Visualisation and Simulation of the Nao Robot using ROS

Abstract

The below thesis outlines the investigation into the viability of the Robot Operating System for use with the Aldebaran Nao robots by Maynooth University’s RoboCup team RoboEireann. The current state of the RoboEireann system is discussed with regards to its disadvantages, and what ROS could possibly do better. To address this problem, ROS is used to solve the problem of localisation, which is a common problem in RoboCup, and indeed for robotics in general. The robot is connected to ROS running on the computer, and an ARToolkit ROS package is used to localise the robot using AR tags. This is semi-successful. While the robot could localise successfully upon seeing an AR tag, the driver connecting ROS to the Nao only works when the Nao is running the software it is initially installed with. RoboEireann’s robots run on different code, which is developed around RoboCup’s requirements. This led to the conclusion that for ROS to be applicable in a RoboCup context, a driver would have to be written specifically for the RoboEireann codebase. However, the modularity of the ROS framework, as well as it’s interoperability with the robotics research community, would make writing this driver a worthwhile endeavour.

Table of Contents

TEST TEXT

Introduction 1.

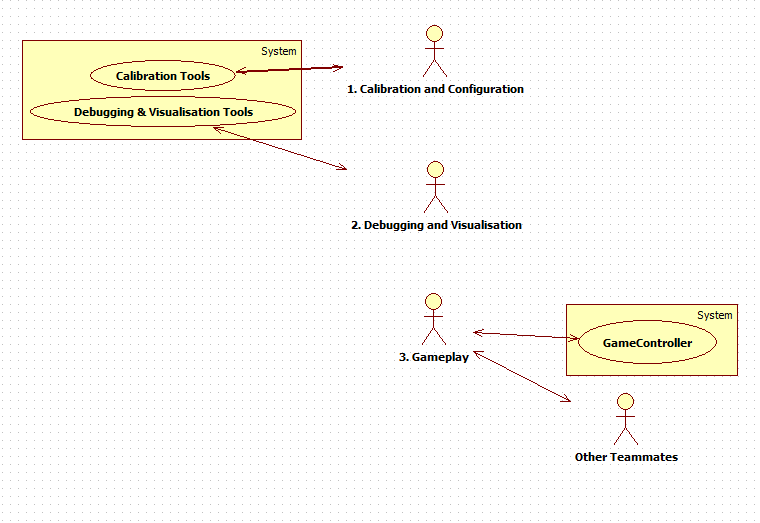
The purpose of this project is to investigative, to explore the viability and the usefulness of using ROS as a platform for visualisation and simulation for the RoboEireann team to aid in debugging. This would be of interest to the team and in terms of robotics in general because as a rule, debugging in robotics can be notoriously difficult. To see why it would be of interest specifically to RoboEireann, we first need to look at their current codebase, hattrick.

The hattrick codebase has been worked on for the past 7 years now, but despite this, it has its own limitations and drawbacks which have been caused by a number of factors. Over the years, hattrick has been worked on by a large group as different students have come and gone. Any time a student writes code and leaves the team, the amount of legacy code grows. Many of these modules, and their interdependencies, are badly documented. This means that editing or removing modules can have unforeseeable knock-on ramifications. In this sense the code is partially monolithic. This also makes the learning curve incredibly steep for new contributors, which is not ideal, especially in a team that changes so much. So in this there is a need for a well-abstracted, well documented system.

Another issue is with using external tools for things like calibration and debugging. With the tools and software currently used, many connect to hattrick differently, which means that a sizeable amount of code on the hattrick system is solely written for connecting with different tools. Also some of the tools require using the same port on the Nao, which means they cannot be used at the same time. This can get problematic when it comes to debugging. A single bridge where data can be sent and received by multiple entities would be hugely advantageous in this regard.

In a standard RoboCup game setup, there are three main stages the robot will be in.

1. Calibration & Configuration (not in play)
2. Debugging & Visualisation (not in play)
3. Gameplay



Before the game begins, the first two steps will be repeated; ideally until the team is satisfied, but typically until the game starts. For the first two stages, the team can connect to the robot to use external tools to perform calibration and debugging, but for the gameplay state, the robot is acting autonomously, only communicating with its teammates and the GameController computer.

With regard to this game structure, and in the scope of this project, ROS could be useful in the first two stages. ROS could greatly simplify and speed up debugging and calibration, for two main reasons. Firstly, with ROS, a driver could create a single bridge between ROS and the team’s machines. This bridge would be a point of entry for any ROS-compatible tool the team needs for debugging and calibration, without any contention, meaning multiple tools could run at once. Secondly, ROS is a framework designed for general-purpose robotics; many common problems or tasks in robotics have solutions in ROS, meaning that the team can use pre-existing and well-documented libraries and APIs to tackle a problem, rather than reinventing the wheel. Also, alongside ROS’s own impressive suite of tools there is a variety of third-party tools supported by ROS.

<UML of hattrick connecting to tools and ros connecting to tools>

These advantages provided by ROS could help the team develop for the robot more efficiently by not having to worry about low level concepts as much. It could also aid with the extreme learning curve. On a side note, while the above argument makes the case for using ROS in the first two stages mentioned above, it could also be used in the third stage, gameplay, by running ROS directly on the robot itself, however, this is beyond the scope of this project.

If the aim of the project would be to test the viability of ROS in a RoboCup scenario, then the approach taken would have to reflect this. This approach was to use ROS to solve a problem in RoboCup that would be consistent with the above narrative. Upon taking this approach, the viability of ROS for that given challenge would be assessed, as well as its ease of use with the Nao. This would also give a general feel for the system and its ecosystem. The problem addressed was the problem of localisation.

Technical Background 2

2.1 Robotics Concepts

Seeing as this project deals primarily with robots, some basic robots concepts are required.

2.1.1 Links, Joints and Kinematic Chains

A link is a rigid body in a robotic system. This can be a robot’s limb, or any other rigid object needing to be represented in a robotic system. Two links joined together produce a pair, e.g. a hinge or ball & socket joint. A link can be said to represent the system’s attachment to the world, and is referred to as the fixed link. A series of interconnected links and pairs is called a kinematic chain.

2.1.2 Frames & Transformations

A frame is a set of axes used to describe a location and orientation in space, typically attached to a link. When working in three-dimensional space, a frame has 3 axes at right angles to each other, and 6 degrees of freedom. This means 6 numbers are needed to define its position and orientation, i.e., x, y and z position, as well as rotation around each axis. However a frame’s location information has no meaning without another frame to compare it to. This difference between two frames’ locations and orientations is called a transformation. When a number of frames are linked together by transformations any frame can be compared spatially to any other frame by following each transformation between them along the kinematic chain. Transformations are said to go from a “parent” frame to a “child” frame.

<image of two frames and a transform>

2.1.3 Localisation & Odometry

Localisation refers to the robotics challenge of a robot knowing where it is in a given environment. Usually some context has to be given, i.e., a robot could be programmed with a map and must find where it is with respect to the map. In a RoboCup setting, localisation refers to the robot knowing where it is on the pitch. To aid in this, most robots generate an odometry frame, to be used as a frame of reference when moving. This is generated when the robot is switched on, and the transformation between the robot and the odometry is calculated by the robot’s movement speed and direction.

2.2 ROS

The Robot Operating System, or ROS as it will be called in this document, is an open source collection of tools and libraries built for the purpose of robotics. It has been developed since 2007 and has moved through alphabetically named versions since 2010. The version I was using for this project was Indigo Igloo, more commonly known simply as Indigo. At present, it is the most stable and widely supported version. As ROS Indigo was aimed at Ubuntu 14.04, that is the version of Ubuntu that was used in this project.

2.2.1 Packages and filesystem

ROS’s filesystem is divided into packages. Some of these come as part of the main ROS install, but many, many others created and maintained by developers in the community. There are currently over 2000 packages available, these supporting approximately 80 commercially available robots. Packages are atomic in that it is the smallest individual thing you can build in ROS, and are designed on the principle that each one has a certain amount of functionality, but not too much that it becomes difficult to use by other packages and software. In this way, the system is very well distributed and abstracted.

A package will contain, at minimum, a package.xml file, and a CMakeLists.txt file. The package.xml file lists metadata for the package, such as the package’s name, version, maintainers, as well as the packages dependencies. When building the package, the CMakeLists.txt file describes how to build the code and where to install it.

A package’s dependencies can be split into first-order and indirect dependencies. First-order dependencies are the ones listed in the package.xml file, directly from one package to another, but indirect dependencies are transitive. If package A depends on package B, and package B depends on package C, then A indirectly depends on package C. Because packages are made to be small, most packages have a lot of dependencies.

2.2.2 Nodes, Topics and Messages

At runtime, ROS operates by using nodes and topics. A node is an executable file within a ROS package. Nodes can publish or subscribe to topics, which are named buses for exchanging messages. A topic can have multiple publishers and subscribers.

2.2.3 TF

The TF (Transformation & Frame) package is a powerful tool within ROS for keeping track of multiple coordinate frames at once. Given a list of frames and transformations, the TF package keeps the relationship between all frames in a tree structure with time stamped transforms. This lets the user observe any given transform between any two interconnected points at any given point in time. The TF library contains methods to make the broadcasting and listening of frames and transformations easy for the user, allowing us to focus on higher-level problems. In the TF tree, every frame has at most one parent, and can have any number of children.

2.2.4 ROS Command line utilities

ROS in general is operated through the terminal, and it contains a suite of command-line tools for file system navigation, running executables, debugging, amongst other things, which form the basis for most actions in ROS. Listed here are some of these relevant to this project.

* roscd – This allows the user to change directories with just a package or stack name.
* rosrun – This runs an executable from a specified package.
* roslaunch – roslaunch allows the user to use xml files with the “.launch” suffix to run multiple executables locally, and remotely using SSH. The XML format includes commands that allow the user to specify arguments, remap topics, and more
* rosbag – With this, users can record messages from all topics, which are saved to a bagfile, and can later be played back for analysis.

2.2.5 Rviz

Three-dimensional visualisation in ROS is done through Rviz. This powerful package allows the user to view almost any information in ROS that can be represented visually. That includes frames and transformations, camera feeds, depth clouds, and more. It also allows virtual objects to be drawn to augment this visualisation. These could be block markers to indicate objects in the world, or a line to represent a boundary or a path for a robot to follow. It also supports the import of meshes so robots can be represented with a model rather than with a tree of frames and transformations.

2.3 The Aldebaran Nao

The robot used in this project, and by the RoboEireann RoboCup team, is the Nao V4, produced by Aldebaran Robotics. It is a humanoid robot standing at 58 centimetres tall. It is articulated with joints similarly to a human to give it a similar range of movement. It is also outfitted with two cameras, one facing straight out from its face and the other pointing from its face towards its feet. Two sonar sensors face out from its chest at slight angles to the left and right. Two microphones and two speakers sit on the sides of the robot’s head. The Nao’s head contains a motherboard with:

* Atom Z530 processor
* 1 GB RAM
* 2 GB Flash Memory
* 8 GB Micro SDHC

2.3.1 NAOqi

NAOqi refers to the name of the operating system running on the Nao robots, whereas the NAOqi framework is the programming framework used to program NAO. It addresses robotics-related needs such as parallelism, resources, synchronization, and events. The framework allows homogenous programming, information sharing and communication between modules.

The NAOqi framework:

* Is cross platform
* Is cross language, with an identical API for C++ and Python
* Provides introspection, i.e., it knows which functions are available in different modules

2.4 RoboCup Standard Platform League & RoboEireann

In RoboCup’s Standard Platform League, two teams of robots play soccer fully autonomously. Each team consists of no more than 5 playing robots and one coaching robot. The coaching robot observes the game from an elevated position and gives tactical advice to its teammates. The pitched played on measures 9 metres by 6 metres. The Aldebaran Nao is the standard platform used, to allow teams to focus on software development rather than the mechanics of the robots themselves.

RoboEireann is Maynooth University’s RoboCup team, and currently Ireland’s only team. Originally the team entered in 2008 with the University of Newcastle, Australia, under the team name “NUMmanoids”, where they won first place. Since 2009 they have competed under the name RoboEireann.

RoboEireann’s robots’ codebase run with two main processes. A custom NAOqi module called libagent runs all low-level communication with NAOqi. It also creates the shared memory used for communicating with the hattrick process. Hattrick is the second main process. This is where the majority of RoboEireann’s own code lies and it consists of three parts.

* NaoInterface is the main entry point for hattrick, it launches the CognitionAgent thread, runs MotionAgent, and communicates with libagent
* MotionAgent
* CognitionAgent

2.5 Augmented Reality

Where virtual reality involves creating a fully simulated world, augmented reality involves adding a simulated layer onto the real one. This is usually done with the aid of computer vision to recognise objects in the real world, and usually with real time video feed. <needs more>

2.6 ARToolKit

ARToolKit is an open source library for creating augmented reality applications. It makes use of square marker patterns, such as AR Tags. Using the camera calibration, and the known dimensions of the AR tag compared to its dimensions and skew in the camera’s view, <needs more>

2.6 Previously Completed Work

Others have completed work in this field. Some of it was of use in this project, and some wasn’t useful for the purposes of this project, but gives insight into what can be done with the Nao robots using ROS.

2.6.1 Edinferno

The University of Edinburgh’s RoboCup team “Edinferno” currently use ROS, however they use it in a capacity beyond the scope of this project. While this project is aimed at using ROS as a tool for visualisation and debugging, the Edinferno team install ROS on the Nao robots themselves and use this as the controlling software for the robot. The team have <needs more>

2.6.2 ROS packages for the Nao

There are currently 37 maintained ROS packages developed specifically for the Nao, these include drivers, controllers and GUIs. The ROS packages used in this project, apart from my own code and the

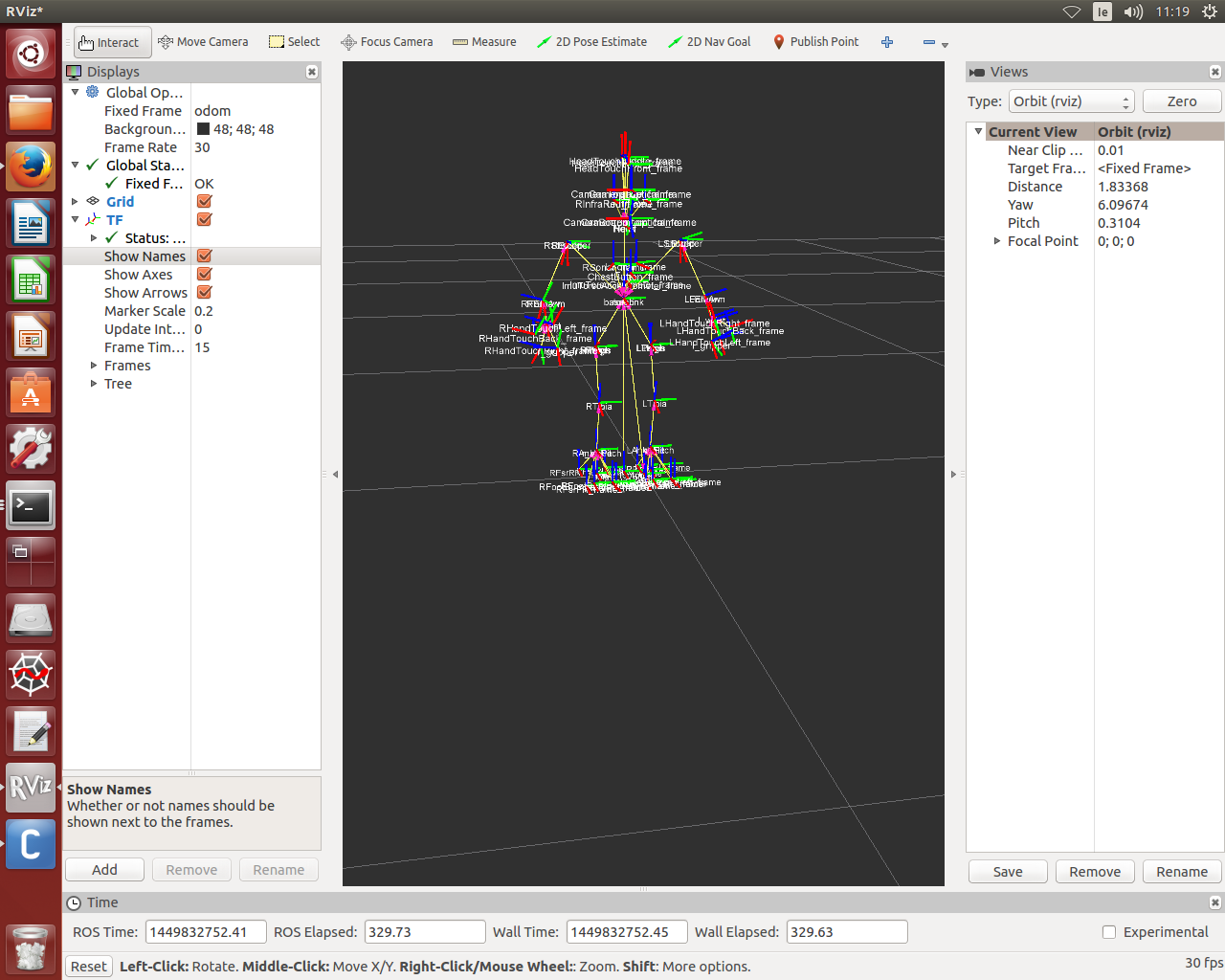
The Problem

The specific problem to be attempted in this project is, using ROS and ROS-compatible tools, to localise the robot on the pitch. Localisation is an important problem in RoboCup and in robotics in general. If we can effectively localise the robot on the pitch using ROS, this will be a good measure of how viable ROS could be in RoboEireann’s hands. <write more>

The Solution

The first step in this solution would be to get ROS running and connect the Nao to it.

Once the nao\_bringup package was running, the robot could be visualised in Rviz and a variety of its sensor data could be accessed through ROS. Using rostopic all topics could be viewed that the robot’s sensor data was being published to, and using “echo” these messages could be viewed in terminal. In Rviz, all the robot’s frames and transformations between these frames were visible, giving a skeleton view of the Nao. Meshes for the Nao exist online as a package, and once these were downloaded, a colour model of the Nao was visible in Rviz.



*Nao robot visualised in Rviz with frames (blue, green, and red lines) and transformations (yellow arrows) visible*

At this point, the NAOqi driver was interfacing with the robot, which was running the base NAOqi configuration which is the factory configuration on the Nao. As the RoboEireann team strip out quite a lot of modules that aren’t necessary for RoboCup, these modules had to be stripped out to prove that the robot could still communicate with ROS without these. The only base modules that are left running in RoboEireann’s robots are “albase”, “alsystem”, and “dcm\_hal”. Using SSH to access the Nao’s software, these three modules were disabled on the autoload file via comment. The autoload file lists the modules to be run when the Nao is booting up. Unfortunately, when the nao\_bringup was launched after these modules were disabled, the process failed. Error messages indicated an error with the naoqi\_driver package. The logical next step would be to try to get this naoqi\_driver to work with the RoboEireann codebase.

To do this, there were two options. Either a new ROS package would have to be developed to interface with the RoboEireann code, or the RoboEireann code would have to be changed so to the NAOqi driver it would resemble the NAOqi code. The latter was determined to be hugely unviable, due to the huge amount of legacy code in the RoboEireann codebase to familiarise myself with in a short timeframe, also that interfering with the RoboEireann code would hinder the RoboEireann team. The second option was chosen by default, however, after an attempt to reverse engineer the naoqi\_driver package from its source code on GitHub, it was discovered to contain over 10,000 lines of code. Because of this, this option was also deemed unrealistic with the given timeframe.

At this point, the best option would be to continue exploring the possibilities of ROS with the Nao robot. The decision was then made to continue on with the project as if the two were compatible, as a demonstration of what could be done with ROS and the Nao robots if a RoboEireann-compatible driver package was written for ROS. After researching potential avenues of investigation, a ROS package was found that contained AR Toolkit packaged for ROS use. After working through demos and researching AR Toolkit’s capabilities, the decision was made to use the ROS ar\_tools package to localise the Nao on a simulated RoboCup field using AR tags.

While experimenting with the ar\_tools package using the Nao’s camera, a problem was encountered. The topic that the Nao driver node publishes its camera data to was different to the topic that the AR Tools node subscribed to. Roslaunch has a remap command that can be specified in the launch file, however this did not seem to be working. To understand how this remap command worked a simpler publisher/subscriber example was written, where the remap command worked as expected. A remap command in the launch file will change the topics that the subsequent nodes publish to. Unfortunately, the remap command didn’t seem to work anywhere in the nao\_bringup launch file or the naoqi\_driver launch file, which was launched within the former. To remedy this, the header file within the ar\_tools package was edited so that it would subscribe to the same topic that the Nao driver node published to.

Once ARToolKit was working with the Nao, it was applied to the problem of localisation. To test this, an AR tag was hung 80cm (the height of the goal crossbar in RoboCup) from the ground. This would represent the goal. After running the driver node, the ARToolKit node, and Rviz, the robot was visible as well as a transform from the robot’s camera to the AR tag. The AR Toolkit node also drew a visualisation marker at the frame. Using the nao\_walker node, teleop keyboard commands were sent to the robot and the relative movement between the robot’s odometry, the robot, and the tag could be seen.

Next, two nodes were written. The first would perform a transform between the AR tag and the pitch. The frame was placed where the centre of the pitch would be. The second, using Rviz’s libraries, drew a rectangle 9 metres by 6 metres (The dimensions of a RoboCup pitch). Upon testing this, the pitch was not drawn accurately. This was because the AR tag used to represent the goal measured roughly 15cm by 15cm, and was rigidly transformed to a far larger object. This meant that any error in the pose and location of the AR tag would have a proportionally greater error in the pose and location of the pitch. The robot was seen to be floating above the simulated pitch, or clipping through it.

To solve this problem, a transform was needed that would make sure that the pitch’s position and rotation relative to the Nao wouldn’t change beyond the way a football pitch in real life normally would. Position on a football pitch for this purpose is largely two-dimensional, the only information that should change should be the x and y position, and rotation around the z-axis. A TF Listener object was created, which had methods to return the transformation between two specified frames at a given time. From this listener, the relevant transform information was pulled from the transform between the AR tag and the camera, and with this information, a transform from the pitch to the robot’s footprint was created.

This created another problem, this time relating to ROS’s TF system. The robot’s TF tree created by the NAOqi driver listed “odom”, the robot’s odometry frame, as the parent of “base\_footprint”, the robot’s footprint. A frame can only have one parent, so when we created the transform from the pitch frame to the footprint frame, a conflict was created. This was easily remedied, by swapping the frames so that the transformation went from the footprint to the pitch. The TF library had a method to calculate the inverse of a transform, so making this change was easy. Once these changes were implemented, the pitch was estimated far more accurately in Rviz.

Evaluation

Conclusion

The implications of this project are very specific to RoboCup and RoboEireann. <write more>

There are a number of paths the RoboEireann team could take in the future regarding ROS. The most obvious project that could be undertaken would be the writing of a hattrick-compatible driver. In this project, the naoqi-driver has been shown to open up new possibilities for working with the Nao robots. Having multiple tools communicating with the Nao all from a single bridge, and having no contention between the different packages, would be an invaluable asset to the RoboEireann team. In addition to the tools shown in this project, ROS also has support for OpenCV, Gazebo, Moveit! and more.

Another, more complex route that could be taken, would be to run ROS directly on the Nao. If this project were to be completed, development for RoboEireann could be simplified greatly. This is because ROS had packages and tools to address a multitude of common robotics problems, solutions which would otherwise have to be written into the hattrick system.

References

Aldebaran NAOqi Documentation - <http://doc.aldebaran.com/1-14/dev/naoqi/index.html>

Official ROS website - [www.ros.org](http://www.ros.org)

Official ROS wiki - wiki.ros.org

Official ARToolKit website – www.artoolkit.org

Morgan Quigley, Brian Gerkey, & William D. Smart – Programming Robots with ROS, O’Reilly Media

Appendices

More things to write about

-legacy code

-need for abstraction

-constantly changing team

-extreme learning curve

-need for modularity