Konzept für

Bachelor Thesis

Realisierung einer verteilten Kommunikation mit einem humanoiden Roboter am Beispiel des NAO auf Basis einer funktionalen Schnittstelle namens scaleNao

Inhaltsverzeichnis

Einleitung

Computer jeglicher Art ermöglichen uns Probleme effektiver und effizienter zu lösen und gestalten dabei unseren Alltag mit. Mit Hilfe von immer kleinere, immer schnellerer, immer robusterer Hardware ist es möglich Computersysteme immer mehr als ubiquitären Systeme in den Alltag zu integrieren (vgl. Aritoni 2011, S. 222).

Ubiquitäre Systeme zeichen sich durch den kleinen Formfaktor, die Robustheit gegen Erschütterungen und durch den geringen Stromverbrauch als mobil aus. Sie können autark reagieren (Smart Spaces), sind nahezu unsichtbar, indem sie in alltägliche Gegenstände integriert werden und können lokal skalieren, indem sie sich individuell an vorhandene Ressourcen und Anfordrungen anpassen. Außerdem können sie mit anderen Geräten interagieren (Information Appliance). Die passende Hardware ist die Voraussetzung, jedoch die passende Software, die alles vereint, ist die Herausforderung. Die beste Software ist die, die man nicht bemerkt (vgl. Luck, 2009).

Im Alltag verwenden wir die verschiedensten, meist analogen Werkzeuge. Oft in Unterstützung mit sehr einfachen eingebetteten Systemen (Waschmaschine, Wecker). Diese Erfüllen jedoch nicht die Anforderung eines ubiquitären Systems. Insbesondere Information Appliance und Smart Spaces sind nicht umgesetzt.

Eine Besonderheit stellt das Handy (oder das Tablet) dar, da dabei schon viele Möglichkeiten gegeben sind. Insbesondere in Verbindung mit dem Cloudcomputing wird die Landschaft der heterogenen Systeme aufgebrochen, indem sie mobil sind und verschiedenste Ressourcen verbinden.

Das Cloudcomputing bietet nicht nur einen Datenspeicher, sondern viel mehr eine dezentrale Datenbank von Wissen. Die heute vorhandenen Roboter sind meist Spezialisten, die nicht dazulernen oder sich neues Wissen während des Betriebs beschaffen können. Diese bedeutet sie können nur in einer vordefinierten Umgebung vordefinierte Aufgaben erfüllen (vgl. Neto 2010, 1.1).

Der Begriff App hat schon den Alltag geprägt und sagt nichts anderes aus als Wissen für eine spezielle Aufgabe als Dienst zur Verfügung zu stellen. Diese Apps haben als Ziel meist eine konkrete Plattform. Hierbei sind insbesondere die Plattformen von Android und Apple zu nennen. Handys sind jedoch alles andere als unsichtbar und autark.

Eine neue Dimension erschließt sich bei einem selbständig bewegenden System, welches flexibel nutzbare Hardware bietet und gleichzeitig selbständig in der Lage ist umliegende Ressourcen zu nutzen, diese intelligent zu verbinden und dabei auf Wissen aus der Cloud zurückgreift um zu agieren. Das ist der moderne, humanoide Roboter.

Ein Roboter ist letztendlich ein eingebettetes System, welches in der Lage ist sich selbständig fortzubewegen, die Umwelt zu erfassen in beispielsweise Bild, Ton, Ortungsverfahren (z.B. Sonar) und Werkzeugen (bei humanoiden: Arme und Beine), um in die Welt einzugreifen. Letztendlich wird dieses System durch einen eigenes Controller mit eigenem Betriebsystem und eigenem Speicher gesteuert, was in der Lage ist all diese Funktionalität auf einer niedrigen Ebene - d.h. sehr niedrigen Abstraktionsstufe - zu steuern. Auf dieser niedrigen Ebene ist der Roboter ein autonomes System.

Insbesondere im Bereich Bewegung von Lasten wird der humanoide Roboter hinter dem Menschen zurückbleiben, da die Mobilität (Gewicht und Akkuleistung) sonst nicht mehr gegeben wäre.
Jedoch ist im Bereich Informationsverarbeitung der Roboter dem Menschen weit voraus und wird in Zukunft ein wichtigen Teil im Alltag des Menschen einnehmen. Dafür müssen jedoch u.a. noch folgende Anforderungen erfüllt werden:

Der Roboter ist ein Echtzeitsystem, d.h. er muss in einer für den Menschen nicht bemerkbaren Verzögerung, sicher reagieren können. Das heißt die interne Rechenkraft muss ausreichen, um sich selber in jeder möglichen Kombination von Einzelaktionen zu steuern. Um das Echtzeitsystem zu gewährleisten müssen alle weitere Funktionalitäten auf anderen Geräte ausgelagert werden, die beispielsweise durch ein Netzwerk zur Verfügung stehen.

Diese externen Geräte berechnen Probleme auf einer höheren Ebene, die unabhängig von der Hardware und damit auch unabhängig von einem speziellen Roboter sind. Ein bestimmter Bestand an Hardware wird vorausgesetzt. Wenn er dabei auf Ressourcen von außerhalb angewiesen ist - d.h. Awendungen auf höhrerer Ebene ausführen möchte - braucht dieser eine Möglichkeit möglichst portabel, performant und sicher zu kommunizieren. Portabilität ist wichtig, da die externen Ressourcen und der Roboter mit verschiedenen Sprachen und Abhängigkeiten arbeiten und diese keine Hürde beider Kommunikation darstellen dürfen (vgl. Weynes 2003, S. 44). Dies wird gewährleistet, indem die Serialisierung keine Sprachmittel genutzt, sondern nur ein reines Datenformat benutzt, was in jeder Sprache abbildbar ist. Dabei ist auf auf ein möglichst geringen Pool an Voraussetzungen zu achten.

Auf Grund der geringen Bandbreite ist die Performanz wichtig und wird gewährleistet indem die Serialisierung im Binärformat vorliegt. Gerne wird beispielsweise XML benutzt wird, welches für den Menschen besser lesbar ist, jedoch von Maschinen, um so schlechter lesbar und übertragbar ist (vgl. Sumaray 2012, 2.).

Sicherheit bzw. Fehlertoleranz wird gewährleistet, indem beispielsweise Prüfsummen und Rückmeldungen verschickt werden, sodass überprüft werden kann, ob eine Nachricht verschickt wurde und ob sie korrekt verarbeitet wurde.

Die HighLevel Anwendung, die abstrahiert von der Hardware abläuft, kann viel mehr (parallele) Ressourcen zur Verfügung haben, als ein eingebettetes System. Damit dies auch funktionieren kann benötigt man eine funktionale Sprache (und damit auch Schnittstelle zum Roboter) um die nötige Parallelisierung sicher und effektiv umzusetzen.

Ist diese Kommunikation nicht gegeben, kann ein Roboter nicht nicht selbständig in seine Umgebung integrieren und wird auf ein stupides laufendes Konstrukt von Sensoren und Motoren zurückgestuft.

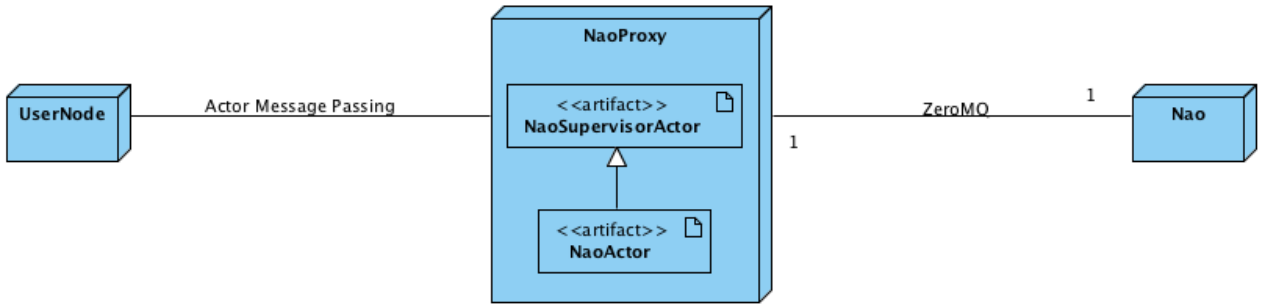
Diese Bachelor Thesis zielt darauf ab am Praxisbeispiel des Aldebaran Nao v4 eine Remotekommunikation zwischen einem netzwerkfähigem Rechner und dem Nao eine Kommunikation zu ermöglichen die die genannten Anforderungen an ein ubiquitäres System erfüllt und damit ein wichtiges Bindeglied zwischen Low und Highlevel zur Verfügung stellt.

Die Besonderheit besteht darin eine entfernte Kommunikation zu ermöglichen, die schnell und auf einer hohen Abstraktionsstufe ist. Dies ist derzeit durch eine C++/Python SOAP Schnittstelle nicht gegeben.

Es ist dabei nicht das Ziel den Roboter zu manipulieren, sondern ihn nur durch ein einziges Modul zu erweitern, dass die Kommunikation möglich ist. Außerdem soll die Netzwerkkommunikation und die Sicherstellung der Fehlertoleranz nicht neu entwickelt werden, sondern auf vorhandene Software zurückgegriffen werden. Ein Einbinden von einer Cloud ist ebenfalls nicht Teil dieser Arbeit. Eine Umsetzung der Schnittstelle des Roboters in einer funktionalen Sprache soll partiell erfolgen um die Funktionstüchtigkeit zu demonstrieren.

Verteilte Sicht

Ziel ist es dem Benutzer (Programmierer) eine entferne Schnittstelle zur Verfügung stellen, die sowohl fehlertolerant, schnell als auch angenehm zu benutzen ist.

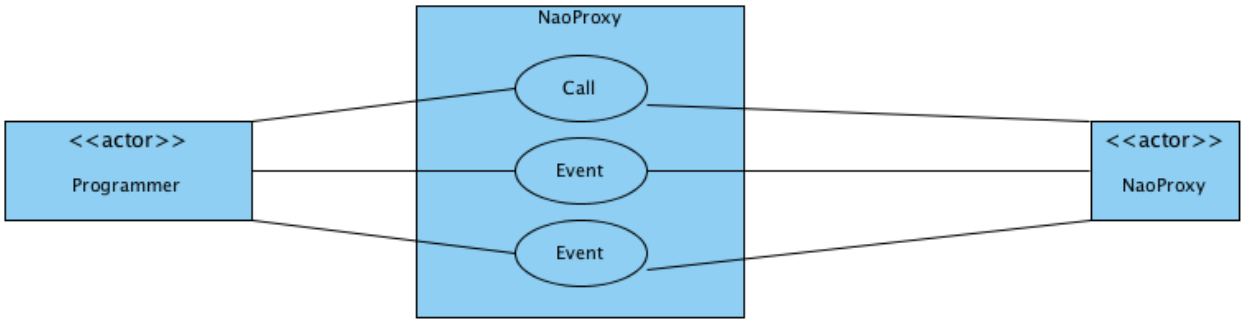


Die Idee ist, dass jeder Actor mit einem (vorerst zentralen) NaoActor kommunizieren kann und dieser n einer 1 zu 1 Beziehung mit dem Nao kommuniziert und eine eventuelle Antwort wieder an den UserActor zurückreicht. Das scaleNao das Akka Aktoren Konzept nutzt, kann Benutzer und scaleNao auf verschiedenen Nodes laufen. scaleNao nutzt zum kommunizieren ZeroMQ statt SOAP um schnell, jedoch auch fehlertollerant mit dem Nao zu kommunizieren. Diese Nachrichten werden von einem eigenen Naoqi Modul verstanden, welche die Abbiludng auf die native Naoqi API 1.12 bereitstellt. Hier nicht weiter behandelt.

Softwareanforderungen
- JVM mit Java 1.6
- Scala 2.10
- Akka 2.0

NaoActor

Der NaoActor kommuniziert direkt mit dem Nao, d.h. er baut eine Verbindung auf und vermittelt alle Nachrichten zum Nao und vom Nao. NaoActor ist damit die zentrale Einheit für die Interaktion zwischen Programmer und dem Nao.

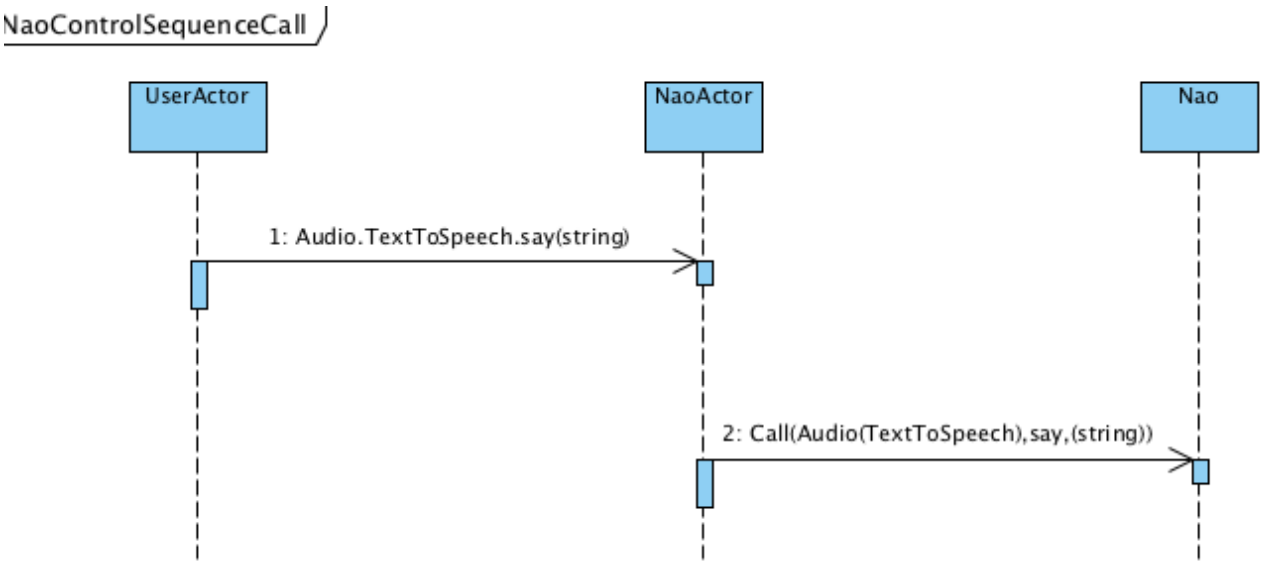


Dabei versteht er folgende DataMessages:
- InMessage
  - Answer
  - Event
- OutMessage
  - Call

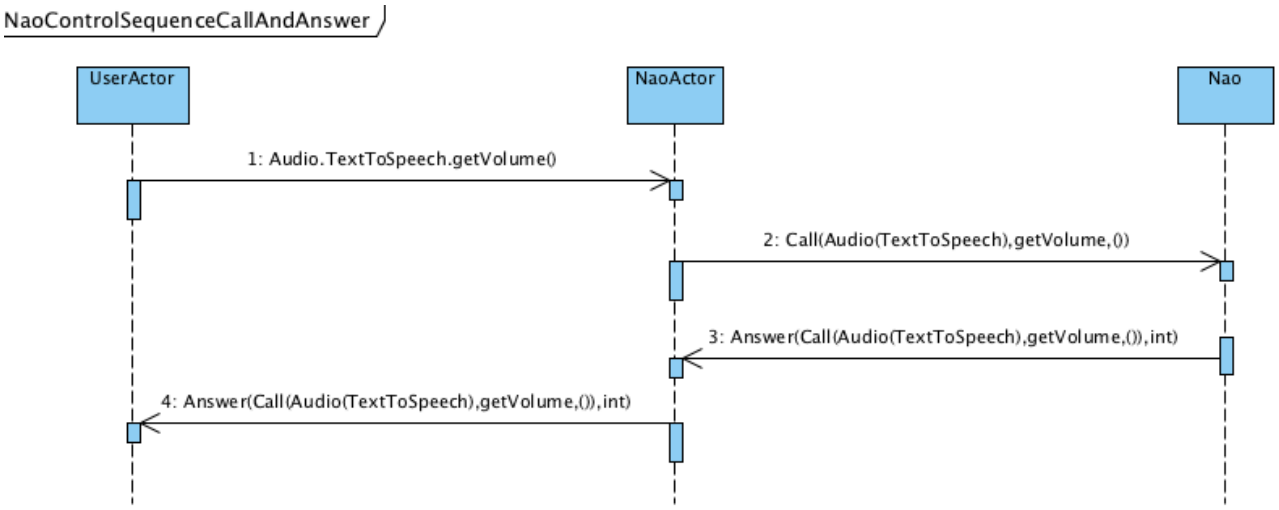
Zusätzlich versendet dieser ErrorMessages und InfoMessages an den Nao oder UserActor um auf technische Probleme oder technische Ereignisse hinzuweisen. Alles sind unter dem Typ Message zusammengefasst.

DataMessages
Datamessages Bündeln eine beliebiges Objekt (Any) mit einer Semantik. Scala Reflections zieht daraus die nötigen Daten wie Parameter und Rückgabetyp.

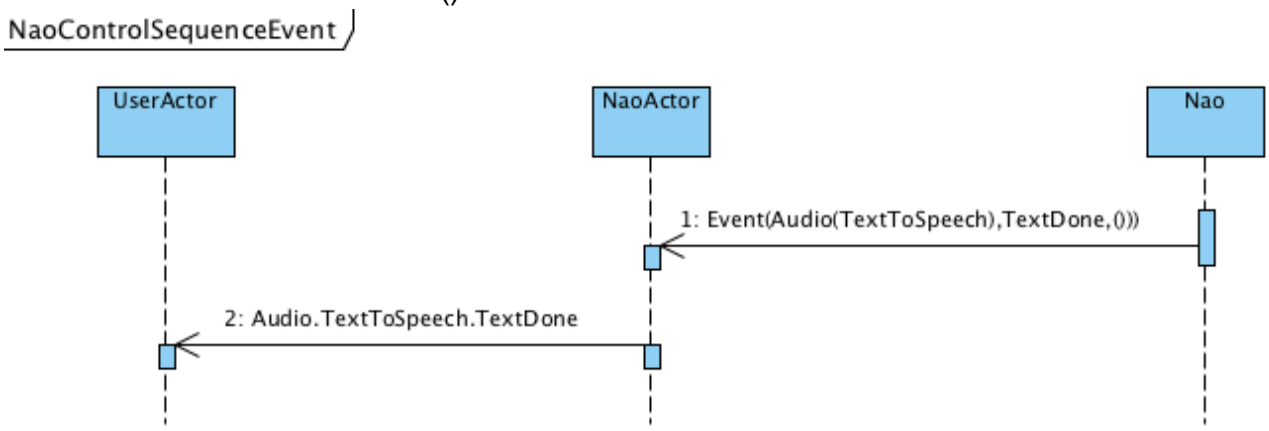
Call
Call enthält eine Methode vom Typ Method, deren nullstellige Funktionen deren Parameter sind und die die Funktion NaoValue answerValue() als Rückgabewert beinhalten müssen.



Answer
Enthält Call als Identfizierung und NaoValue answerValue()



Event
enthält NaoValue answerValue()



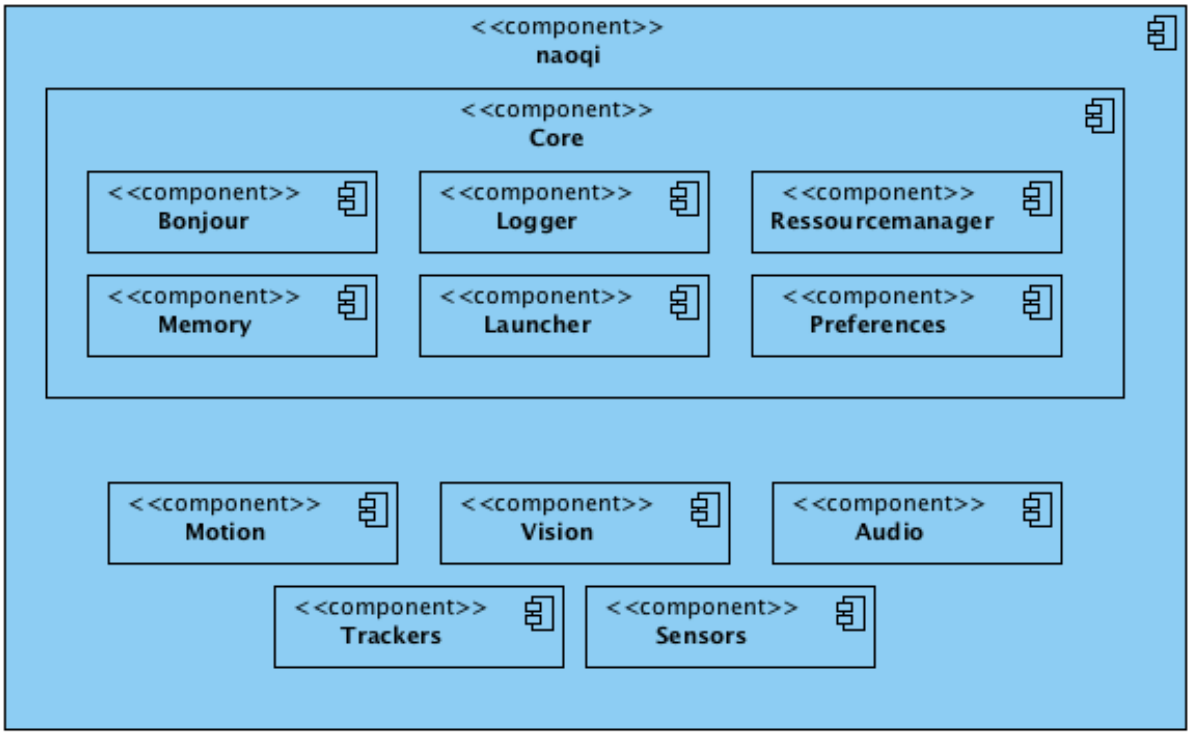
NaoValue
Typkonversation von Scalatypen nach Byte, die in folgenden Typen gekapselt sind und von C++ Modul im Nao verstanden werden müssen.

- Int32
- Float32
- NaoString
- NaoUnit

Verständnis der NaoActors
Der NaoActor ist als technische Realisierung zu verstehen, die zwar alle Funktionalität und Flexibilität bietet, diese jedoch in der Fachlichkeit keinen Beitrag liefert und somit wenig Comfort und damit auch Produktivität für den Programmmer bietet.

scaleNaoqi

scaleNao stellt dem Benutzer die Naoqi API nach außen auf eine ähnliche Weise wie die originale API dar. Es gibt Proxys, Modules und Methoden, deren Signatur vollständig kompatibel sind. Der entscheidende Unterschied ist, dass scaleNao die Bereitstellung von Proxys und Modules verwaltet. D.h. der Benutzer erhält eine Schnittstelle, die aussieht, als wäre alles initialisiert, in dem dieser scaleNao startet. scaleNao verwaltet selbständig die Verbindung zum Nao und geht dabei von einer sicheren Umgebung aus, die zwar abstürzen kann und in der es mehrere Naos geben kann, jedoch in der es keine konkurrierenden entfernten Zugriffe auf einen Nao gibt.



Beispiel in python mit SOAP

```
import sys
from naoqi import ALProxy
if (len(sys.argv) < 2):
    print "Usage: 'python texttospeech_say.py IP [PORT]'"
    sys.exit(1)
IP = sys.argv[1]
PORT = 9559
if (len(sys.argv) > 2):
    PORT = sys.argv[2]
try:
    tts = ALProxy("ALTextToSpeech", IP, PORT)
except Exception,e:
    print "Could not create proxy to ALTextToSpeech"
    print "Error was: ",e
    sys.exit(1)
tts.say("This is a sample text!")
```

Beispiel scaleNao mit ZeroMQ

```
object MyFirstNaoTest extends Actor {
  override def act = {
    import scaleNao.raw.NaoActor.Nao
    val nao = Nao("Nila", "localhost", 9001)
    import scaleNao.raw.NaoActor
    NaoActor.start
    NaoActor ! nao
    import scaleNao.raw.messages._
    import scaleNao.api._
    NaoActor ! Call(Audio.TextToSpeech.say("abc"))
  }
}
```

Offene Fragen

- Verfügbarkeit durch Hypervisor
- Sichtbarkeiten: Raw Request für alle zulassen?
- Hinzufügen eines Moduls: überhaupt nötig?
- In wie weit müssen und dürfen zusätzliche Module Besonderheiten aufweisen (wenn man auf raw request verzichten möchte).