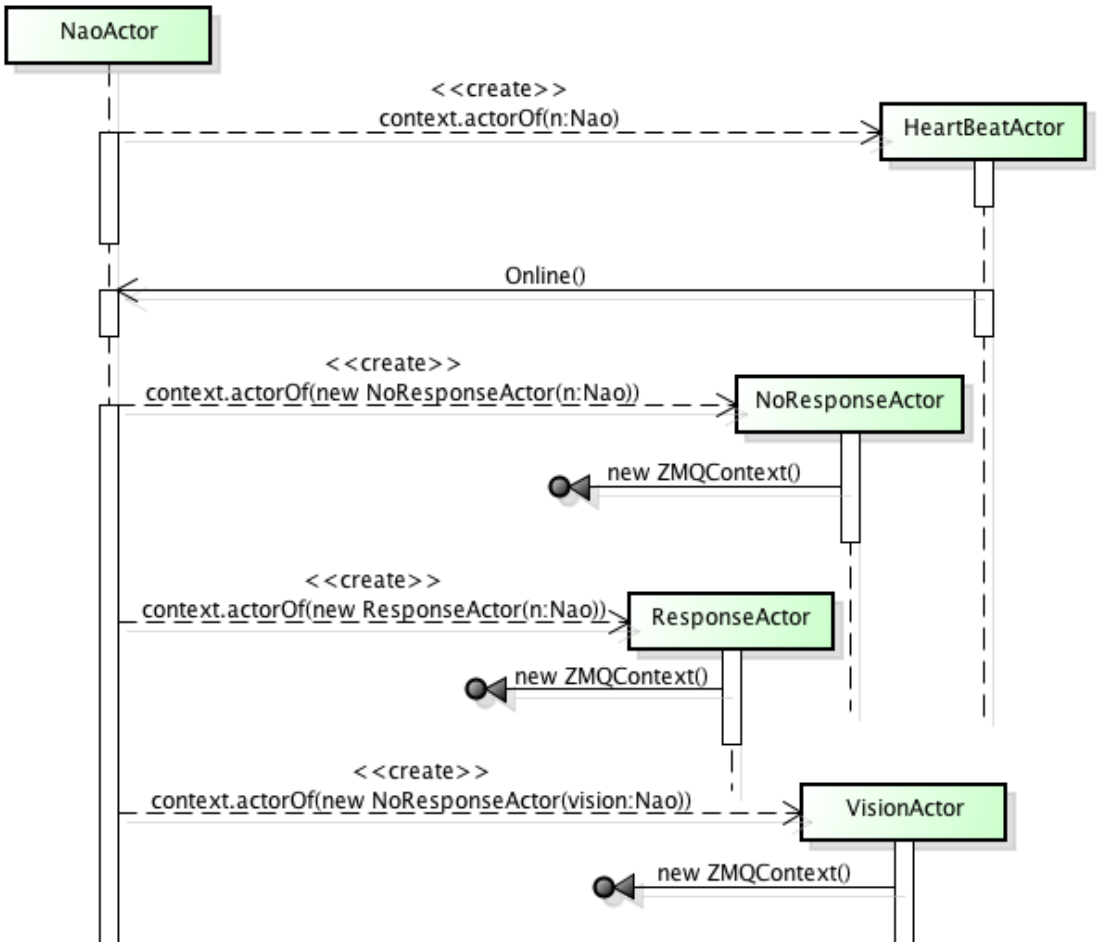


Abbildung 4.2: Initialisierung

Ist der NaoActor mit seinen Kindern initialisiert, kann dem NaoActor eine Connect Nachricht geschickt werden, um die Akorreferenzen seiner Kinder zu erhalten. Der NaoActor antwortet mit dem Tripel (reponseActorRef, noResponseActorRef, visionActorRef). Das Tripel enthält die Aktorreferenzen von dem NoResponseActor, dem ResponseActor und dem VisionActor; diese können verwendet werden, um an einen ausgewählten Aktor einen Request zu schicken. Die drei Aktoren kann man als Worker auffassen, die auf Anfrage die Kommunikation zwischen einem beliebigen Aktor und dem Nao realisieren.

Die Anfrage an den ResponseActor und den NoResponseActor ist die Nachricht Call (s. Formel 4.6). Call kapselt die Protobuf-Nachricht HAWActorRPCRequest. Der Call enthält einen Modul-

namen, einen Methodennamen und eine Liste von Parametern als MixedValue. In den beiden Aktoren wird der Call in die Protobuf-Nachricht umgwandelt und an den Nao geschickt.

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

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

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

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

Die erste Idee, war einen eigenen Aktor zu verwenden, der nur Nachrichten an den Nao schickt und empfängt. Dieser Aktor wird für jeden Call instanziiert und, nachdem die Antwort weitergeleitet wurde, wieder geschlossen. Das Blockieren wird dann in diesen neuen Aktor

delegiert. Man hätte dazu einen Pool von Aktoren anlegen müssen um die Zahl der Aktoren unter Kontrolle zu halten.

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

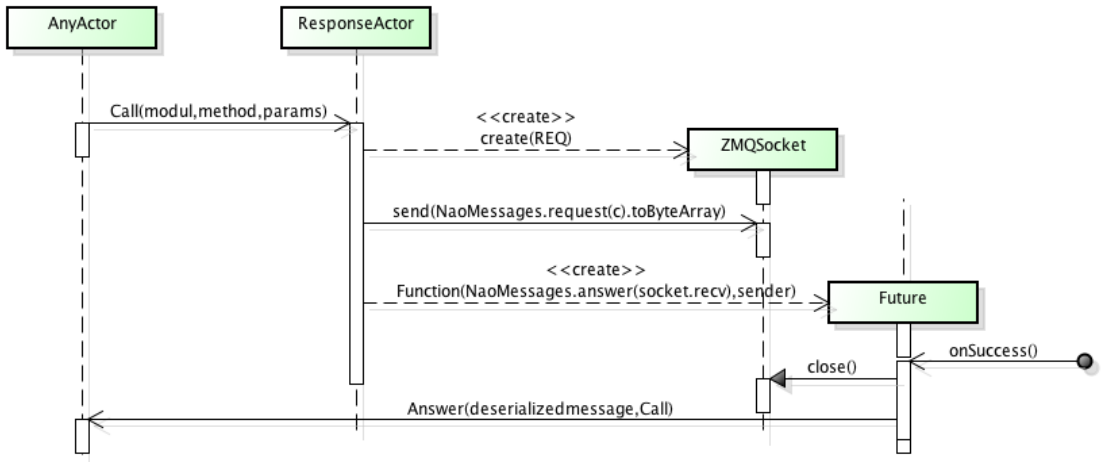


Abbildung 4.3: Request-reply Kommunikation mit dem ResponseActor

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

Es stellt sich an dieser Stelle die Frage, wie der NoResponseActor mit ankommenden Calls verfährt, während eine Antwort auf den vorherigen Call noch aussteht. Eine erste Lösung war folgende: Der Aktor verfügt über eine Mailbox, aus der der Aktor nicht nur Nachricht herausnehmen kann, sondern diese auch nach dem Herausnehmen zur Seite gestellt werden können. Dies wird Stashing genannt. Die Nachrichten können nicht direkt in die Warteschlange gelegt werden, da sonst eine Endlosschleife für Stashing entstehen würde.

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

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

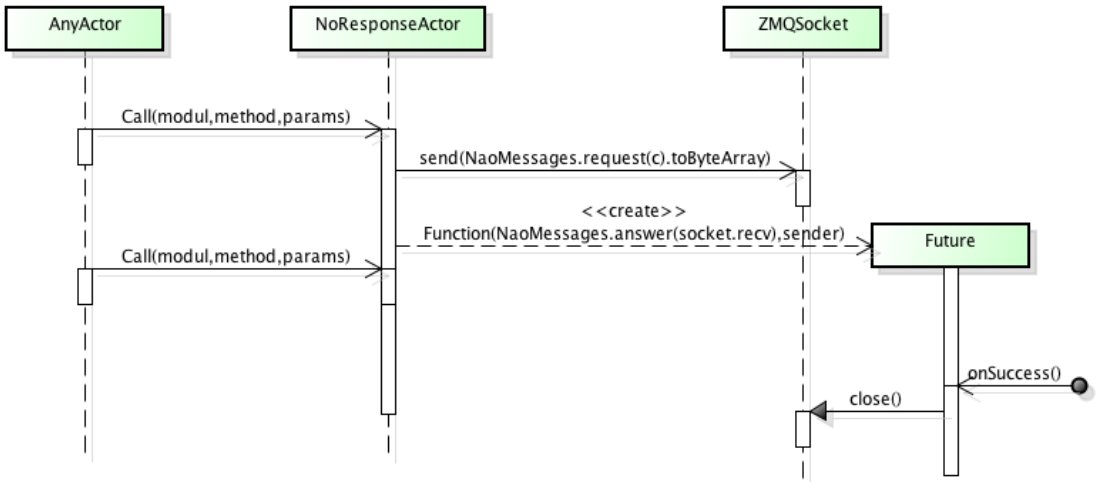


Abbildung 4.4: Vorgetäuschte unidirektionale Kommunikation

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

Farbräume (s. Codebeispiel 4.12) und 1-30 Frames pro Sekunde unterstützt. Die Bezeichnungen wurde von der Naoqi API übernommen.

```
object ColorSpaces extends Enumeration {
  type ColorSpace = Value
  val kYuv = Value(0)
  val kYUV422 = Value(9)
  val kYUV = Value(10)
  val kRGB = Value(11)
  val kHSY = Value(12)
  val kBGR = Value(13)
}
```

Codebeispiel 4.12: Farbbereiche des HAWCamServers

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

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

Codebeispiel 4.13: Nachricht VisionCall für die Anfrage an den VisionActor

Der HeartBeatActor soll in einem bestimmten Intervall erneute Testnachrichten schicken, um den Nao nicht unnötig zu belasten. Dafür wird auch ein Timer benötigt. Es sollen auch der ResponseActor und der NoResponseActor in der Lage sein eine Verzögerung für bestimmte Nachrichten einzuleiten. Beispielsweise ist es nicht sinnvoll, innerhalb von wenigen Millisekunden den Arm nach oben und wieder nach unten zu bewegen. Der Nao kann Bewegungen in der gegebenen Zeit nicht ausführen. Auch hierfür ist ein Timer nötig, um eine Verzögerung zu realisieren.

Man hätte diese Timer auch in die Aktoren verlagern können, die den ResponseActor oder NoResponseActor benutzten. Wenn in einer neuen Version des Naoqis die Verzögerung unter-

stützt wird, wäre die Anpassung sehr aufwändig geworden. So müssen nur der ResponseActor und NoResponseActor angepasst werden.

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

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

Codebeispiel 4.14: Verwendung des after pattern zum zerzögernden Ausführen

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

```
1  responseactor.delay {
2        almotion.setposition = 200
3  }
```

Codebeispiel 4.15: Konfigurationsbeispiel für Verzögerungen

Eine Methode delay, der Methodenname und Modulname übergeben werden, sucht die dazu passende Konfiguration und gibt die Verzögerung in Millisekunden zurück. Sollte eine Methode

in der Konfiguration nicht definiert sein, wird eine Verzögerung von 0 zurückgegeben. Der Request wird durch das Pattern after erst nach der Verzögerung an den Nao geschickt.

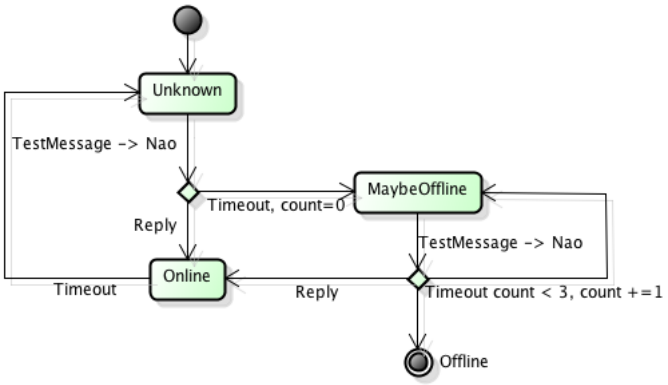


Abbildung 4.5: Heartbeat Zustandsautomat

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

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

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

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

Codebeispiel 4.16: Message Passing mit Timeout

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

```
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   }
```

Codebeispiel 4.17: Reaktion auf Timeout des Futures

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

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

## 4.5 Verteilung

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

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

Im Rahmen des Projekts ist eine AndroidApp, die das Kamerabild des Naos darstellt und eine Bewegungsteuerng ermöglicht, entstanden. Es ist eine Desktop Applikation zur Steuerung mit

der Tastatur oder dem XBOX Controller entwickelt worden. Außerdem ein Watchdog, der Statusinformationen des Naos, wie Batteriezustand, überwacht. Die Idee ist es, dass alle Applikationen das Aktorensystem RobotMiddleware verwenden. RobotMiddlware stellt unabhängig vom Nao Operationen wie Laufen bereit, die intern in Call Nachrichten ungewandelt werden um mit dem naogateway zu kommunizieren.

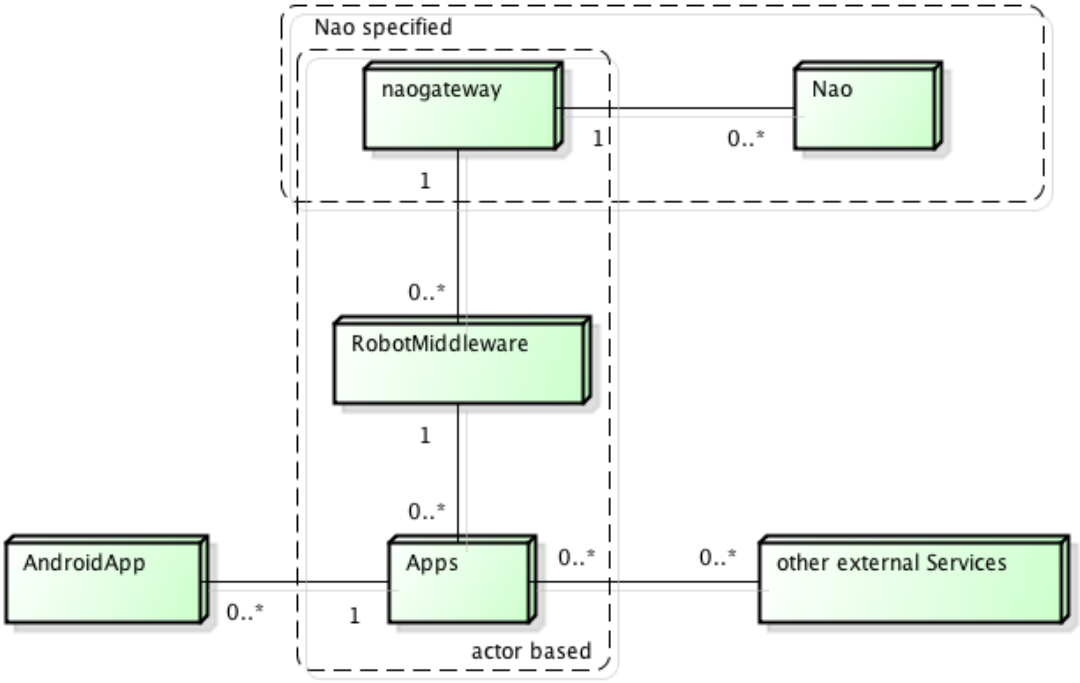


Abbildung 4.6: Verteilungssicht

Applikationen sind unabhängig voneinander 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.

Für die Kommunikationsmechanismen ist ein etwas tieferer Blick in die einzelnen Komponenten nötig (Abb. 4.7 Kommunikationsschema zwischen den Aktorensystemen). Im Grundsatz gibt es vier Bereiche, die miteinander kommunizieren müssen.

Die RobotMiddleware bietet ein trait RobotInterface, welches die hochsprachlichen Methoden wie Laufen bereitstellt und beim Aufruf die Methoden in Nachrichten umsetzt, die über den eigenen Aktor NaoTranslator an naogateway delegiert werden, dort in Naoqi spezifische Auf-

rufe umgewandelt werden und zu Nao geschickt werden. Die Antwort wird über naogateway wieder zum NaoTranslator zurückgeleitet.

Applikationen agieren in einem eigenen Aktorensystem, um völlig unabhängig zu sein. Die Apps haben die Möglichkeit, auf die NaoInterface API zuzugreifen und gleichzeitig externe Dienste zu nutzen, um eine vollwertige App zu schaffen, die ein Nao steuern kann. Die AndroidApp benötigt eine Serverapplikation, da sie selber nicht Teil des Aktorensystems ist. Die AndroidApp kommuniziert ausschließlich über einen ØMQ Socket. Die Idee ist, Nachrichten vollständig über Protobuf abzubilden und dabei von speziellen Roboterschnittstellen, wie Naoqi, zu abstrahieren. Akka unterstützt nativ Protobuf als Serialisierer, sodass sich dieses sich ins System einbinden lässt. Anzumerken ist, dass bisher nur die Kommunikation zwischen dem naogateway und dem Nao über Protobuf erfolgt. Die AndroidApp kommuniziert derzeit direkt über ØMQ mit dem Nao. In der Abbildung 4.7 sind die noch nicht umgesetzten Bereiche rot markiert.

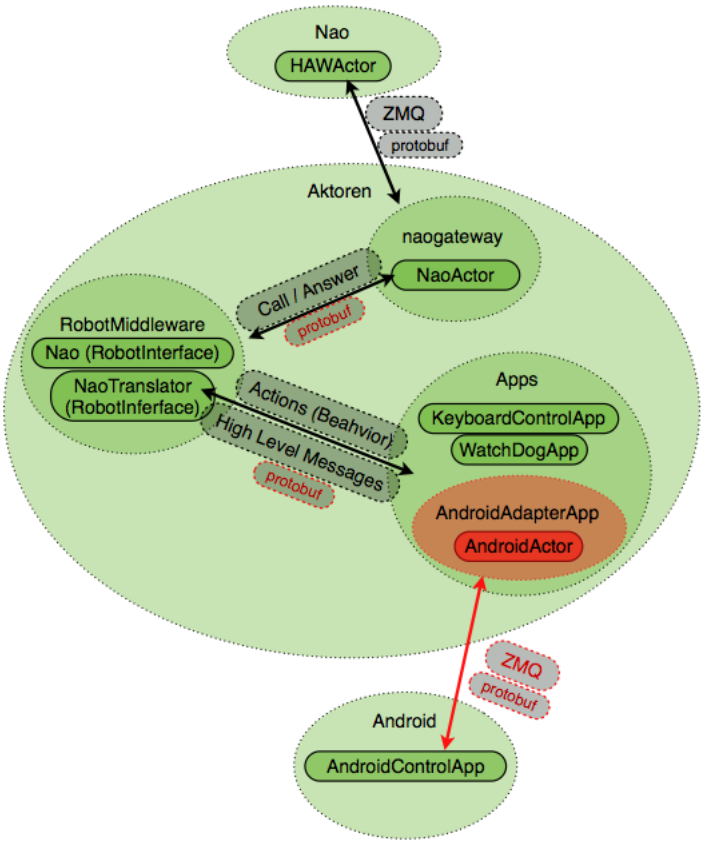


Abbildung 4.7: Kommunikationsschema zwischen den Aktorensystemen

## 4.6 Strukturierung auf Dateisystemebene

Der naogateway ist in seiner Ordnerstruktur wie folgt aufgebaut:

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

    * naogateway/value - Die Nachrichten sind im Package naogateway.value organisiert.
  - zeromq - Der jzmq Adapter zum Ansprechen der Bindings wird als Quelltext ausgeliefert und ist im Quellcode-Ordner zeromq abgelegt.

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

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

Codebeispiel 4.18: build.sbt - SBT Konfigurationsdatei

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

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

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