Retrofitting Bioloid robot with Raspberry Pi: hardware communication and remote management interface with ROS

by Patrick Roncagliolo, Marco Lapolla, Alessio De Luca

The aim of our work is to develop a relatively affordable robotic platform that could be used as a test-bed for validation of computer vision and machine learning algorithms in other projects led by the engineering department of the University of Genoa.

In particular, the first iteration of the platform, referred either as *Bioloid* or *RaspBioloid* robot, is made by off-the-shelf components, in order to lower the time required for assembly. On the software side, we used *Python* as programming language, for faster development and testing purposes.

Introduction

The *Bioloid* takes its name from the homonymous robotic development kit by *ROBOTIS*[1]. It comes as a number of parts such as motors, joints, sensors and control units and a set of standard build configurations, such as a humanoid, a dinosaur, and so on.
We chose the humanoid configuration, but as long as the onboard computer and peripherals (that we are going to describe later) can be fitted to the robots body, there is no specific reason to consider one configuration above the others.
The humanoid configuration uses 18 *Dynamixel* motors, 3 of them for each arm and 6 of them for each leg. The head and the hands are fixed; the head contains a hub that merges the control chains and supply power to them.

The standard microcontroller can be programmed with *ROBOTIS RoboPlus*[2] software suite, but fails to provide an open platform to interact with third-party hardware and an evolved custom software logic. This is the reason why we decided to swap the default microcontroller with a more powerful *Raspberry Pi 3*[3].

An open platform such as a Linux-powered *ARM* dev board can be interfaced with third party USB peripherals, can be expanded with the so called “shields” (plug-in boards that can be stacked on the *Raspberry* itself), and can be exposed on a network for remote control by a more powerful host (the “master node”).
The master node could then implement an evolved and resources demanding control logic that can integrate also external tools such as laser scanners, point cloud cameras and more, and close the loop by instructing the robot on the next action to take.
Running all the calculations on the onboard *Raspberry Pi* would pose a number of problems, such as:

- great power consumption levels (and consequent battery drain);
- overheating of the System on Chip (that can lead to frequency scaling or damages);
- need of wired connection for out-of-field peripherals (such as *Kinect* or *Hokuyo*);
- lower closed-loop rate.

[1] www.robotis.us/robotis-premium
[2] www.robotis.us/roboplus2
[3] www.raspberrypi.org/products/raspberry-pi-3-model-b

Requirements and HW/SW choices

The requirements for the robotic platform we chose to develop were:

- Linux onboard computer;
- wireless setup;
- battery monitoring;
- motors control and feedback;
- inertial data readings;
- simple and modular software approach;
- an interactive graphical user interface to manage the robot remotely.

The software requirements led to the choice of *ROS*[4] (Robotic Operating System) as the IPC middleware to use. *ROS* is a framework that enables multiple software nodes, that could be running on different machines, to exchange data or commands on an IP network. Nodes can be implemented in *C++* or *Python* (and other unofficially supported languages): as already stated, we chose *Python*. ROS is open-source and free to use, and runs on *ARM* too.
The community and some hardware manufacturers provide a huge variety of packages (mainly containing *ROS* nodes) that are ready to be downloaded and executed without changes. Software repositories for Ubuntu platforms are also available with prebuilt packages, and this is essential on a slow *ARM* SoC such as the *Raspberry Pi* one. Otherwise, *C++* nodes compilation onboard could take a lot of time and cross-compilation is not so straightforward neither.

We chose a number of third-party products to be fitted on top of the existing robot configuration. We introduce each of them briefly recalling the installation process:

- *PhidgetSpatial 3/3/3* (by *Phidgets*)[5]: an IMU (Inertial Measurement Unit) that can measure linear acceleration, angular velocity and magnetic field strength. It has been placed under the front plastic cover of the robot body and fitted with screws to the underlying frame. It is connected to the RPi with a USB-A to USB-Mini-B cable;
- *USB2AX* (by *xevelabs.com*)[6]: an unofficial USB dongle that can read and write on a 3-wire *ROBOTIS* proprietary “AX” serial bus. It is directly plugged in a USB port of the *RPi* and connected to the motor chains hub with a 3 wire cable;
- Modular RPi case (by *ModMyPi*): an ABS clear case needed to properly secure and protect the dev board itself and the UPS extension board to the back of the robot. We had to create 4 reversed keyholes on the bottom of the case to lock it on 4 matching additional screws, partially screwed on the back of the robot. We also created the necessary openings on the I/O ports side to accommodate external battery cables and 3 holes on top to allow reaching reset buttons;
- *Raspberry Pi 3 Model B*: an SBC (Single Board Computer) with a quad core, 64 bit CPU, 1 GB of RAM, Wifi, Bluetooth, Ethernet and USB ports. Storage is provided as an external SD card and there’s a 40-pin header that makes it extensible with plug-in “shields”.
- *UPS Pico Shield 3.0B+ Advanced* (by *PiModules*)[7]: a *RPi* shield that provides advanced management of onboard battery and external power sources, including both USB and external batteries in a 7-20 voltage range. Various features are exposed via *i2c* bus to the *RPi* and hot-swap of input sources is supported. We soldered the parts that were not already

[4] www.ros.org
[5] www.phidgets.com/?tier=3&catid=10&pcid=8&prodid=1025 (now discontinued)
[6] http://www.xevelabs.com/doku.php?id=product:usb2ax:usb2ax
[7] http://www.pimodulescart.com/shop/item.aspx?itemid=51

assembled, and we created a path to the *RPi* hard-reset pin by soldering an additional wire, because we experienced troubles with the standard, gold plated, spring pin.

We used *Ubuntu 16.04 LTS* on both robot and master node, because of its stability and ease of setup; furthermore the *ROS Kinetic LTS* distribution is fully supported on this *Ubuntu* version. We also used *NetworkManager* “nmcli” tool to create a wifi access point for direct connection with the *RPi* and enabled *OpenSSH* server.

Project development

Motors subsystem

We published our software and detailed setup tutorial on *GitHub*, and we structured our repository[8] as a set of independent packages using *git submodules*. The main repository acts as a meta-repository that points to the latest commit of each submodule and also contains general setup wiki pages.
Here’s the list of the submodules. Each submodule corresponds to a *ROS* package and we will discuss them in detail later:

- *motors*
- *motors-msgs*
- *motors-srvs*
- *pico*
- *pico-srvs*
- *control-panel*

The robot has 18 *Dynamixel AX-12*[9] motors; they communicate on a 3 wire bus (GND, VDD, DATA)[10]. The data line is shared among all devices connected to the same controller (in this case, our *USB2AX*) and a custom serial protocol[11] called *Dynamixel Protocol 1.0”* is used.
While we were in the initial phases of our project, we discovered that the current implementation of the *Dynamixel SDK*[12] (an userspace driver for *Dynamixel* I/O) was written in a lot of languages except *Python*.
Python C bindings were provided by exposure of *C* symbols with the *ctypes Python* module[13], with no attention to details such as providing description of function prototypes: that would have enabled runtime checks on how symbols were called from *Python*; since *Python* is a dynamic language, it would have allowed incorrect parameters in calls to the underlying API.
We managed to add all this type of runtime informations and to map the style of the *C++* implementation by providing classes and not only a perfect match of the *C* functions obtained by the basic binding.
After having submitted those improvements as a pull request[14] on the official upstream repository, *ROBOTIS* revealed us that a native *Python* implementation was in an internal development phase. They released us a first release candidate that had a lot of bugs, partly because their developer was not accustomed to how *Python* really works. We reviewed all the code and we contributed a set of quite big commits[15], discussing changes and conceptual errors with them. After a lot of testing and

[8] www.github.com/roncapat/RaspBioloid
[9] http://support.robotis.com/en/product/actuator/dynamixel/ax_series/dxl_ax_actuator.htm
[10] http://support.robotis.com/en/product/actuator/dynamixel/dxl_connector.htm
[11] http://support.robotis.com/en/product/actuator/dynamixel/dxl_communication.htm
[12] https://github.com/ROBOTIS-GIT/DynamixelSDK
[13] https://docs.python.org/2/library/ctypes.html
[14] https://github.com/ROBOTIS-GIT/DynamixelSDK/pull/181
[15] https://github.com/ROBOTIS-GIT/DynamixelSDK/pull/182 and https://github.com/ROBOTIS-GIT/DynamixelSDK/pull/185

validation, we are proud to say that now *Dynamixel SDK* has a native and fully working *Python* implementation and we helped to make this happen faster.

Dynamixel SDK handles low-level communication, like serial port management, packet tx/rx and error detection. We also dived in the *Dynamixel Workbench*[16] codebase and we made a native *Python* porting, but this higher level abstraction of the protocol is quite chaotic due to the obscure ways they handle differences and detection of protocol 1.0 and 2.0; we were much confident in how the SDK worked that we decided to use only the SDK in our motor node implementation. It may be the case to work for a newer, simpler and shorter implementation of the *Dynamixel Workbench* from scratch, instead of losing time on side-by-side ports of a confused implementation.
Working at a lower level also gave us the occasion to get a huge performance gain, as explained below.

The *ROS* node that handles motors opens the serial port and then by default sets torque on each of the 18 motors. It also writes some default parameters we chose.
Dynamixel protocol is based on read and write packets. A memory table contains all the exposed parameters of the robot and they can be read only or writable too. By specifying memory offset and length of a field, parameters are then read or overwritten. Of course, a packet brings some overhead: the header bytes, the type byte, the filed address and length and the checksum.
The three data that we want to continuously read are: current position, current speed and current load.
If we would have chosen to use the *Dynamixel Workbench*, each high level call (eg. get_current_position()) would have involved a request read packet and obviously a response packet. During this round-trip time, the other motors could not be read or written (protocol 1.0 does not support sync read feature): 3 read packets plus 3 response packets are needed in order to read status from 1 motor.
Luckily, we know in detail how the lower-level *Dynamixel SDK* works and by looking at the memory mapping of the *AX-12A*, position, speed and load are adjacent fields. By issuing a single read request packet (8 bytes long, or a 2/3 reduction of the bytes required for three distinct read calls) for a memory area as wide as 6 bytes instead of 2 each call, we receive a response packet 6+6 bytes long (instead of three 6+2 bytes packets, thus a 1/2 reduction of bytes needed): this has been a great achievement.
The *ROS* node runs an infinite loop that reads data from each motor and publish it to a correspondent *ROS* topic. The node also provides basic services such as the ability to enable/disable torque of a single motor or all motors and the ability to move a motor to a specific goal angle with a given speed. We internally keep track of position goals and torque status, in order to publish them among the data of each motor at every loop cycle, since we need them for GUI feedback (as explained in a following section of this document).
The interesting thing is that all this read-publish / listen-write work must be done from the same process, because, obviously, the serial port could not be shared. Of course, the calls to the services (asynchronously processed by the *ROS* framework) must not interfere with the infinite reading main loop. We used a mutex to avoid race conditions between master thread and service callback workers.
It would be interesting to explore a thread-safe implementation of the *Dynamixel SDK*: currently it wrongly uses a boolean flag to know if an operation is pending or not, instead of proper synchronization primitives.
One lesson we learnt about the choice of motors is that protocol is very important and we should ensure that the bus is used efficiently. A protocol that could handle the concurrent read a field in a chained response, motor-by-motor and broadcast writes would allow better performance.

[16] https://github.com/ROBOTIS-GIT/dynamixel-workbench

IMU subsystem

As we mentioned before, the *Bioloid* robot is equipped with an IMU; this board has a 3-axis accelerometer, gyroscope and compass with high resolution readings at low magnitudes.
This type of IMU does not keep track of the angular position, so we need to use a filter node[17] that allows us to calculate it by integration. We also do not have position data, nor linear velocity, but only linear acceleration: this means that estimating position by integration would not be ideal, because we would introduce errors, the result could diverge and we would have to offset precisely the gravity acceleration.
If we would like to track the robot, we should use point cloud camera like *Microsoft Kinect* or laser scanners such as *Hokuyo* and calculate position with sensor fusion techniques.

The manufacturer already provides a set of libraries and *ROS* nodes to interact with this board. The software architecture can be described as follows: *libphidgets* is the low level communication driver; phidgets API provides a generic *C++* abstraction of the communication primitives with Phidgets devices and an *Imu* class for higher level abstraction. Finally, *phidgets_imu ROS* package uses the aforementioned API to provide readings[18]. Optionally, a filter listens the imu node and calculates angular position if needed.
We observed a publish ratio of 250 Hz, that is at least an order of magnitude better than what we could achieve with Kinect processing, as demonstrated by our colleagues of the Robotics Engineering course, so no action was required here for performance improvement.
However, given that the *Madgwick filter* we used ignores covariance data that is published from the imu node, we tried to modify both the imu node, the filter node and the format of the messages exchanged between them, removing the covariance fields and thus reducing overhead and bandwidth usage. We got the same frequency figure as before the code simplification, so we believe that this tweak should not be seriously considered as necessary.

Power subsystem

Power management is a critical aspect of the project. We needed a battery because we wanted to create a 100% wireless robot: cables can lead to noise in tasks such as camera detection and tracking of the robot, they can interfere with robot movements and connectors (and relative I/O ports) could get damage if the robot falls to the ground. We wanted the robot to be free to move. In order to accomplish this, we currently use a 11.1V 3C LiPo battery strapped on the back of the robot. An Y-cable provides power to the UPS shield and to the motor hub; battery can be swapped with a cable coming from a bench power supply in situations when robot does not need to move freely.

No *ROS* node existed yet for the *Pico UPS shield*. We wrote a little driver based on *i2c*, exposing the most useful functionalities of the shield to our custom *ROS* node. The *i2c* addresses we used are described in the manufacturer manual.[19]

The *pico* node offers status information as a *ROS* service: we think that the data provided by this subsystem are not subject to great changes within a period of a few seconds, thus not requiring continuous loop readings. We could also have implemented a read/publish loop with a sleep primitive, but we thought that a service call to retrieve the data whenever we want is better, because this way we do not have to wait for the first published message after subscribing to the topic, or we do not have to implement a trigger service to avoid this delay.
The main service provided retrieves voltage values of onboard battery and external sources, current source and onboard battery charge status. We also implemented a file-safe system shutdown as a separate service offered by the same node. In order to properly setup FSSD, first a client have to

[17] http://wiki.ros.org/imu_filter_madgwick
[18] http://wiki.ros.org/phidgets_drivers
[19] https://github.com/modmypi/PiModules/wiki/UPS-PIco-Programmers-Registers

submit the desired low level voltage value, and then enable/disable the feature with a dedicate service call. If FSSD is enabled, the node runtime loop will check if external source voltage drops under the desired value and if so, it does a system shutdown.
The actual implementation of FSSD has led to a question: how to properly shutdown system without the commonly required permissions to run shutdown commands in a traditional Unix shell? On ubuntu, typically a "*sudo shutdown now"* have to be issued. This question led us to the usage of *DBus*[20] as RPC framework widely used in modern Linux distributions. In particular, we managed to call the same *DBus* service that the classical "shutdown" button in every desktop environment calls in order to shutdown the system, without any permission required. The *DBus* service responsible to shut down the machine in Ubuntu 16.04 is called *logind* and exposes a *PowerOff* method.
However, we encountered an obstacle while testing the full project setup by running roslaunch from a different machine: we discovered that some *DBus* actions are subject to *polkit*[21] permission restrictions. Formerly known as *PolicyKit*, *polkit* is a centralized system for managing authentication requirements for a certain command. The *polkit* configuration for *logind* requires interactive authentication for some actions like *PowerOff* if the *DBus* client requesting it is a process in "inactive" shells (*SSH*, *VNC*...): *roslaunch* connects to the robot and instantiates nodes through an *SSH* connection, so the *pico* node couldn't shut down the system if launched remotely. We solved the problem by adding a *polkit* rule to override this behaviour, and allows PowerOff if issued by the "bioloid" user, no matter the type of session (local or remote).
Switching off the *RPi* was only half of the problem. The battery is connected both to the board and to the *AX* hub: we used the *Pico Shield* onboard terminal blocks, that expose contacts of a bi-stable relay on the shield itself, to short or not the positive wire that powers the *AX* hub. We wrote a very simple one-shot *systemd* service that switches on the motor bus at system startup and switches it off at system shutdown by issuing the two related *i2c* commands.

Remote control panel

The control panel GUI is obviously the part of the projects that showcases the use of all the data and services described above. We developed the GUI with *PySide2*[22] framework that is an official *Qt5* binding for *Python*. We actually started developing using the unofficial *PyQt*[23] port, but as we were writing code, the *Qt Foundation* announced the first technical preview of *PySide2*, so we switched framework with little changes.

The control panel is actually a *ROS* node. Both *ROS* and *Qt* are high level frameworks with a lot of details hidden, such as thread pools for callback executions. *Qt* callbacks are called "slots" and they are executed whenever a connected "signal" is emitted. *ROS* callbacks are called if a service is triggered or if some data were published on a given topic. Calling *Qt* primitives directly from *ROS* callbacks (eg. for updating motor angular position in GUI) lead to race conditions between the two frameworks. The correct fix and approach was to emit signals from *ROS* callbacks and let the *Qt* event loop manage the forwarded data by calling the correct slot at the right time.

The GUI is very simple: we have a single window that shows a blueprint-like drawing of the front view of the robot; on the right side there's a panel where we show data coming from the imu filter and the battery.
The robot view was obtained by extracting and manipulating *SVGs* from the official assembly *PDF* manual by *ROBOTIS*. This is not a screen grab, but actually the original vector drawings. Spinboxes with angles in degrees are tied to each motor with an arrow and the small checkboxes next to them

[20] https://www.freedesktop.org/wiki/Software/dbus
[21] https://www.freedesktop.org/wiki/Software/polkit
[22] https://www.qt.io/qt-for-python
[23] https://riverbankcomputing.com/software/pyqt/intro

allow to enable/disable torque on an individual motor. There are two buttons to enable/disable torque for all motors.
The interesting thing is that we derived the *QDoubleSpinBox* class and we provided a method that not only updates the value showed, but if it's near the goal position of the motor, shows a green color as background; instead, if the current load exceeds a certain value, the background becomes red. When a spinbox is focused (cursor inside) the value stops to be updated and we wait for user input; when the spinbox loses focus, the value begins to be updated again.
On the right panel, we also implemented FSSD management: user can set the desired low level value and enable/disable FSSD.
IMU data is provided at a 250 Hz rate and aggregate motors publish rate on the *RPi* is roughly 750 Hz. Each IMU packet that arrives leads to the update of 9 distinct *Qt* labels, bringing to a total of at least 2000 GUI repaint events per second. In order to lower the overhead and display data at a more human refresh rate, we manually managed the repaint process: a timer periodically enables GUI updates, triggers a repaint event and disables the GUI updates again. This way, we lowered waste of CPU cycles by a significant amount.

Software integration

We added a *ROS* launch configuration to the GUI package: this file is an *XML* description of the operating scenario, including the name, address and credentials of each remote *ROS* machine involved, the environment variables needed, what nodes to start, in which order and on what machines.
Our launch file is designed to be launched from the master node and knows how to connect to the *Bioloid*: an *SSH* connection on an address that belongs to the special subnet of the robot wifi access point we properly configured.
Once connected, imu, ups pico and motor nodes are started on the robot system, while the *madgwick* filter and the control panel are launched on the local master node. The launch files also specifies interconnections among nodes by means of topic names and service names needed to map publishers/subscribers and callers/listeners appropriately.