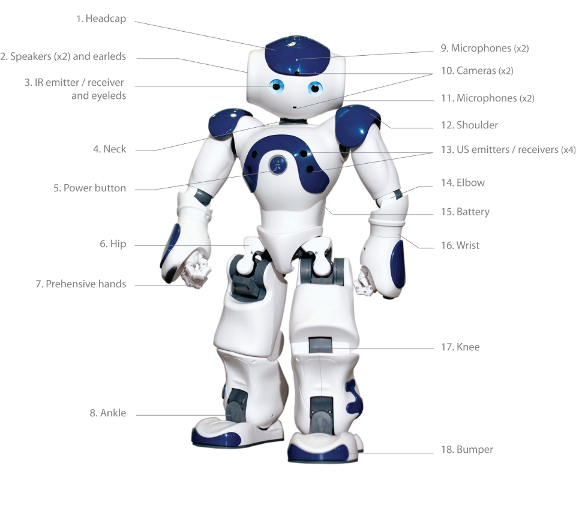
Nao Experiments



Author: Rene Bakx

Date: December 2015

Contents

1 Introduction 3

2 Nao architecture 3

2.1 Brokers and proxies 4

2.2 Device Communication Manager (DCM) 5

2.3 Language support 5

3 Ways to work with the Nao 6

3.1 With Choregraphe only 6

3.2 On the Nao using a SSH connection 6

3.3 Choregraphe Python script versus standalone Python script 7

3.4 On a laptop using the SDK 8

3.5 Simulation 8

# Introduction

This document describes the experiments with the Nao robot. Purpose of these experiments is to get to know the Nao robot and its NAOqi framework. The resulting documents (including this one) and code is available at <https://github.com/rbakx/Nao/tree/master/NaoExperiments>.

Summary of the experiments:

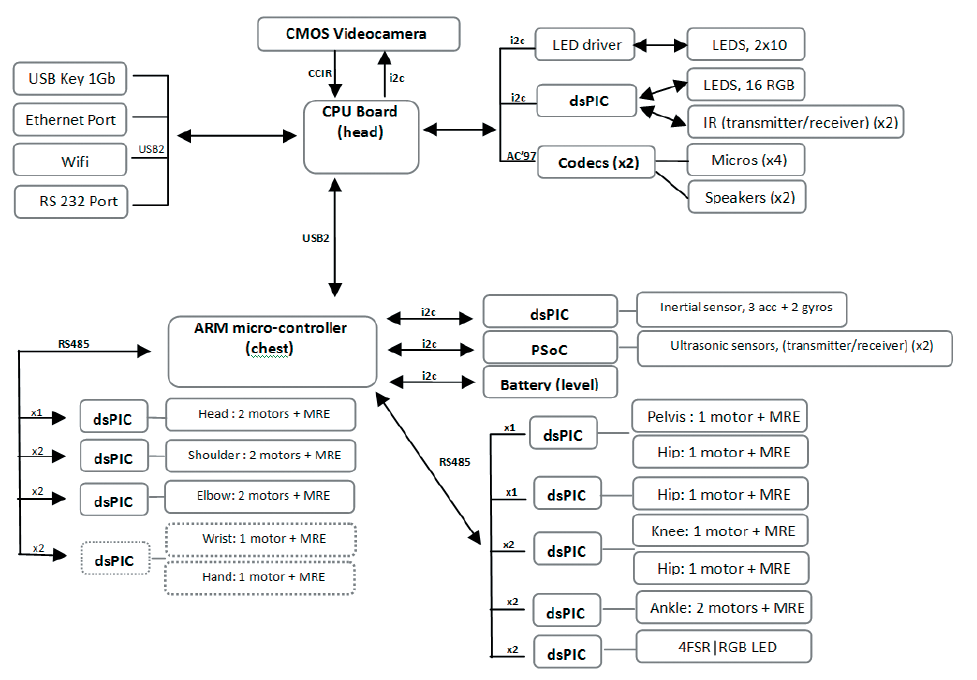
* Investigation of the different ways of working with Nao.
* As a project a behavior was created which:
  + Recognizes and greets persons.
  + When a person is not recognized within a certain time the Nao will ask for the name and learns the new face.
  + Two types of recognitions are possible:
    - With the Nao speech recognizer. This recognizer has the disadvantage that in only works with predefined words.
    - With the Google Speech To Text API. This works for all spoken text. Fifty requests a day are free.

This behavior was developed as a standalone Python script. In Choregraphe using standard boxes it proved to be impossible to create this behavior because of the complexity.

# Nao architecture

First read <http://doc.aldebaran.com/2-1/dev/naoqi/index.html> for a good introduction on the NAOqi framework.

The picture below shows an overview of the hardware architecture of the Nao.

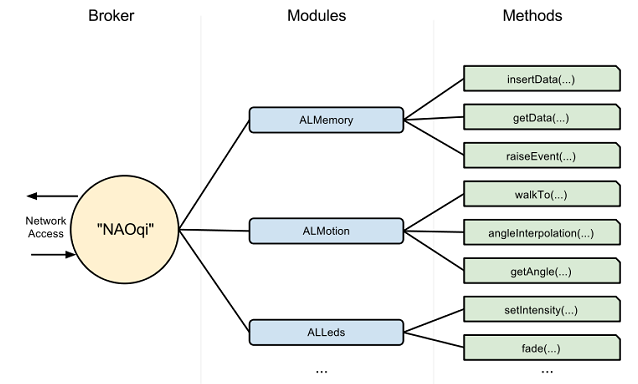


The Nao H25 V5 Evolution robot architecture has the following features:

* The head contains the main CPU board with an Intel Atom Z530 1.6 GHz processor running Linux based NAOqi 2.0.
* The CPU board in the head is connected to head sensors with I2C.
* The CPU board in the head is connected through USB with a ARM-9 microcontroller board in the chest.
* The ARM microcontroller board in the chest controls the motors and power.
* The ARM microcontroller board in the chest is connected through RS485 (two wire point-to-multipoint serial communication) with DSP’s which control the motors.
* The Nao contains Two HD cameras, four microphones, sonar rangefinder, two infrared emitters and receivers, inertial board, nine tactile sensors, eight pressure sensors.
* 25 degrees of freedom.

## Brokers and proxies

A clarifying picture from the Aldebaran NAOqi documentation with the text: “*The broker provides lookup services so that any module in the tree or across the network can find any method that has been advertised. Loading modules forms a tree of methods attached to modules, and modules attached to a broker.*”

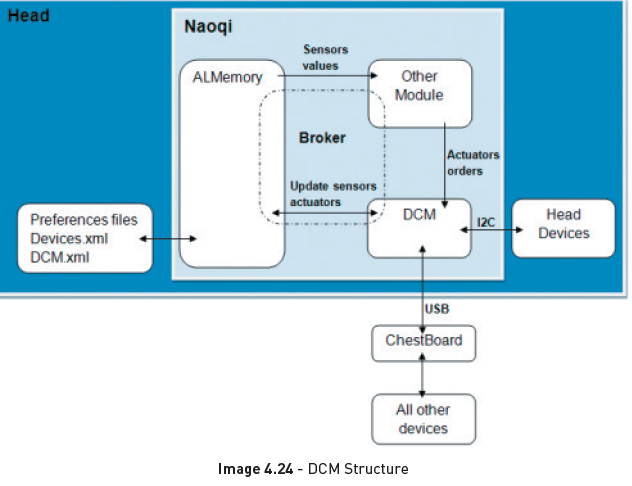


Summary: NAOqi is a framework with many modules (shared objects libraries) with functionality like motion, vision, audio, memory etc. They can be accessed by creating a proxy for a module. The proxy exposes all methods of a module and talks with a broker which exposes all modules available. The broker listens to an IP address and port (default 9559). This is a so-called proxy to broker connection. The broker will have the list of available libraries loaded (in ‘autoload.ini’). The libraries contain modules and methods and the broker will pass the call to the prober library. There are two brokers: the main broker (NAOqi broker) for local calls and myBroker (Python broker) for remote calls. When creating a module (needed when subscribing to an event and registering a callback method) a broker must be present for registering the module. A Choregraphe Python script running in the NAOqi framework will have the main broker available, but a standalone Python script has not. In this case it is necessary to create myBroker. This myBroker connects to the main broker. This is a so-called broker to broker connection. This way the modules at each side can access the modules at the other side, needed for example when subscribing to events of a remote module and specifying callback methods.

Calls to a proxy can be made non-blocking by using the ‘post’ object of a proxy.

## Device Communication Manager (DCM)

Another important component is the Device Communication Manager (DCM). See the picture below for an overview.



The DCM manages all communications to the sensors and actuators (except camera and sound). Sensor and actuator data is stored in the ALMemory module, which is also the source for the generation of events.

## Language support

Many Languages are supported, the top three being C++, Python and .Net languages. For all of these a SDK is available.  
Normally the Nao libraries are developed in C++. This is the only language with real time performance, below 10 ms loops. Python is mostly used when creating behaviors in Python. Python is also used inside Choregraphe boxes.

# Ways to work with the Nao

## With Choregraphe only

This method is the most convenient when creating simple behaviours. However quite soon the boxes and connections in the GUI become very cluttered. Because one has no real control over the behavior of each box and the related events, it gets virtually impossible to get the right behavior. For example creating a timeout when doing speech recognition is not a standard option of the ‘Speech Reco.’ box. Creating this 100% reliable using other boxes like a ‘Delay’ box requires a lot of additional connections and proved to be very difficult. A Quote from Mike Beiter's "An introduction to robotics with Nao", halfway: “Things are becoming unwieldy using Choregraphe boxes, so we’ll switch to using Python".

Choregraphe can be used for event based programming and time based programming. This document focuses on event based programming. Time based programming uses Timeline boxes and is useful when choreographing dances for example.

In Choregraphe a Nao behavior is created by using boxes and connecting them. One can use existing boxes or create new ones. The types are: Diagram, Timeline, Python or Dialog box. A standard box contains MyClass(GeneratedClass), a class derived from ‘GeneratedClass’ with standard methods such as ‘onInput\_onStart()’ or ‘onUnload()’. When the behavior MyClass is instantiated and the methods are called when the input is stimulated. Important to note is these methods will run as separate threads. Events can be subscribed to by using the event selector in the GUI and drawing a connection between the event and an input related to a method in MyClass. See below in the ‘Choregraphe Python script versus standalone Python script’ section for more details.

One can write a Python script which makes calls to the Nao API through proxies. Example call to obtain a text to speech proxy: ‘tts = ALProxy("ALTextToSpeech")’. Note that in this case no IP address and port is required as the Python module and the ALTextToSpeech module are both in the main broker. This proxy can then be used to make calls to the text to speech API, for example: ‘tts.say("hello World!")’.  
The behaviors are located in ‘/home/nao/.local/share/PackageManager/apps’. Behaviors are in ‘.xar’ files which are ‘.xml’ files containing Python code within <content> tags.

Debugging the Python script can be done printing to the Choregraphe Log viewer like: ‘self.logger.info("\*\*\*\*\*\* This is a info message \*\*\*\*\*\*")’.

## On the Nao using a SSH connection

This method is most convenient when developing in Python. One can develop Python scripts with print statements on the laptop and run it on the Nao. This is possible as the Nao already has Python installed. Compare this with developing in C++; the Nao has no development environment installed for C++, so then using the SDK on a laptop is more convenient. When the Python script runs fine, it is easy to put in in a Choregraphe Python box. In Choregraphe one can combine it with other boxes.

With e.g. ‘ssh nao@192.168.1.137’ and password ‘nao’, one can connect with a laptop to the Nao. Using FileZilla it is possible to transfer files between the laptop and the Nao. The home directory is ‘/home/nao’.

One can write a standalone Python script and execute it from the Nao command terminal. Print statements will show in the terminal. One can make calls to the Nao API through proxies. For example a call to obtain a text to speech proxy: ‘tts = ALProxy("ALTextToSpeech", “localhost", 9559)’. This proxy can then be used to make subsequent calls to the text to speech API, for example: ‘tts.say("hello World!")’. In the standalone Python script the main broker is not available so met new modules are needed, myBroker has to be created which connects to the main broker. See the section ‘Brokers and proxies’ above for more detail.

## Choregraphe Python script versus standalone Python script

When transferring this standalone Python script to a Choregraphe Python script inside a box, please note the following:

* In Choregraphe the Python script runs in the NAOqi framework. This means that ‘MyClass(GeneratedClass)’ is instantiated automatically by the NAOqi process. The \_\_init\_\_() method is the constructor and will be called. The standard methods like ‘onInput\_onStart ()’ and ‘onUnload ()’ are being called as separate threads in the NAOqi process. This means that when for example ‘onInput\_onStart()’ is triggered, a separate thread will start. This thread must also be stopped in a defined way. This can be done by setting a boolean ‘self.bIsRunning’ to False in the ‘onUnload()’ method. The ‘onUnload ()’ method will be called when the onInput\_onStart() thread ends or when the behavior is stopped. The behavior can be stopped in Choregraphe using the red 'Stop' button, or by using the 'Run Behavior' box and activating the 'onStop' input. In Python this means the ALBehaviorManager method stopBehavior() is called.
* The Python standalone script is not started by the NAOqi framework but contains a main() function. To be compatible with a Choregraphe Python script it is convenient to still define ‘MyClass(GeneratedClass)’ and in the main() function use the methods as used in Choregraphe.
* Put the main code in ‘onInput\_onStart()’. This means that the main function of the standalone script has to be moved to onInput\_onStart().
* Print statements will not show but if needed can be replaced with ‘self.logger.info()’.
* In Choregraphe, events can simply be connected to an input in the GUI. To make use of events in a standalone Python file one has to use subscribeToEvent() like described above.
* There are two ways to let the box react on a event:
  + By using the Choregraphe GUI. Use the event selector in the GUI and make a connection between the event and an input related to a method in MyClass. The underlying event subscription mechanism is configured by Choregraphe when connecting two boxes. This is used for example in the standard box ‘Bumbers’ connected to the ‘Left/RightBumperPressed’ events.
  + With Python code, both in a Choregraphe Python script as in a standalone Python script. One can subscribe to an event and register a callback methods with ‘subscribeToEvent()’, specifying the module name and callback method name. In a Choregraphe Python script the MyClass class is already registered as module to the main broker by the NAOqi framework. For the module name ‘self.getName()’ is used which is the name of the MyClass class. This is done for example in the standard ‘Speech Reco.’ box which subscribes to the ‘WordRecognized’ event. In a standalone Python script the class containing the callback method is not registered as a module to a broker yet so this has to be done. This can be done by creating myBroker and calling ’ ALModule.\_\_init\_\_() in the ‘\_\_init\_\_()’ constructor of such a class. When more events are needed it is convenient to have more classes with callback functions. These classes then also have to be registered as a module. It is important that before ending the exit() function of these modules is called. Otherwise the next time ALModule.\_\_init\_\_(self, name) will fail because the modules are already registered!

## On a laptop using the SDK

This method is most convenient when developing in C++. As the Nao has no C++ development environment installed (as opposed to Python), it is convenient to do the C++ development on a laptop using the SDK. This method can also be used when developing in Python but the benefit compared to developing through a SSH connection is less. An exception is when there is no real Nao available or execution on a simulator before execution on a real Nao is desired.

Installing the SDK on a laptop gives the most freedom in working with the Nao. It is possible to develop behaviors in languages like C++, Java or Python. The behavior can be run from the laptop. This means the Nao is controlled from the laptop which makes calls to the Nao API through proxies. These proxies can connect directly to the main broker or to myBroker. The latter in case new modules are being defined on the remote (laptop) side. See the section ‘Brokers and proxies’ above for more detail.

At the moment OS X El Capitan is not supported.

## Simulation

When there is no real Nao available or execution on a simulator before execution on a real Nao is desired, one can use a Nao simulator. Disadvantage with the current Nao simulators is that one cannot simulate realtime vision and speech recognition. There are three types of Nao simulators:Choregraphe local NAOqi

* Choregraphe local NAOqi. Inside Choregraphe you can connect to a virtual robot. It is a limited simulation environment with no gravity, vision, audio or sensors.
* Using Webots for Nao. Webots for NAO is a lite version of Webots of Cyberbotics. Webots for Nao is a complete simulation environment and can connect to Choregraphe. It has a simulated camera view but no audio.
* Creating your own simulation environment using the simulator-sdk. The simulator-sdk contains the robot models. The SDK can connect to a simulator-sdk through a HAL. One has to write plugins which use the simulator-sdk libraries to find out the state of the robot and visualize it in the own simulation environment.