A Comprehensive ROS Interface for the Aldebaran NAO

By Alicia Bargar

Mentors: Jillian Greczek and Dr. Maja Mataric

Abstract - **The Aldebaran NAO is a humanoid robot currently being employed by robotics labs across the country for use in research. ROS (Robot Operating System) is an increasingly popular open source platform for robotics researchers for developing, using, and sharing their controllers. However, there is no currently existing interface for the NAO with ROS that incorporates all of the NAO's functionality. My project aims to bridge this gap by creating a ROS stack that interfaces with the NAO in areas of key functionality and provides a framework for easily extending the stack to comprise other NAO abilities. Furthermore, the user interface with my project is designed to be both usable for new researchers and capable of greater complexity for more detailed actions.**

I. MOTIVATION

Human-Robot Interaction (HRI) is a growing field in computer science and psychology that analyzes potential for the use of robots as social agents. Some potential future applications of this research are robots used as therapists, as instructors, or as in-home companions.

As HRI expands in numbers and scope, the necessity for a common platform on which to share research grows. ROS (Robot Operating System) is an open-source operating system developed by Willow Garage, based on Switchyard written at Stanford, which may provide the necessary underlying platform. ROS is "designed around complex mobile manipulation platforms, with actuated sensing" [4]. It accomplishes this by a universal format for communication and passing and running code on separate machines [4], thus permitting the use of multi-device systems even if the devices are running on separate pieces of software. Specific design goals of ROS are to be "peer-to-peer, tools-based, multilingual, thin, and open-source" [5]. ROS is being increasingly adopted by labs and universities around the world, with courses being taught at Stanford University, Tokyo University, and Sapienza University of Rome, among others [6].

The Aldebaran NAO is a humanoid robot that stands approximately 2 feet tall. It comes equipped with LEDs, touch sensors, camera, speakers, and microphones [1]. This combination of its anthropomorphized shape and sensor-related abilities allows it to display characteristics such as human-like behavior imitation and kinesthetic, sensory, and social communication. These features make the NAO a potential interactive stimulation robot [2] and thus a candidate for research labs for studying social interactions between humans and robots. Other research applications using the NAO include humanoid robot navigation, object perception, and locomotion [3]. Overall, more than 350 universities and research labs are currently employing the NAO in their work [3].

The ROS stack **nao_robot** contains parts of NAO functionality, such as motion and camera vision. However, no one ROS stack currently provides access to all of the NAO's capabilities. An experiment in HRI may require the use of the

NAO's lights, audio, and motion functions at once during the process social interaction; this is not currently available in a single package. A comprehensive NAO interface thus is needed to facilitate HRI research with the NAO.

The purpose of my research was to create a ROS interface for the NAO, focusing on the components: lights, audio, motion, and vision, with the ability to expand for functions and modules. Additionally, the interface should retain a high level of usability and customizability for the user.

II. RELATED LITERATURE

The NAO comes equipped with two programming software packages: Choregraphe and NAOqi. Choregraphe [7] is a visual programming software in which the user creates programs by dragging, dropping, and connecting boxes containing functions. Choregraphe contains basic functionality for the NAO; more advanced lights, audio, and motion functions are not premade boxes but may be programmed and imported into the function library. A major feature of Choregraphe is the ability to chain functions together to make a more complex action, such as sitting down or standing up, and then store the new action as its own box. However, Choregraphe's box system provides limitations as well. Choregraphe can only implement hard-coded functions and reactive behaviors; directed actions and programs running multiple sensing devices require a more complex interface.

NAOqi [8] is the main software that runs the NAO. It communicates to the NAO through proxies. A proxy is an object that acts as the module it represents [8]. Each proxy represents a particular module and contain all of that module's methods. All modules can write to or read from the robot's memory. Overall, NAOqi is designed for "homogenous communication between modules, homogenous programming, and homogenous information sharing" [8]. NAOqi currently lacks an interface with ROS. Thus researchers attempting to use the NAO in multidevice systems must learn NAOqi in order to create their own NAOqi wrapper prior to beginning research. Furthermore, ROS packages not specific to the NAO cannot be run without an additional NAOqi interface.

There are two existing methods for working with the NAO via ROS: using ROS directly on the robot or wrapping NAOqi to work with the robot remotely. The first of these is cross-compiling ROS on the robot [9]. I elected to forgo this route and work with the NAO remotely instead, for this method would require manually updating the NAO's binaries every time there was an update in NAOqi or ROS.

Two ROS packages for interfacing with the NAO remotely are **nao_robot** and **nao_common**, both developed by researchers in the Autonomous Intelligent Systems Lab and the Humanoid Robots Lab at the Albert-Ludwigs-Universität in Freiburg, Germany. The ROS stack **nao_robot** wraps the NAOqi proxies pertaining to the NAO's motion and camera [10]. The **nao_camera** node takes image and camera information from the NAOqi proxy and converts them into ROS *Image* and *CameraInfo* messages so that they can be read and interpreted by other ROS programs. A similar wrapper exists for the motion functions and converts them into previously existing message types like *JointState* and *JointTrajectory* messages. Additional controls are provided in the form of function calls detailing velocities and timing within an animation as well as built-in functionality for walking. Furthermore, the **nao_driver** package contains publishers for tactile sensory data and diagnostics. The **nao_common** stack provides tools for running the NAO remotely from a computer or with a gamepad [11].

The **nao_robot** and **nao_common** stacks appear to be the only existing ROS interfaces for the NAO. The *say()* method from the audio module is included within the nao_robot stack, but is not accessible for user calls and no other audio methods are included. Furthermore, no functionality for lights exists within this stack.

II. FRAMEWORK

It is important that functionality for the NAO is covered in three key areas: lights, audio output, and motion. Ideally the stack should also contain audio input and vision functionality. Due to the wide variety of potential applications using this stack, the user should maintain a high level of control and customizability. In addition, this stack should be accessible to those without prior experience with ROS and the NAO, to facilitate a more efficient training process. Thus usability factors heavily in the design. Finally, the package's structural design should allow users to easily create and implement additional functions and modules.

For my project, I developed a ROS stack for interfacing with the NAO remotely. I created modules focused around lights, audio output, and motion, and laid out the framework for modules for audio input and camera vision. I designed function calls to be easy to learn and to design or combine for additional complexity. Furthermore, I built infrastructure that will ease future expansion of the stack for additional functions and modules.

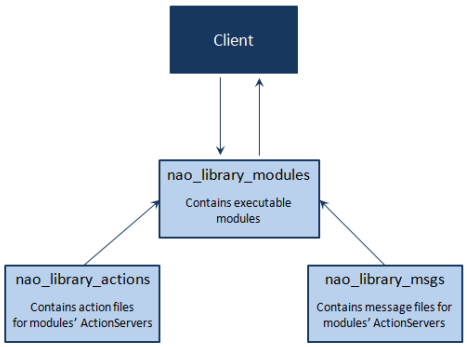

Figure 1: the nao_library layout

The ROS stack I created is named **nao_library**. A ROS stack is a collection of packages, each of which serves as a directory containing an XML file and dependencies [5, pg. 4]. Organization of the files within is otherwise determined by the packages' creator. This stack contains three packages: **nao_library_modules**, **nao_library_actions**, and **nao_library_msgs** [Figure 1]. The packages **nao_library_actions** and **nao_library_msgs** contain message files that the interface contained in **nao_library_modules** uses to communicate (Figure 1). The modules within **nao_library_modules** interact with the client via Action Servers, which are inherited from the ROS package actionlib [12].

A. Servers and Clients

ActionServers and ActionClients are similar to typical servers and clients; the client sends a message to the server, the server performs some action, and then the server may return a message back to the client. For Action Servers and Clients, the message sent from the client to the server is referred to as the goal and the message back is the result. An optional feedback message may also be declared to send information back to the client while the server is working. These messages are designed in an action file. ActionServers and Clients use goal preemption, which allows a new goal sent to preempt any previously accepted goals [12]. This allows a

user to declare a long action for the NAO to take, but then have the option to cancel the goal in favor of a new action at any point in the robot's progress.

The action files for **nao_library_modules**'s ActionServers are contained in **nao_library_actions**. Automatic generation of the action goal and result messages occur when the package is built. Non-action messages are contained in the **nao_library_msgs** package. This separation between action and non-action messages prevents mishaps when the automatic generation of the action messages occurs.

B. Action Goals

Each module has one corresponding action. Each action's goal contains an array of function messages, defined in **nao_library_msgs** specifically for each module. A function message describes everything that would need to be known about a single function. Their attributes are: *function* (the function's name, str), *time* (rospy Duration object), *str_param* (str), and other parameters that comprise all possible arguments for the module's functions.

This goal design provides the ability to send multiple functions in a single message to the server. This limits the number of messages that need to be sent and allows the server to handle timing of the functions relative to each other. Additionally, this allows the user to create a long action for the robot to perform by chaining functions together. The client can then send the goal to the server and move on to other tasks while the server is working. In particular, other goals can be designed and sent to separate modules to perform while the current server is working. This design incorporates the ease and utility of linking functions in Choregraphe with the function messages' additional control and customizability.

Result messages are designed for each module according to the information a client will normally expect or desire to receive. All modules provide access to basic functions for getting data and event information from the NAO. The *str_param* (string) is an available parameter in each function message so that it is available for these functions. Results from these function calls are contained in a list *memList* (list of strings), an available parameter in each action result message. The purpose of this encapsulation is to ensure that there is always a placeholder for these results.

C. Modules

Modules are contained in the **nao_library_modules** package. Each module consists of an Action Server and a function library. Each function library wraps either the relevant NAOqi proxy's functions or the **nao_robot** stack. The modules also each contain a dictionary named *function_table*, inherited from **NaoBasic**. This dictionary's keys are NAOqi function names and the values are the corresponding functions in the function library.

The **NaoAudioInput**, **NaoAudioOutput**, and **NaoLights** modules directly wrap NAOqi. The packages **NaoAudioOutput** and **NaoAudioInput** also inherit **nao_audio**, which is a wrapper for the NAOqi proxy ALAudioDevice and contains methods relevant to both packages. The **NaoMotion** and **NaoVision** modules wrap the **nao_robot** stack [10] to take advantage of the additional control it

provides, but with added functionality.

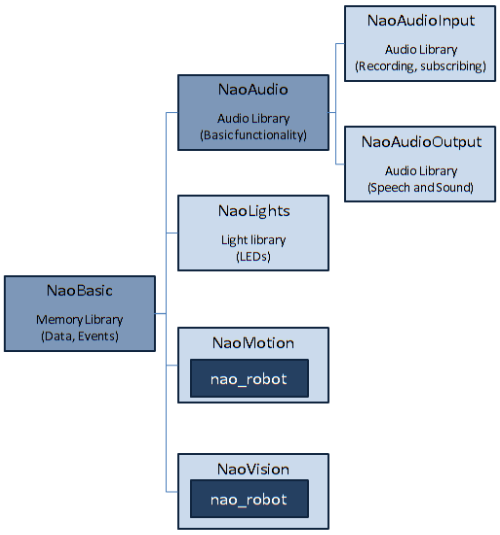

Figure 2: nao_library_modules layout. Light gray boxes indicate executable modules. Medium gray boxes indicate inherited classes. Dark gray boxes indicate existing ROS packages accessed within the module.

Each module inherits the **NaoBasic** class. **NaoBasic** provides all structure not specifically related to the action servers or the function library. Two optional parameters are available when a module is initialized, apart from the necessary parameters for the NAO's ip address and port. These are a boolean value called *autostart* and a tuple named *launchpath*. The name *launchpath* refers to a launchfile and the package containing it; the tuple is structured in the following manner: (package, launchfile). If *autostart* is set to true when **NaoBasic** is initialized, **NaoBasic** automatically tries to launch the launchfile as a subprocess in a separate thread. This causes the module and the processes started by the launchfile to simultaneously initialize and interact with each other without the need of extra terminals. This mechanism allows the motion and vision modules wrap **nao_robot**'s related servers and publisher. Reassigning *launchpath* allows other modules to be used instead.

When a goal is sent in to a module's server, the server's execute function parses each function message for the function name, the time, and groups the rest of the parameters as a list called *args*. Then the following line is called:

```
result = run(function, args, sleep_time)
```

(Note: the variable *time* is renamed *sleep_time* to avoid confusion with the Python builtin module.)

The function *run* is contained in **NaoBasic**. First it uses the function name to call up the actual function from the dictionary. Then, the function is run on *args* as a variable length argument list, as if the user had made a function call instead of sending in a message. After the function is performed, the program sleeps for the time set by the variable *sleep_time* and the result is returned.

This method aims to emulate as closely as possible a programmer's typical function calls outside of ROS without sacrificing the control that the goal definitions provide. Its generality allows it to be employed by a large variety of functions, and the simplicity allows the functions' designs to be as straightforward or complex as desired. Furthermore, it reduces the work of setting the server to run a new function to the simple task of adding a key-value pair of the function and its name to the *function_table* dictionary.

Finally, by encapsulating the function call within the run method, an overall uniformity for function calls is established. The disparate methods for communicating with the NAO: directly accessing NAOqi or sending messages to **nao_robot**, are encapsulated within the function definitions. This lets the user write functions for the NAO and send them to the **nao_library_modules** nodes without requiring analysis regarding what software is best suited for each function.

NaoBasic also contains the library for the NAOqi Memory Proxy functions. By defining the functions in this package and placing their relevant key-value pairs in the dictionary, we ensure that all modules have the ability to access data and events from the NAO's memory. This aligns with NAOqi's design in which all modules could write to and read from the robot's memory.

Some modules contain feature-specific class attributes as well for usability purposes. **NaoMotion** provides the name of each joint modified as well as the changed angle value, so that the user can identify and parse angles according to their corresponding joints, an option not previously available. This feature in combination with the new function *getAngles* (not accessible to the user in **nao_robot**) allows the user to create animations in addition to running them within their ROS program for instantaneous motion decision making. **NaoVision**'s framework provides the user with the ability to subscribe to the NAO's image stream and record in separate threads simultaneously within the server. Finally, **NaoLights** contains a python dictionary relating the NAOqi-specified led groups with the names of the light in each group in order to give the user more flexibility when getting and setting the color or intensity of a group; this lets users interact with groups of lights in the same manner as a single light.

III. APPLICATIONS

A. Function calls

For basic function calls, I aimed to provide a high degree of both usability and customizability for the user. Previously, no lights or audio functionality was included within a ROS package, so NAOqi had to be used by necessity. For the following example, therefore, I shall compare my function calls to NAOqi's function calls. We shall assume *ledsProxy*, the proxy to the NAOqi ALLeds module, and *led_client*, the ActionClient for communicating with **NaoLights**, are already initialized.

The following function turns one light on the NAO's head to half intensity via NAOqi and **NaoLights** respectively:

```
ledsProxy.setIntensity("Brain0", 0.5)

brain_half = LedFunction(function =
'setIntensity', lights = ['Brain0'],
i_values = [0.5])
```

Now let's consider a more detailed scenario.

The following is the process to turn the NAO's eyes green in NAOqi:

```
#Turn both eyes green: NAOqi
ledsProxy.setIntensity(
             "LeftFaceLedsRed", 0)
ledsProxy.setIntensity(
             "RightFaceLedsRed", 0)
ledsProxy.setIntensity(
             "LeftFaceLedsGreen", 1)
ledsProxy.setIntensity(
             "RightFaceLedsGreen", 1)
ledsProxy.setIntensity(
             "LeftFaceLedsBlue", 0)
ledsProxy.setIntensity(
             "RightFaceLedsBlue", 0)
```

This is how I implemented the same method using an ActionServer and ActionClient in ROS:

```
#Turn both eyes green: nao_library
green_eyes = LedFunction(function =
'setColor', lights =
['LeftFaceLeds','RightFaceLeds'],
i_values = [0,1,0])
led_goal =
nao_library_actions.msg.LedGoal([green_
eye])
led_client.send_goal(led_goal)
```

By creating the wrapper, I was able to include new functions like *setColor()* that can be used

with a high degree of readability. Furthermore, the optional parameters allow the user to highly customize the function. Below is the code for turning both eyes green, and then reverting to their original color after two seconds:

```
#nao_library: Turn both eyes green,
then back after two seconds
       green_eyes = LedFunction(function
= 'setColor', lights =
['LeftFaceLeds','RightFaceLeds'],
i_values = [0,1,0], time = Duration(2),
revert = True)
       led_goal =
nao_library_actions.msg.LedGoal([green_
eye])
       led_client.send_goal(led_goal)
```

Only the parameters time and revert need to be set to design this more complex behavior.

Each module contains this function layout with varying parameters. The motion module retains some additional complexity due to the parameter *JointTrajectory*. This message type was used in the **nao_robot** stack and retained due to its capability for making very controlled animations; one can define positions, velocities, and accelerations for each joint.

B. Linked Functions

Defining each function as a Function message instead of a goal allows the user to send multiple functions to a user at a time. For example, if the user wanted to NAO to nod and then stand and wave, assuming nod, stand, and wave, this is how it would be done if each was defined as a goal:

```
motion_client.send_goal(nod)
motion_client.wait_for_result()
motion_client.send_goal(stand)
motion_client.wait_for_result()
motion_client.send_goal(wave)
```

Waiting for each result would be necessary to prevent goal preemption. The following is how the action would be implemented in nao_library, assuming each animation is a defined *MotionFunction*:

```
motion_goal =
nao_library_actions.msg.NaoMotionGoal([
nod, stand, wave])
motion_client.send_goal(motion_goal)
```

Changing the time variable for each *MotionFunction* determines how long the server waits after each function, allowing the user to determine timing between these linked functions as well.

C. Animation

In **NaoMotion**, the function *getAngles()* allows the user to receive the names and current angle values of the requested joints at any point in time. One use of this function is the ability to make instantaneous decisions regarding positions in an animation for the robot. After receiving the joint names and angles from the server, the client can perform such tasks such as making decisions based on the data or manipulate the angles to create a new pose. Provided is an example:

```
time_from_start = Duration(1.0)

getAngles =
newMotionFunction("getAngles",
["RArm"])

motion_client.send_goal(nao_library_act
ions.msg.NaoMotionGoal([getAngles]))

result = motion_client.get_result()

new_pose = []

for name, angle in zip(result.name,
result.position):

    if name == "RShoulderRoll":

        new_pose.append(angle)
```

```
    else:
        new_pose.append(min(angle +
0.5, 1))
point_1 =
JointTrajectoryPoint(time_from_start,
new_pose)
```

Now the point can be called again in a *setPose()* function to return to the pose, or used as part of an animation in a JointTrajectory. In this example, we left the right shoulder roll joint untouched and increased the value of all the remaining angles.

Poses can be combined for multiple parts of the robot merely by combining the names list, ie. `["LArm"] + ["RArm"]`, and angles list, ie. `larm_pose + rarm_pose`; the only necessity is that both the names and angles list remain in the same order.

D. Interaction with Multiple Servers

In the **nao_robot** stack, goal calls can be made to the jointAngles, jointStiffness, and jointTrajectory servers [10]. For the package **nao_library**, this is generalized and expanded to the servers for **NaoLights**, **NaoMotion**, **NaoAudioOut**, **NaoAudioIn**, and **NaoVision**. The initialization for a system involving clients to the **NaoLights**, **NaoMotion**, and **NaoAudioOut** servers appears as the following:

```
motion =
actionlib.SimpleActionClient("NaoMotion
",
nao_library_actions.msg.NaoMotionAction
)

audio_out =
actionlib.SimpleActionClient('nao_audio
_out',
nao_library_actions.msg.NaoAudioOutActi
on)

lights =
actionlib.SimpleActionClient('NaoLights
',
nao_library_actions.msg.NaoLedAction)

rospy.loginfo("Waiting for servers...")

lights.wait_for_server()

audio_out.wait_for_server()

motion.wait_for_server()

rospy.loginfo("Done")
```

Each goal can then be created and sent as usual, as long as it is being sent by the client connected to the proper module; the common naming convention aims to prevent confusion on this issue. This initialization allows multi-functionality to take place within a single program. Furthermore, the option of linking functions allows one module to perform a list of functions while the program continues to create and send goals to the other modules, thus allowing additional flexibility within the program.

IV. FUTURE WORK

A. Modules

The **NaoVision** and **NaoAudioInput** modules' frameworks are currently complete but require some additional functionality and testing prior to distribution. The design and creation of a tactile sensor module would be a beneficial addition. The class **NaoBasic** could also be expanded to include diagnostic information regarding the NAO. The **nao_robot** stack [10] includes publishers relevant to each of these potential modules; additional research would help determine whether the servers should wrap NAOqi or the **nao_robot** stack for best performance.

B. Multithreading Vision

The current vision module is built using threads for streaming and recording the image stream coming from the NAO's camera. These threads are capable of running in tandem. This allows for a potential generalization of the

module into a separate package, devoted to running multiple thread subscribers for image streaming and recording. The image stream can easily be reassigned to that from another camera by changing the *launchpath* variable. Initialization, running, stopping, and deletion of threads would be handled by the user; threads would be contained in a hashtable and referred to by user-given IDs.

V. CONCLUSION

The purpose of the **nao_library** is to provide maximum functionality. New lights and audio modules, which lack previously-existing counterparts, exist in this stack. The **nao_robot** stack had an excellent infrastructure for wrapping NAOqi's motion proxy and for converting the NAOqi's image stream to ROS Image messages, so the motion module and vision module became interface wrappers for this stack. The existing motion functionality is improved upon by providing a method for getting the joint state angles with their corresponding joints. This functionality allows the user to design and perform animations using the same software, which was not previously possible.

In order to better provide future functionality for the user's needs, the stack is designed to be easily extendable as well. Making a newly-written function accessible within a module requires only adding a key-value pair corresponding to the function name and function within the *function_table* dictionary. When creating a new module, inheriting and initializing **NaoBasic** creates the primary infrastructure. The module otherwise requires two class attributes: a *function_table* dictionary and an action server, and an *execute_cb()* function that takes in the goal message and runs functions through **NaoBasic**'s run function. This simple setup for creating a new module allows the user to focus on their more immediate needs in function and message creation.

Because this stack is developed for research needs, a high priority was placed on the usability of the system. This allows experienced users to focus on their research as opposed to simple NAO interfacing, and facilitates new researchers' training on the system. Function creation is straightforward, and it is simple to chain functions together in a goal in order to make more detailed actions. The stack itself is clearly organized and source code organized in files with logical names indicating purpose and related files. The infrastructure of each module in **nao_library_modules** is contained as concisely as possible within the parent class **NaoBasic**, with the other modules containing their custom structural elements in a logical and linear manner. This structure makes the program's inner workings clear and therefore easy for new users to learn and eventually exploit.

The **nao_library**'s interface provides users with a high level of control. The ability to create long actions from simple linked functions with timing allows the user to create complex behaviors within their programs. As a result of this feature, as well as access to more detailed NAOqi functions than Choregraphe provides, the ability for independent simultaneous actions between servers, and a design for easy extension, the **nao_library** provides users with numerous options to customize it according to their needs.

The **nao_library** is a ROS interface for the NAO with increased functionality. Due to the general nature of the package and its potentially broad applications, usability and customizability underlie its design. Similarly, the structure of the package allows for easy expandability for custom functions and modules as needed by the user. The hope is to optimize the stack for research purposes. Universities and laboratories

using the NAO can thus incorporate it into multidevice systems with full functionality. This will allow researchers to focus less on training and incorporating the various NAO-software, and work on their research instead using a single underlying interface.

REFERENCES

1. "Hardware Platform – Corporate– Aldebaran Robotics | Key Features." *Aldebaran Robotics*. Web. 6 Aug. 2012. <http://www.aldebaran-robotics.com/en/Discover-NAO/Key-Features/hardware-platform.html>.
2. Libin, A. and Libin, E. (2004). Person-Robot Interactions from the Robopsychologists' Point of View: The Robotic Psychology and Robotherapy Approach. In *Proceedings of the IEEE*, 92(11).
3. "Introduction Research – Corporate – Aldebaran Robotics | For research ". *Aldebaran Robotics*. Web. 6 Aug. 2012. <http://www.aldebaran-robotics.com/en/Solutions/For-Research/introduction.html>.
4. Conley, K. "ROS/Introduction – ROS Wiki." *ROS Wiki*. 3 Feb. 2012. Web. 6 Aug. 2012. <http://www.ros.org/wiki/ROS/Introduction>.
5. 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. *ICRA Workshop on Open Source Software workshop of the IEEE International Conference on Robotics and Automation*.
6. Bravo, A. "Courses – ROS Wiki." *ROS Wiki*. 19 June 2012. Web. 6 Aug. 2012. <http://www.ros.org/wiki/Courses>.
7. "Choregraphe – Corporate– Aldebaran Robotics | Software." *Aldebaran Robotics*. 6 Aug. 2012. <http://www.aldebaran-robotics.com/en/Discover-NAO/Software/choregraphe.html>.
8. "NAOqi Framework – NAO Software 1.12.5." *Aldebaran Robotics*. Web. 6 Aug. 2012. <http://www.aldebaran-robotics.com/documentation/dev/NAOqi/index.html>.
9. Dunham, T. "nao/Tutorials/Cross-Compiling – ROS Wiki." *ROS Wiki*. 12 Jan. 2012. Web. 6 Aug. 2012. < http://www.ros.org/wiki/nao/Tutorials/Cross-Compiling>.
10. Hornung, A. "nao_robot – ROS Wiki." *ROS Wiki*. 11 June 2012. Web. 6 Aug. 2012. <http://www.ros.org/wiki/nao_robot>.
11. Hornung, A. "nao_common – ROS Wiki." *ROS Wiki*. 11 June 2012. Web. 6 Aug. 2012. <http://www.ros.org/wiki/nao_common >.
12. James, J. "actionlib – ROS Wiki." *ROS Wiki*. 6 June 2012. Web. 6 Aug. 2012. <http://www.ros.org/wiki/actionlib>.