frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 26 April 2016
doi: 10.3389/frobt.2016.00024

The iCub Software Architecture: Evolution and Lessons Learned

Lorenzo Natale, Ali Paikan, Marco Randazzo and Daniele E. Domenichelli*

iCub Facility, Istituto Italiano di Tecnologia, Genova, Italy

The complexity of humanoid robots is increasing with the availability of new sensors, embedded CPUs, and actuators. This wealth of technologies allows researchers to investigate new problems like multi-modal sensory fusion, whole-body control and multi-modal human-robot interaction. Under the hood of these robots, the software architecture has an important role: it allows researchers to get access to the robot functionalities focusing primarily on their research problems and supports code reuse to minimize development and debugging, especially when new hardware becomes available. But more importantly, it allows increasing the complexity of the experiments that can be carried out before system integration becomes unmanageable, and debugging draws more resources than research itself. In this paper, we illustrate the software architecture of the iCub humanoid robot and the software engineering best practices that have emerged driven by the needs of our research community. We describe the latest development of the middleware supporting interface definition and automatic code generation, logging, ROS compatibility, and channel prioritization. We show the robot abstraction layer and how it has been modified to better address the requirements of the users and to support new hardware as it became available. We also describe the testing framework, and we have recently adopted for developing code using a test-driven methodology. We conclude the paper discussing the lessons we learned during the past 11 years of software development on the iCub humanoid robot.

Keywords: humanoid robotics, software engineering, software middleware, Quality of Service, test-driven development

OPEN ACCESS

Edited by:
Samer Alfayad,
Université de Versailles Saint-Quentin-en-Yvelines, France

Reviewed by:
Arnaud Blanchard,
University of Cergy-Pontoise, France
Vincent Hugel,
Université de Toulon, France

****Correspondence:***
Lorenzo Natale
lorenzo.natale@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *23 November 2015*
Accepted: *04 April 2016*
Published: *26 April 2016*

Citation:
Natale L, Paikan A, Randazzo M and Domenichelli DE (2016) The iCub Software Architecture: Evolution and Lessons Learned. Front. Robot. AI 3:24. doi: 10.3389/frobt.2016.00024

1. INTRODUCTION

The rapid evolution of humanoid robots is pushing the requirements on their software infrastructure. The availability of low-cost, off-the-shelf sensors for depth perception, IMUs, tactile and force sensing allows robots to be equipped with richer and redundant sensory systems. New actuators give joints higher maximum torque, allow designers to increase the dexterity of the robots and to implement force or impedance control. New technology for optical or magnetic encoders allows measuring movement in various points in the kinematic chain providing redundancy, fault tolerance or, in presence of elastic elements, accurate torque measurement. This evolution opened up new research problems, such as multi-modal sensory fusion, whole-body force control, and multi-modal human–robot interaction to mention just a few. However, exploring these research directions comes at a high cost in terms of software development. When existing hardware is replaced with new one, it yields to software obsolescence, new development, debugging and consequent changes in low-level software layers that trigger the redesign of higher layers. Experiments that build on top of simpler capability are possible only if the software architecture supplies researchers with appropriate

tools that allow them to focus on the goals of their research. Also, the software itself evolves. Languages, operating systems, and libraries get upgraded and change, sometimes without maintaining backward compatibility. The appearance of Robot Operating System (ROS) and its rapid adoption and growing community (Quigley et al., 2009) have changed how people develop software and pushed many robot developers to provide ROS compatible interfaces for their software or to adopt it altogether.

These problems have been present since the beginning of the development of the iCub platform and through the past 11 years of its evolution. The iCub is a humanoid robot platform that was designed for research in cognitive system. Its main goal is to support experimental research and, for this reason, it is not designed with a specific application in mind. The hardware development of the robot was also driven by research goals. The software infrastructure of the robot was designed and adapted following these constraints. At the lowest level, it had to support new hardware as soon as it was released and reduce the impact of hardware changes to the user code. At the higher level, the software architecture was designed to support rapid prototyping of experiments that required integration of many capabilities: visual and speech perception, control of attention, learning, reaching, grasping, and, more recently, balancing and walking.

In this paper, we provide a review of the software architecture of the iCub robot including recent developments aimed to better support the evolution of the robot and the needs of the research community. We describe the Yet Another Robot Platform (YARP) middleware and how it has been extended with facilities to increase determinism in time-critical loops. We present new tools and best practices that have been adopted for logging and to aid developers in the task of developing component interfaces, defining new data types, and interoperate with software developed for other robots (including software from the ROS eco-system). We describe the robot abstraction layer, which allows the same code to control the real robot or a simulation, on-board, or through a network link. This abstraction layer separates high-level components from the hardware implementation including the communication infrastructure. The core of the robot abstraction layer has not changed much in the years preserving backward compatibility, but it has been extended to better support new control modes and sensors. We describe the component that provides access to the robot, i.e., *robotinterface*, showing how it can be configured depending on the available hardware and to run time-critical control loops directly on the robot.

Robotic software applications quickly grow in the number of components and are therefore difficult to engineer and develop. Software engineering best practices suggest to divide such systems in simple units that are independently developed, tested, and integrated at a later stage. In the second part of the paper, we describe the tools that we have developed to support deployment and monitoring of components (i.e., the *yarpmanager*) and, more recently, testing. Test-driven development (Beck, 2003) was used to develop the YARP middleware but was adopted only recently to validate the robot software interface and the control algorithms. We developed the *Robot Testing Framework* (RTF) that allows testing robot software using the real robot as well as simulations. We describe the design choices that have driven the development of the RTF and how it has been adopted for testing software components and interfaces of the iCub robot. We conclude the paper with a discussion on the lessons we have learned in the past years of software development on the iCub robot drawing the conclusions of this work.

2. A BRIEF OVERVIEW OF THE iCUB AND ITS EVOLUTION

The iCub is an open humanoid robot platform that was developed for research in cognitive systems (Metta et al., 2010; Parmiggiani et al., 2012). It has 53 joints actuated with brushless and DC motors. Motion generation is carried out in dedicated boards embedded on the robot and interconnected through a local bus (initially we used CAN bus, but shifted recently to Ethernet to increase the available bandwidth). These boards host programmable CPUs that can perform position control with trajectory interpolation, velocity, and torque control. The iCub was initially equipped with cameras for vision, microphones an IMU on the head, and motor encoders for measuring motion. This initial set of sensors grew with time, by introducing 6 axis F/T sensors in various points of the kinematic chains (roughly located at the shoulders and hip, and eventually at the ankles), and a system of tactile sensors[1] that, starting from the hands and forearms, has been extended to cover a large part of the whole robot (for a total of 4000 sensing units located on the arms, torso, legs, and feet soles). At the same time, inertial units and gyroscopes became inexpensive and easy to integrate in the electronics that control the tactile systems and the motors. The robot mounts on the head a PC104 computer equipped with an Intel CPU that runs Linux. This computer works as a bridge interconnecting the CPUs on the local bus with the external cluster of computers that performs heavy computation outside the robot. Connection with the external cluster is achieved using either gigabit Ethernet or wireless.

The software architecture of the robot can be broadly separated in two layers. The firmware consists in the first layer and runs on the embedded CPUs. Communication between boards and PC104 computer uses a custom networking protocol (over CAN or Ethernet). The second layer consists in all the software components that run on the head computer and on the external cluster. These components communicate using a peer-to-peer publish-subscribe architecture implemented using the YARP middleware (Fitzpatrick et al., 2008; Metta et al., 2010).

3. MIDDLEWARE

Best practices in robotics advocate adoption of component-based development (Brugali and Scandurra, 2009) and the so-called separation of concerns between computation, communication, coordination, and configuration (Bruyninckx et al., 2013). Following this approach, components encapsulate robot functionality in a way that promotes interoperability, composability, and reuse, irrespective of the robot, programing language,

[1]Tactile sensors on the iCub are based on capacitive technology (Maiolino et al., 2013).

operating system, computing architecture, and communication protocol being used.

Computation is the core of the components and includes functionalities and algorithms. Communication allows modules to exchange data in a way that is agnostic with respect to the underlying operating system, medium or protocol. The separation of concerns is implemented by the middleware in the form of read/write primitives to receive and transmit data. Coordination is the code required to orchestrate modules: it determines how modules interact to achieve a certain task (Lütkebohle et al., 2011; Klotzbücher and Bruyninckx, 2012; Paikan et al., 2014a). Configuration allows reconfiguration of modules to better adapt them to specific domains, it allows controlling all parameters that affect the functioning of the system, including dependencies across modules and protocols.

Software middleware supports some or all the functionalities described above. Generic middleware like CORBA,[2] Ice,[3] D-Bus,[4] or ∅MQ,[5] provide complete communication backbones. They are rarely employed in robotics because they lack specific components and have a steep learning curve. Robotics middleware [OROCOS (Bruyninckx, 2001), Player (Collett et al., 2005), YARP (Fitzpatrick et al., 2008), Orca (Brooks et al., 2005), ROS (Quigley et al., 2009), OpenRDK (Calisi et al., 2012), Mira (Einhorn et al., 2012), LCM (Huang et al., 2010) to mention just a few] provide a subset of communication paradigms (Remote Procedure Call and/or publish-subscribe). Some middleware defines interfaces for families of devices (Collett et al., 2005) for better modularity and portability.

The iCub software architecture is similar to the port-based software abstraction (Stewart et al., 1997) and is built on top of the facilities provided by the YARP middleware. To better illustrate the communication patterns supported by the YARP middleware, we will adopt the terminology introduced in (Eugster et al., 2003) for publish-subscribe architectures. Eugster et al. (2003) introduce three levels of decoupling to characterize various flavors of communications, namely: *space decoupling*, *time decoupling*, and *synchronization decoupling*. *Space decoupling* is achieved when components produce messages without being explicitly aware of the number and location of the receivers. *Time decoupling* guarantees message delivery even if senders and receivers are not active or connected at the same time. Finally, *synchronization decoupling* requires that messages are sent and received asynchronously by the communicating entities. When communication is asynchronous, it is sometimes important to guarantee that messages are correctly received by slow recipients [this is called *persistence* and is another key property of publish-subscribers architectures Eugster et al. (2003)]. In this respect, robotic middleware often implements policies that aim at reducing latency in real-time control loops, even at the cost of dropping messages [e.g., Fitzpatrick et al. (2008) and Dantam et al. (2015)].

YARP (Fitzpatrick et al., 2008) implements a variant of the publish-subscribe paradigm, i.e., the *observer pattern* (Gamma et al., 1995), which is a type of distributed publish-subscribe providing space and synchronization decoupling. In addition it is multi-platform, in that it provides a portable abstraction for the operating system, the communication protocol and the robot hardware. In YARP *Port* objects deliver messages of any size and type across a network, using various underlying protocols – including shared memory. *Ports* can be configured to implement publish-subscribe with different levels of decoupling and dynamically reconfiguration of connections and protocols. YARP *Ports* have read and write primitives that can be blocking or non-blocking for synchronous or asynchronous communication. A component that uses *Port* objects to perform a synchronous write, waits until all receivers confirm reception of the message. Similarly, a component that performs a synchronous read waits until a new message is received by the *Port*. By default, *Port* objects in YARP are configured for both synchronous read and write: this guarantees correct delivery of messages without extra code. The *BufferedPort* object is a specialization of a *Port*, which provides synchronization decoupling. *BufferedPorts* are active objects able to store and handle messages internally either for transmitting or receiving them using dedicated threads. Possible buffering policies are: First In First Out (FIFO) and Oldest Packet Drop (ODP). In the first case, messages are queued in a list that grows and guarantees that no messages are dropped. In the second case, the size of the queue is fixed, and new messages overwrite old ones to guarantee minimum latency. Read operations in a *BufferedPort* can be blocking in case we want execution to wait for incoming messages. Publish subscribe is convenient for one-way communication. In some cases, however, communication requires replies. In YARP, this is called RPC and is supported *via* two specialization of the *Port class*: *RPCServer* and *RPCClient*, respectively, for managing the server and client side of the communication.

In the iCub software, architecture components are runnable pieces of software (usually implemented as executables, but sometimes also as software drivers) that export a certain interface using one or more *Port* (or *BufferedPort*) objects. The component sends and receives data through its *Port* objects; depending on the type of service provided by the components the port can be configured for synchronous, asynchronous or RPC operations.

YARP is multi-platform and implements all its functionalities on Linux, Windows, and MacOS. With respect to other robotics middleware-like OROCOS (Bruyninckx, 2001), Player (Collett et al., 2005), and ROS (Quigley et al., 2009), the communication layer of YARP supports a richer set of protocols. A notable example is the multicast protocol, which reduces bandwidth and transmission overhead in case of one-to-many communication [which to the best of our knowledge is implemented only by LCM (Huang et al., 2010)]. More importantly, YARP implements a plug-in system that allows users to write custom protocols and interconnect it with other systems (like cameras providing images in mjpeg format or a web server). This mechanism was used to add ROS compatible protocols [i.e., *tcpros*, *xmlrpc*, see Fitzpatrick et al. (2014) for more details], port monitor (Paikan et al., 2014a) and to add QoS and channel prioritization, as described in Section 3. The Thrift IDL provides a language for defining component interfaces that is more flexible than the one that can

[2] http://www.corba.org/
[3] https://zeroc.com/products/ice
[4] https://www.freedesktop.org/wiki/IntroductionToDBus/
[5] http://zeromq.org/

be implemented using ROS's services. Similarly to Player (Collett et al., 2005), YARP provides interfaces for hardware devices, and, in particular, a sophisticated hardware abstraction layer for motor control, as described in Section 4. These interfaces simplify the control of robots hiding complexity due to the underlying communication layer and vendor-specific APIs.

Communication performance of various middleware has been compared experimentally in Hammer and Bäuml (2014). In this study, YARP demonstrated comparable – and sometimes even faster – performance than ROS. Interestingly, round-trip time of both middleware was consistently better than OROCOS. It is worth noticing that YARP was not designed to support hard real time. The recent QoS and channel prioritization extensions partially cope with this, providing significantly performance improvement in terms of communication and scheduling jitter, especially when adopted together with the RT-PREEMPT Linux kernel. For applications that require lower latency and higher determinism OROCOS (Bruyninckx, 2001) and aRDx (Hammer and Bäuml, 2014) may be a preferable choice. It is worth noticing that OROCOS offers a YARP compatible transport that can be used for interoperability with YARP applications.

YARP is written in C++, but similarly to ROS, it can be used from other languages. Through SWIG,[6] YARP provides language bindings for many languages, namely: Perl, Python, Ruby, Lua, TCL, C#, Java, Octave, Chicken, and Alegro. Java also provides seamless integration with the Matlab environment, although, to achieve better performance, we have recently started developing support for Matlab and Simulink *via* mex functions. YARP was interfaced with Android and iOS devices.

Compilation of YARP is simple. YARP can be compiled under various operating systems, including major distribution of Linux (Debian and Ubuntu), MacOS X, and Windows.[7] Dependencies have been voluntarily kept at minimum. The main dependency is ACE[8] which is – by design – a portable library that can be compiled on a plethora of systems. The build system automatically detects optional dependencies and disables features or plug-ins accordingly. To simplify compilation with Linux, we decided to support a set of distributions and to use features available only on the libraries provided therein. This greatly simplified compilation and distributions of binaries for the Linux system. Compilation on Windows was made more complicated by the lack of a proper packet management system. For this reason we decided to build and distribute precompiled binaries of the required dependencies for all the supported compilers. For compilation on MacOS X, we rely on *homebrew*[9] and maintain appropriate *recipes* scripts for both YARP and the iCub main software. To ensure correct compilation of the software on all the supported platforms, a compile farm performs compilation tests, periodically and upon any commit to the YARP and iCub main repositories.

[6]http://www.swig.org/

[7]At the time of writing YARP (and the iCub main software) were supported on Debian 7-9 and Ubuntu 14.04-16.04. On Windows supported compilers were Visual Studio 10-12. Support for MacOS included the latest release 10.11, codename *El Captain*.

[8]http://www.cs.wustl.edu/~schmidt/ACE.html

[9]http://brew.sh/

TABLE 1 | YARP libraries footprint.

Library	Footprint (KB)
libACE	1673
libYARP_OS	2088
libYARP_sig	244
libYARP_dev	94
libYARP_math	1530

See text for details.

To conclude this section, we report additional information about YARP. **Table 1** reports the footprint of the YARP libraries. The table reports the value obtained running the Linux command *size*. These values corresponds to YARP 2.3.64 compiled in "Release" mode, gcc version 5.3.1, libc 2.21, CMake 3.4.1, libACE 6.3.3, and libGSL 2.1 (the latter is optional and it provide signal processing and linear algebra routines to YARP). Notice that libACE in the Linux environment is optional and is required only for compilation on Windows. YARP is adopted by a large number of people. The iCub community consists in approximately 30 teams. YARP is also adopted on the COMAN (Tsagarakis et al., 2013) and in the projects Walkman,[10] DREAM (Vernon et al., 2015), and Fireswarm,[11] while OROCOS includes a YARP compatible transport protocol. YARP received contributions from 75 developers in total and from 29 developers in the past 12 months (source: Open HUB[12]).

3.1 Logging

YARP provides macros that allow users to log messages with increasing levels of importance and severity (i.e., trace, debug, info, warning, error, and fatal). These macro print on the standard output using the facilities provided by the host operating system. To implement the logging system we follow these guidelines: (i) in a distributed architecture messages should be collected from different machines, (ii) logging should be optional to avoid using unnecessary resources and finally, (iii) it should be possible to collect output from components that have been written without YARP. These features have been achieved by relying on the *yarprun* service. The latter is a software service that is used to execute components remotely using a GUI (the *yarpmanager*, as described in Section 8). *yarprun* spawns processes and can therefore manipulate and redirect their output. This offers a simple way to log the output of all components without modifying their code or forcing adoption of YARP specific macros for logging. *yarprun* prefixes all messages from a component with the name of the machine on which it is running and the process identifier of the component itself. All messages are redirected to common recipient using YARP *Ports*; the recipient can either be a command line tool or a graphical interface that allow logging, filtering, and visualization of these messages. This solution allows capturing and logging the output of components even when they do not use YARP's logging functions (although with limited functionalities).

[10]https://www.walk-man.eu

[11]http://fireswarm.nl/

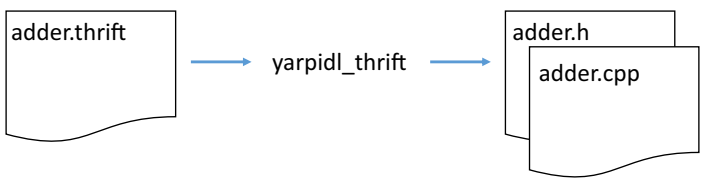
[12]https://www.openhub.net

3.2. The Thrift IDL

A *Port* object can transmit a data type only if proper serialization and deserialization functions are defined. YARP supports a few basic types that already include serialization functions: *Vector*, *Matrix*, *Image*, and *Bottle*. The first three types are self-explaining, they define containers for double precision floating point vectors, matrices, and images with various pixel types. The *Bottle* object is a list of mixed type values: it can store arbitrarily integers, strings, doubles, lists, or binary blobs of memory. This type is quite flexible and can be used to send virtually anything, provided the sender and receiver agree on the data they exchange. The compiler in fact cannot determine if the data sent through a *Bottle* is correctly parsed by the receiver. This can be acceptable for small applications but become soon a limitation, especially when modules are developed asynchronously by different developers. Services are also a concept that is not natively supported by YARP. Services can indeed be implemented with YARP, but the programmer has to manually write all the code required to parse incoming messages and prepare replies. Writing this code is boring and error prone, its maintenance becomes soon complex and difficult.

All these problems have been solved in YARP with the adoption of an Interface Definition Language (IDL) based on the Thrift language. The Thrift IDL can define services. From this definition, the *yarpidl_thrift* compiler generates all the code that implements the communication between the clients and the service across the YARP network. This process is exemplified in **Figure 1**. A new network type can be implemented in a similar way. In this case, the *yarpidl_thrift* compiler generates the .h and .cpp files for the C++ class that implements the type in YARP. The compiler automatically generates serialization and deserialization routines. See **Figure 2** for details.

The Thrift IDL is currently adopted as best practice for defining new types and interfaces for components in the iCub software architecture. An important feature of the Thrift YARP compiler is that it copies verbatim all comments from the Thrift file to the C++ implementation class. This feature is used to document the interfaces using Doxygen. All Doxygen comments are added to the Thrift file(s) that defines the interface of components, when code is generated, these comments are copied in the resulting C++ header files and parsed to produce the documentation. In this way, the documentation of the interfaces is stored with the




```
adder.thrift
service Adder {
      /** Documentation for get_answer*/
      i32  get_answer();
      /** Documentation for get_answer*/
      bool set_answer(1:i32 val)
      /** Documentation for add*/
      i32  add (1:i32 x);
}
```

```
adder.h:
class Adder: public yarp::os::Wire {
public:
       /** Documentation for get_answer */
       virtual int32_t get_answer();
       /** Documentation for get_answer */
       virtual bool set_answer(const int32_t val);
       /** Documentation for add */
       virtual int32_t add(const int32_t x);
};
```

Cient code

```
client.cpp:
#include "adder.h"
int main() {
    Port port;
    Adder client;
    port.open("/client");

    client.yarp().attachAsClient(port);

    client.set_answer(10);
    client.add(1);
    cout<<client.get_answer();
}
```

Server code

```
adderserver.h:
#include "adder.h"
class AdderServer : public Adder {
public:
    int32 _val
    int32_t get_answer()
       { return _val; }
    bool set_answer(const int32_t val)
       {_val=val; }
    int32_t add(const int32_t x)
       { _val+=x; return _val}
};
```

```
server.cpp:
#include "adderserver.h"
int main() {
    Port port;
    AdderServer server;
    server_port.open("/server");

    client.yarp().attachAsServer(port);

    while(true)
      {...}
}
```

FIGURE 1 | Service implementation using Thrift. The definition of the service *Adder* is written using the Thrift IDL (*adder.thrift*); from this file, the *yarpidl_thrift* compiler generates the corresponding C++ interface (*adder.h*) and the full implementation of the client (*adder.cpp*). At the client side, function calls take care of marshaling the parameters and shipping them across the YARP network. At the server side, a *Port* object receives these messages, performs de-marshaling, and invokes the corresponding C++ function. Notice that comments in the Thrift file are preserved in the C++ class. Bottom: the client main function instantiates the YARP *Port* that will be used for the communication, it then instantiates the client object, attaches the *Port* to it and invokes the service functions as if they were local calls. At the server side, the developers provides implementations for the service functions in an new class (*AdderServer*) that overrides the virtual functions defined in *Adder* (*adder.h*); it then instantiates the *Port* that handles the communication and attaches it to an instance of *AdderServer*. At this point, all messages received by the *Port* at the server side are dispatched to *AdderServer*.

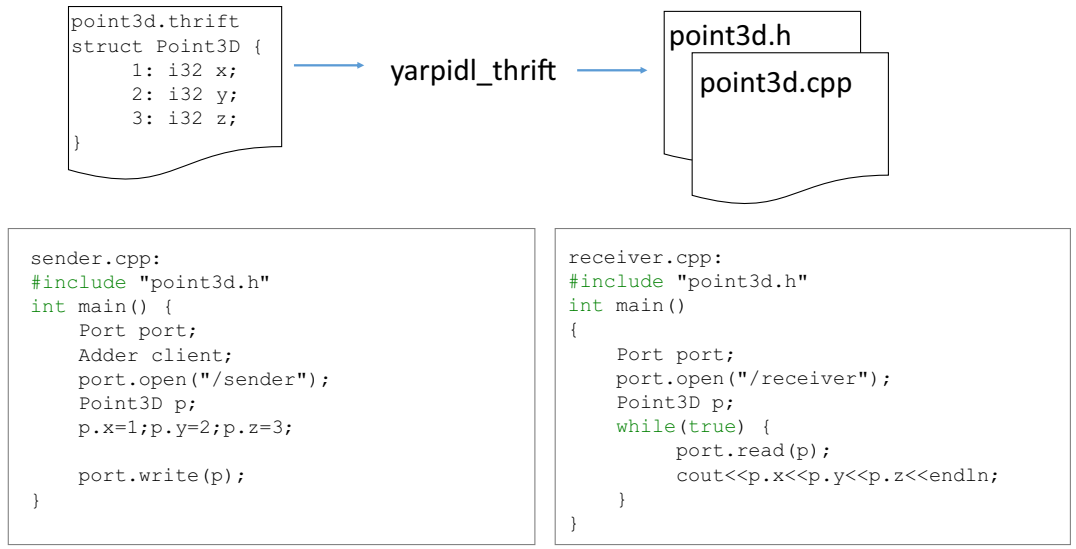


FIGURE 2 | Specification of a new type using Thrift. The *yarpidl_thrift* compiler generates the C++ object that implements the type *Point3D* starting from its specification in the Thrift IDL. Bottom: this object contains serialization and deserialization functions and can be used directly for read/write operations in a YARP *Port* or *BufferedPort* (not shown).

code that generates them; this results in better documentation and easier maintenance.

A recurring pattern in the development of software on the robot is the following: A module defines a set of inner parameters that modify its behavior and exports them through a *Port*, together with a set of functions for manipulating them. Thrift allows defining a structure for grouping parameters so that YARP can provide support for reading and writing this structure through a *Port*. In addition to this, YARP generates an object called *Editor*, which provides methods for setting and getting individual values within the structure as well as callbacks that can be customized to execute code before and after the value is modified. This feature reduces the amount of code that is manually written and maintained when writing interfaces for modules, which, in robotics applications usually consists in several parameters. The *Editor* has been introduced only recently in YARP but is currently adopted as best practice when writing new modules. **Figure 3** illustrates this concept in more details.

3.3. Increasing Determinism in Distributed Applications

Many robotic applications require real-time functionalities, especially when timing constraints on task execution, data processing, and synchronization are crucial. In distributed architectures, extra care must be taken to avoid mutual interference between components in the communication layer. The YARP middleware deals with this problem by providing functionalities at the level of the connections (in the *Port* and *BufferedPort* objects) that allow assigning different priorities to individual communication channels (we call this approach "channel prioritization") (Paikan et al., 2015b). This approach simply leverages the operating system functionalities to prioritize specific communication channels between publishers and subscribers.[13] The properties of individual connection channels are extended to specify (i) the priority level and scheduling policy of the threads that handle the communication and (ii) the priority of the packets on the network (i.e,. the network Quality of Service parameters). This approach does not require specific components for message prioritization and it does not add any overhead to the communication. In addition, and, more importantly, it allows for remote configuration of Quality of Service (QoS) and for run time, dynamic prioritization of communication channels. Configuring real-time properties such as priority or scheduling policy of the user thread can be done either programmatically from the user code or automatically using component middleware functionalities and dedicated tools (Mastrogiovanni et al., 2013).

When configured for asynchronous communication *Port* objects send and receive user data in separate threads. A conceptual example is depicted in **Figure 4**, where a publisher (Publisher 1) pushes data to two subscribers. When a publisher writes data to a *Port*, it passes it to the corresponding thread. At this point, execution is determined by the operating system and the thread real-time properties decide with which priority the thread will manage to write data to the socket. Similarly at the receiver side, real-time properties of a thread affect the chances that it will deliver data to the user (i.e., high priority will reduce jitter). This is useful when a subscriber receives data from multiple senders, and the application requires to assign higher priority to one of them.

Each connection in YARP has a state that can be manipulated by external (administrative) commands, which in turn manage

[13]Notice that in this section we refer to "best-effort" or "soft" real-time, as opposed to "hard" real-time performance. Because the implementation of YARP has not been developed to support hard real-time constraint, we rely on the RT-PREEMPT Linux to reduce scheduling latency.

```
receiver.cpp:
#include "MyPoint3d.h"
int main() {
     Port port;
     Point3D p;
     Point3D::Editor editor(p);
     port.setReader(editor);
     port.open("/receiver");
     while (true) {
      printft("%d %d %d\n",p.x p.y p.z);
      /* wait some time */
     }
}
```

```
MyPoint3d.h
#include "Point3d.h"
class MyPoint3d: public MyPoint3d  {
public:
      bool will_set_x()
       { prinf "about to set x"; }

      bool did_set_x()
       { printf "finished setting x"; }
};
```

```
$ ./receiver
yarp: Port /receiver active at tcp://127.0.0.1:10001
yarp: Receiving input from /writer to /receiver
0 0 0
about to set x
10 0 0
finished setting x
```

```
$ yarp write /writer /receiver
yarp:Port /writer active at tpc://127.0.0.1:10002
set x 10
...
```

FIGURE 3 | **The *Editor.*** For each new type, the Thrift compiler generates an Editor object, which provides methods for manipulating only a subset of the type. The listing on the top-left shows how to use the *Editor*. An instance of the *Editor* is connected to *Point3D* (see **Figure 2**) and to the module's *Port* object (*i.e. port*). Messages received *by port* are parsed by the *Editor*, which decodes and handles them appropriately. Top-right: optionally, the user can define call-back functions that are invoked before and after data is modified. Bottom: we show how the value of the field *x* in *Point3D* can be modified by sending text messages to a *Port* (left: output of *receiver*, right: text commands sent to the *receiver*'s *Port* using the standard tool *yarp write*).

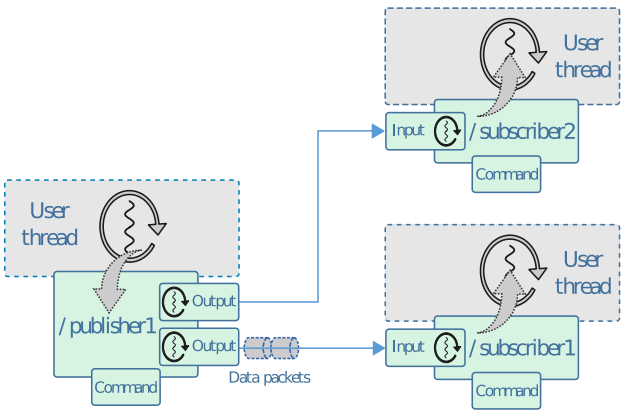


FIGURE 4 | **An example of components asynchronous communication in YARP**. A publisher is pushing messages to two different subscribers using separate dedicated threads.

the connection and/or obtain state information from it. Using *Port* administrative commands, QoS and real-time properties of *Port* objects can be configured with the granularity of individual connections. In the current implementation, the *Port* administrator provides two set of commands that affect the priority of a communication channel: setting the scheduling policy/priority of a communication thread and configuring network QoS parameters (i.e., the TOS/DSCP bits) for the data packets it delivers.

In the example from **Figure 4**, we can configure the real-time properties of the channel that links/*publisher1* to/*subscriber1*:

```
$ yarp admin rpc/publisher1
> > prop set/subscriber1 (sched
((policy SCHED_FIFO)
(priority 30)))
```

The first line "*yarp admin rpc*" simply opens an administrative session with the *Port* object of /*publisher1*. The second line is the real command to the administrative *Port*. It adjusts the scheduling policy and priority of the thread in/*publisher1*, which handles the connection to /*subscriber1* respectively to SCHED_FIFO and 30 on Linux machines.[14]

Ip networks define four classes of services (Almesberger et al., 1999). These classes are selected so that packets can be treated similarly by the OS queuing policy (if available) and in the network switch. For example, a packet with priority class `Low` will be in the lowest priority band (Band 2) of the Linux queuing policy and will have the lowest priority in the network switch. Analogously, data packet priority can be configured *via* administrative commands by setting one of the predefined priority classes:

```
$ yarp admin rpc/publisher1
> > prop set/subscriber1 (qos ((priority HIGH)))
```

This simply sets the outbound packets priority to HIGH for the connection from /*publisher1* to /*subscriber1*.

These parameters can be set for every channel in the same way and jointly define the actual priority of a communication channel

[14]The thread scheduling properties and policies are highly OS dependent and a proper combination of priority and policy should be used.

in our publish/subscribe architecture. Alternatively, real-time properties of the communication channels can be configured from the user code using the YARP API:

```
QosStyle style;
style.setThreadPolicy(SCHED_FIFO);
style.setThreadPriority(30);
style.setPacketPriorityByLevel
(QosStyle:PacketPriorityHigh);
NetworkBase:setConnectionQos("/publisher1", "/subscriber1",
style);
```

We have experimentally demonstrated that channel prioritization significantly reduces latency and increases communication determinism in presence of conflicting connections in robotic applications (Paikan et al., 2015a,b).

4. ROBOT INTERFACE

The interface to the sensors and motors of the robot is implemented by a set of user-level drivers that access the hardware using vendor-specific API. On the iCub, cameras use a IEEE1394 Firewire bus whereas the majority of the other sensors use custom electronics connected *via* can bus or, in recent versions, Ethernet. These devices use custom protocols, whose details are not important for this paper. For such devices YARP defines a set of C++ interfaces that provide an abstraction layer that is independent of the specificities of the hardware components that implement them. Communication with the hardware is achieved by instantiating user-level devices, which directly send messages to the hardware. These devices implement a set of interfaces that allow reading sensor values and controlling the motors at the joint level. A second layer of devices can be instantiated and connected to these (*via* a function called *attach*) to implement functionalities like robot calibration, on-board control loops, and network remotization (see **Figure 5** for more details).

The life-cycle of all objects is decoupled: all instances are created independently and references to drivers are passed to higher layers using the function *attach*. The opposite operation is performed by calling a function called *detach*, which removes all references held by a device before shutting down and releasing memory. In the development of the iCub interfaces have played an important role because they have preserved the user code in face of deep changes in the hardware. The code required to perform basic functionalities like reading images or controlling the motors in position or velocity modes has not changed significantly while

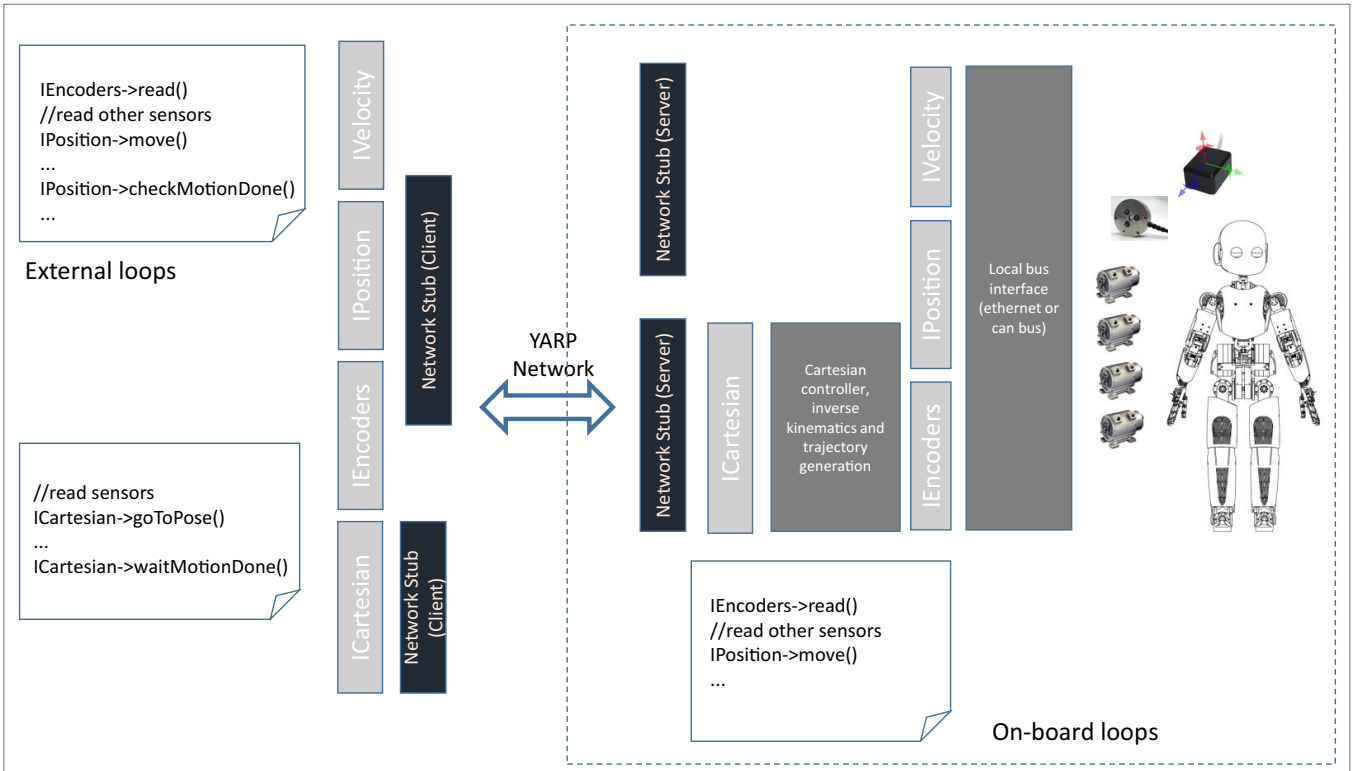


FIGURE 5 | **Interfaces to the robot**. User-level drivers communicate to the hardware using device drivers. These objects (dark gray) implement a set of standard interfaces (light gray) for reading sensors and controlling the motors at the joint level. Other controllers can be connected (*via* an action called *attach*) to these drivers to implement control loops (like for example Cartesian control of the limbs). Special network stubs (black) export the functionalities of the drivers through the YARP network and allow users to execute control loops externally to the robot. To do so, the user instantiates clients network stubs that implement the same interfaces by dispatching requests through the network using YARP *Ports*. These messages are received by the remote servers, which translate messages *via* function call to the hardware device using the user drivers. If needed, the server also takes care of preparing the reply and sending it to the client, which, finally, returns control to the user code.

the robot underwent subsequent revisions of the electronics and improved with the addition of new sensors and capabilities like impedance or force control. The set of robot interfaces are generic and have been implemented various simulators, thus allowing switching control of simulators and the real robot at no cost (Tikhanoff et al., 2008; Mingo et al., 2014; Habra et al., 2015).

The startup of the robot consists in running a single executable called *robotinterface*. This executable reads a configuration file that specifies the list of drivers to be instantiated and their parameters. In addition, the configuration file may specify a list of special objects that perform specific operations using the drivers. The first example is the network stubs objects that provide remote communication. Other objects are calibrators and controllers. Calibrators are objects that implement routines for calibrating the joints and bringing them in a home position (parking). Controllers are objects that implement control functionalities using low-level drivers; an example of this is the Cartesian Controller, which implements on top of joint-level controllers the functionalities required to control the arm (or the head) in Cartesian space. These controllers are time-critical and safety-critical and therefore must communicate with the low-level drivers with minimal latency using function calls (and avoiding network communication).

To allow robot configuration and shutdown *robotinterface* implements, a state machine that can be configured to execute custom activities with a predefined order. **Figure 6** describes the finite state machines. Immediately after execution, *robotinterface* instantiates all objects passing the required parameters (usually specified in a XML file); it then enters the *startup* phase. The latter is further configured to execute custom actions with a specific order. These actions include calibrating the robot and invoking *attach* functions to configure high-level objects. These operations are executed in reverse order to park the robot and uninitialize the objects during *shutdown*. *robotinterface* also defines states that are triggered upon reception of termination signals to interrupt or abort on-going operations. The left side of **Figure 6** illustrates the finite state machine and provides further details. An example of a typical execution of *robotinterface* in a typical run is reported on the right in **Figure 6**. *robotinterface* opens a YARP *Port* that can be queried to know the its state of execution of *robotinterface*. This functionality was added recently to allow modules to make sure that the robot is functional and synchronize their execution with the termination of the *startup* phase.

As we discussed above, the robot interfaces are generic and can be implemented for robots other than iCub. All objects that

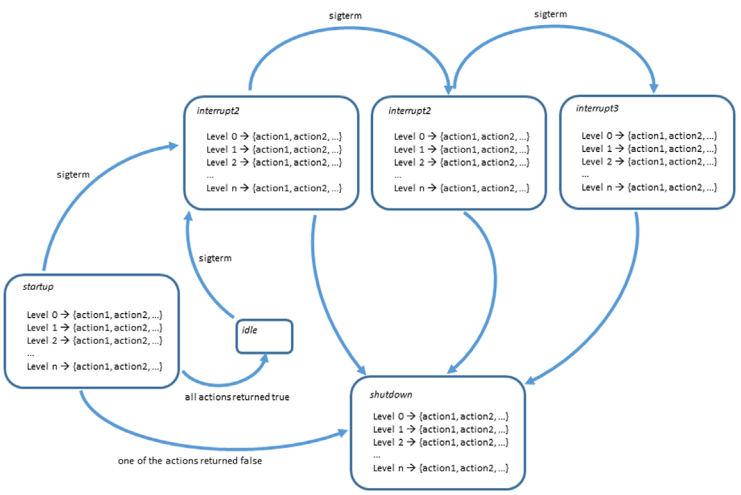


FIGURE 6 | State machine of *robotinterface*. Left: We defined a set of *phases* that are executed upon activation of certain events, namely: *startup*, *interrupt1*, *interrupt2*, *interrupt3*, and *shutdown*. Each phase is further separated in a set of numbered *levels* each linked to a specific set of *actions* to be executed. When entering a given phase *robotinterface* executes all the actions that correspond to the levels that have been configured. Actions at the same levels are executed in parallel in different threads; execution waits for the termination of all threads before proceeding to the next level. In the configuration files, each object can specify one or more actions to be executed, when to execute them (i.e., the *phase*) and in which order (the *level*). The *startup* and *shutdown* phases are executed, respectively, at the beginning and before *robotinterface* terminates execution. The phases *interrupt1, interrupt2, interrupt3* are entered upon subsequent reception of the termination signal (in Linux it is the *SIGTERM* signal) and are followed by the *shutdown* phase. During *startup robotinterface* configures the devices. All network stub objects and controllers register an *action* called *attach* specifying the device they will use to perform their tasks. This is usually done at level 1, i.e., just after all devices have been created. They also register the action *detach* in the phase *shutdown* (level 5, i.e., after the robot is parked). The calibrators objects register actions *calibrate* in the *startup* phase. This is done at level 10, after all *attach* are executed. Calibrators also register the action *park* corresponding to phase *interrupt1* (level 1) so that the robot is brought to the homing position upon reception of the first termination signal. The calibrators also support an *abort* action that is executed to abruptly stop any on-going parking or calibration routines after the third termination signal is received. This action is registered for execution at level 1 in the *interrupt3* phase. The objects that perform Cartesian control are instantiated during the *startup* phase and execute *attach* to get an instance to the joint-level controllers. Proper operation of the Cartesian controllers requires that calibration is performed and that all network objects are instantiated; the *attach* action for these objects is therefore scheduled at level 15. Symmetrically, their disconnection (i.e., *detach* action) is scheduled early on during *shutdown*, i.e., at level 2. Right: Main steps of execution of a typical run of the iCub robot.

are instantiated by *robotinterface* have been implemented using the YARP plug-in system. This means that *robotinterface* is not statically linked against any of the libraries or device driver's API that are required to operate the hardware devices. All the objects instantiated by *robotinterface* are contained in dynamic libraries that are loaded at runtime and can therefore be compiled, maintained, and distributed separately for each robot.

4.1. Motor Control Interfaces

During the development of the robot the motor interfaces underwent several revisions. To the original position and velocity control interfaces, we have added interfaces for open-loop, torque, and impedance control. In this paper, we revise the control modes implemented on the iCub and the corresponding interfaces. Full specifications can be found in the iCub control modes specifications document (Randazzo, 2004) and online in the YARP documentation of control board interfaces.[15]

An important concept in iCub is the fact that high-level components need to be aware only whether the robot can receive position, velocity commands, or open-loop commands controlling the reference for the controller directly. However, they do not need to know how the low-level controllers implement these functionalities. This is achieved by separating control modalities in groups and by defining a special control mode called *interaction mode*. The interaction control mode determines whether the low-level controls PWM with a PID (*stiff interaction*) or the torque in closed-loop (*compliant interaction*). Stiff interaction is the conventional control mode of industrial robots, which are required to execute accurate position/velocity trajectories in controlled environments. In compliant interaction mode, instead, it is possible to control the joint impedance (i.e., stiffness and damping) during the execution of position or velocity commands. To simplify usage many of the interfaces provide functions for controlling individual, all or only a subset of the joints in a kinematic chain.

IPositionControl and *IVelocityControl* define the simplest form of control for the robot. With *IPositionControl*, the controller receives a new reference position, and it generates a trajectory that smoothly interpolates the current and desired state of the robot (position and velocity). On the iCub, the trajectory generator produces velocity profiles that follow the minimum jerk principle. In addition, the interface allows setting the acceleration and velocity values that will be used to generate the trajectory. *IVelocityControl* requires joints to move with a certain speed. The controller accelerates the joints (using a user-defined value) until it reaches the required speed. This control paradigm is suitable for visual servoing, typically for controlling the end-effector or the robot gaze using vision.

IPositionDirect has been introduced mode has been introduced lately to accommodate specific research requirements. It allows skipping trajectory generation and immediately setting the reference value of the position controller. This modality allows generating custom trajectories in small incremental steps.

ITorqueControl sets the torque exerted by a motor. This control mode requires that force feedback is available and that a proper torque loop is implemented. Finally, the *IOpenLoopControl* interface allows to by-pass all controllers and set the PWM reference of the motors directly. This interface is used mostly for fast prototyping control algorithms or for identification. An example is the implementation of control algorithms that use and estimate the parameters of the motors (i.e., back-emf, friction etc.).

The interfaces described above provide the first layer of control at the joint level. Higher level interfaces have been defined for controlling kinematic chains in Cartesian space either in position or orientation (Pattacini et al., 2010). These interfaces separate the robot in different kinematic chains that are controlled independently. Research is today progressing further and new work is currently being done to coordinate whole-body movement for balancing and locomotion (Nori et al., 2015). To support this research, a specific interface for whole-body control is currently being developed.

The state of the robot is available through interfaces that expose joint encoder values (*IEncoders*), motor currents (*IAmplifierControl*), and allow setting and getting control modes (*IControlMode*). An important improvement in the latest revisions of the iCub has been the introduction of additional high resolution encoders that measure the position of the motor shaft (rotor). To give the user access to this additional information, we introduced a new interface (*IMotorEncoders*). Other sensors like F/T, tactile sensors, and IMU are mapped into a generic interface for analog sensors (*IAnalogSensor*), which gives methods for reading the most recent sensor values and perform calibration by setting the zero.

Interfaces are a powerful abstraction. However, getting access to the hardware solely through generic interfaces may be restrictive because a developer often needs access to functionalities that are hardware specific. This usually happens for testing and debugging especially when new functionalities are added to a robot. In this case, there is a big pressure to extend interfaces to make them accessible to the higher levels, and this forces premature design choices and unnecessary code refactoring. For this purpose, we defined a new interface (*IRemoteVariables*) that gives access to generic variables identified with a string of text. This interface defines methods to get the list of available variables as well as methods for setting and getting them individually. This interface can be used to read and manipulate quantities inside the memory space of the control boards. It can be conveniently used for monitoring or changing the internal state of a board for testing, debugging, and code fast prototyping.

4.2. Remotization

As explained in the previous section, YARP provides special objects that remotize interfaces across the network. This is achieved using network objects that come in client-server pairs (identified as network stub in **Figure 5**). The client is loaded locally in the user code; it converts function calls into messages that contain all the parameters and dispatches them across the network to the remote server. Communication is done using three *Port* objects: one for RPC and two for unidirectional communication to and from the server with reduced latency.

[15] http://www.yarp.it/namespaceyarp_1_1dev.html

The RPC *Port* dispatches all the function calls that require a reply and that are not time-critical. Examples of such functions are: getting or setting the PID and changing control modes. Notably RPC also handles commands for moving the joints in position mode with trajectory interpolation. This is because when sending requests to the server, the client always waits for an acknowledgment message. This prevents flooding the server or the network with requests and ensures that no commands are lost. This is not a problem because the additional delay is negligible with respect to the typical time requested by a joint to complete a trajectory.

The other interfaces for joint control (*IVelocityControl*, *ITorqueControl*, *IOpenLoopControl*, *IPositionDirect*) send data to the server using *BufferedPort*. In this case, buffering policy ODP avoids that latency is accumulated in the control loops. Finally, the state of the robot is collected by the server and broadcast periodically using another *BufferedPort*. The client stores this message as soon as it is received and propagates its most recent content upon request by the user (thus avoiding explicit requests). This strategy is convenient because it avoids the need for the client to perform remote requests, and it reduces latency; for this reason the state of the robot has been extended to include not only motor encoders (as it was initially) but other variables like motor currents, speed, acceleration, torque, and status flags. Using the functionalities described in Section 5, the *BufferedPort* that broadcasts the state of the robot can be configured as a ROS topic that publishes the "common" ROS joint state message (*sensor_msgs/jointState.msg*[16]). This allows better interoperability with ROS, for example, using ROS visualization tools like the popular *rviz* GUI.[17]

5. USING THE iCUB SOFTWARE WITH OTHER ROBOTS

Recent efforts have been devoted to provide functionalities at the level of the middleware to interoperate the iCub software with other robots. This was achieved in two ways: (1) by extending YARP so that it provides compatibility with the ROS middleware and (2) by extending YARP *Ports* so that they can be dynamically configured to execute code that manipulates input and output data. The second mechanism was adopted to interoperate iCub with the ARMARX software system. As the latter approach has been already described elsewhere (Paikan et al., 2015c), we provide here more details on the extensions that allow YARP to interoperate with the ROS middleware.

ROS is today the most popular middleware for robotics. Similarly to YARP, it supports developing software architectures based on the publish-subscribe paradigm. A distinguishing feature of ROS is that it requires the user to define all types that are transferred on the network with an IDL. Thanks to this ROS can statically and dynamically check that all the parties involved in a communication expect the same type. Like YARP, ROS uses a central name server which, optionally, can store parameters. To communicate with ROS, YARP had to be extended with protocols that allow communication between the name servers and with the ROS topics. More importantly, the type system of YARP had to be extended to support translating ROS data types in YARP compatible structures.

YARP can be configured to use the ROS name server (*roscore*). This is the simplest solution for ROS users, although it implies that some functionalities are not available in YARP (for example the multicast protocol). Alternatively, the YARP name server (*yarpserver*) can be configured to talk to *roscore* and to propagate queries and topic registrations from/to the ROS. We decided to implement both solutions because each addresses the needs of the YARP and ROS communities.

YARP can register nodes. This can be done in code using dedicated functions in the API or dynamically by using a special syntax when registering a YARP *Port* (for example, the *Port* name */reader@/chatter* creates and associates a topic called */reader* to a node called */chatter*).

YARP *Ports* can understand messages coming from ROS topics or generate valid ROS messages. To do so, YARP needs to be aware of ROS types. We identified two application scenarios. In one case, the user has both YARP and ROS installed. This is the easier case because YARP can read types directly from the ROS installation, and using a compiler (i.e., *yarpidl_ros*), it can generate appropriate data structures. In the second scenario, ROS is not installed on all machines that are running YARP. For this situation YARP provides a type server that can answer queries at run time and provide to YARP programs the information they need to interpret ROS messages.

As an example, **Figure 7** shows the code required to control ROS's *turtlesim* from YARP. Similarly, **Figure 8** shows how to dynamically connect an existing YARP *Port* to a ROS topic without recompiling the code.

6. COMPONENT REUSABILITY AND COORDINATION

Developing high-quality reusable software components requires careful design that strikes a good balance between potential reuse, functionalities, and ease of implementation (Sametinger, 1997). Coordination of software components in a distributed architecture usually adds considerable overhead to the robotic application design process and, often, pulls the development of software components in a specific direction. This can have a negative impact on the development process and can reduce reusability of software components.

Software should be extensible enough to be adapted to possibly unanticipated changes (Zenger, 2004). One direction to extend a module is *via* its interfaces. In YARP, interfaces are implemented by exchanging messages through the middleware connection points (*Ports*). To enhance the reusability of the iCub software, we extended the *Port*'s functionality so that it can dynamically load and execute a run-time script. In our framework, this port extension is called *Port Monitor*: in brief, it allows accessing data passing though a connection from/to the *Port* for monitoring, filtering, and transforming it. Multiple instances of *Port Monitor* can interact to allow an input *Port* to select data from multiple

[16]http://wiki.ros.org/common_msgs
[17]http://wiki.ros.org/rviz

```
#include "Pose.h"
#include "Twist.h"
/* create ROS Node /controller */
yarp::os::Node node("/controller");
/* create a subsriber for Pose.msg */
yarp::os::Subscriber<Pose> pose;
/* subscribe to /turtle1/pose  */
pose.topic("/turtle1/pose");
/* create a publisher for Twist.msg */
yarp::os::Publisher<Twist> cmd;
/* publish to /turtle1/cmd_vel */
cmd.topic("/turtle1/cmd_vel");
while(true) {
      Pose p;
      /* read a new value from the topic */
      pose.read(p);
      cout<<p.x<<" "<<p.y<<" "<<p.theta<<"\n";

      /* prepare a command for the turtle_sim */
      Twist t;
      t.linear.x=1.0;
      t.angular.z=1.0;

      /* publish the command */
      cmd.write(t);
}
```

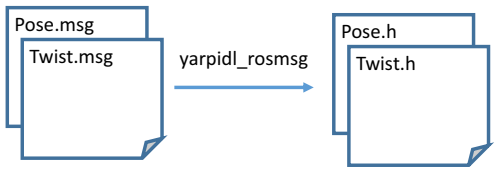


FIGURE 7 | Interfacing YARP with ROS's topics. The figure illustrates the code required to interface YARP with the *turtlesim*. Starting from the ROS messages definition in *Pose.mgs* and *Twist.msg* the *yarpidl_rosmsg* tool produces compatible YARP types, *Pose.h* and *Twist.h*. The code creates a ROS compatible node, a subscriber and a publisher, reads data that contains the state of the *turtlesim* from the topic */turtle1/pose*, and sends velocity commands to the topic */turtle1/cmd_vel* to control it and sends velocity commands to control it.

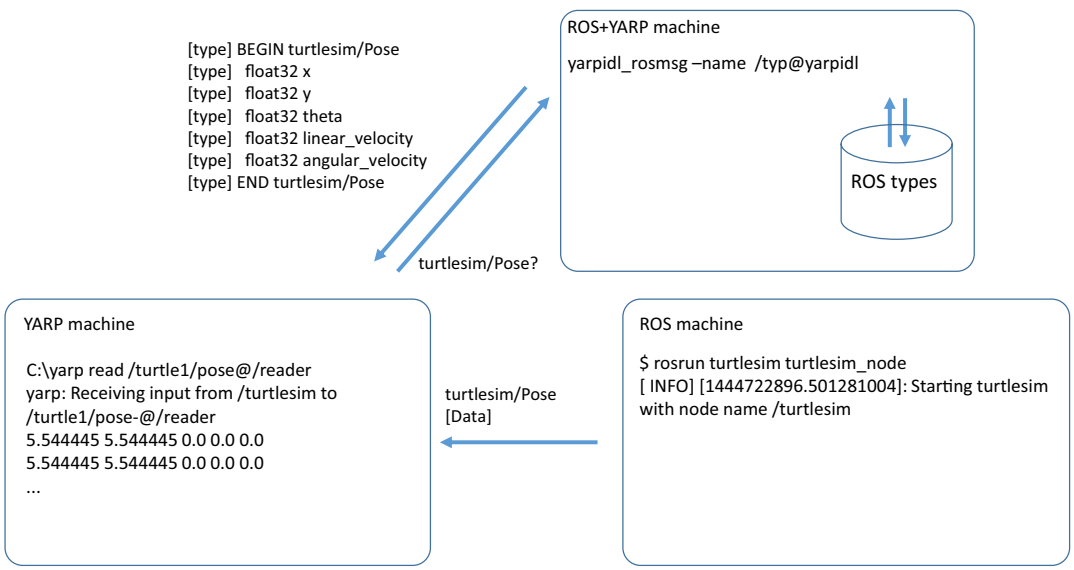


FIGURE 8 | Reading a ROS topic from an existing YARP *Port*. This figure shows how to read the output of the topic */turtle1/pose* from an existing YARP *Port*. On the left, we run the *yarp read* tool, which opens a *Port* and prints to console the data it receives. To make ROS aware of this *Port*, we decorate its name using a special syntax that instructs YARP to create a ROS node called */reader* and subscribes it to the topic */turtle1/pose*. To properly interpret the type *turtlesim/Pose* YARP queries the typeserver. Notice that all processes are running on different machines and that, in this case, the machine that runs *yarp read* knows nothing about ROS. To simplify the figure, we omit all the handshakes required for naming look up and establishing the connections. The opposite operation (writing from a YARP *Port* to a ROS topic) can be done in a similar fashion.

sources in an exclusive way. We call this mechanism *Port arbitration*: it allows coordinating components by specifying arbitration rules in the input port of a component (Paikan et al., 2014b).

6.1. Port Monitoring and Arbitration

Figure 9 (left) represents represents the situation in which the *Port Monitor* (shown as a box with M) is attached to the output

of the Face-Detector module and the input *Port* of the Head-Control module. The *Port Monitor* can load a script file [written using a standard scripting language such as, in our case, Lua (Ierusalimschy et al., 1996)] and can access and modify the data traveling through the *Port* using a simple API. This idea allows adding extra functionalities to a component like data filtering, transformation, and monitoring without modifying or rebuilding it (Paikan et al., 2014a).

As an example, the code below illustrates the pseudo-script in Lua that filters messages from Face-Detector when its confidence level drops below a defined threshold (in this case 0.8):

```
PortMonitor.accept = function(data)
  -- read face_pos from 'data'
  if face_pos.certainty <0.8 then
     return false
  end
  return true
end
```

Using the *Port Monitor*, an *input Port* can be configured to arbitrate data from multiple sources, based on user-defined constraints. **Figure 9** (right) represents a simple application where a humanoid robot looks around in search of a person's face and then tracks it. This is a common coordination problem, which can be solved in different ways (e.g., using a separate coordinator or by extending modules to interact with each other). One way to achieve this is to use a selector in the input *Port* of the Head-Control module and constrain it to receive data from each module under specific conditions. The concept is shown in **Figure 9** (right) where the *Port Arbitrator* is used in the input *Port* of the Head-Control (shown as box labeled with two "M"). The arbitration logic can be written using the Lua scripting language and is loaded by the *Port Arbitrator*.

In our approach, a *Port Monitor* is attached to each connection that delivers data to a *Port*. The *Port Monitor* analyzes the data it receives and produces (or removes) events from a container. A set of constraints in boolean logic determines from the events and for each connection if data is allowed to be delivered to the component (otherwise it is discarded). Arbitration is achieved because only one connection at the time is granted the permission to deliver data to the component. This type of arbitration mechanism can be effectively used to implement complex tasks without resorting to centralized coordinators (Paikan et al., 2013).

7. CONFIGURATION OF COMPONENTS

Components support re-use by exposing a set of parameters that affect their behavior. Parameters that are specified at the command line configure the component when it is executed.

Components choose the name of the *Ports* that they use. Because different *Ports* cannot have the same name, component reuse can be achieved only if components provide a way to change the name of the *Ports*. This is so fundamental that YARP enforces it *via* the environment variable *YARP_PORT_PREFIX*: if specified this variable defines a prefix that is added to all the *Ports* within a component (similarly ROS provides a way to remap topic names at the command line: in YARP, this solution was not viable because YARP does not monitor command line parameters).

All other configuration parameters should be specified at the command line, either explicitly or using configuration files. YARP uses a directory hierarchy to organize these files. One problem we had to solve was how to allow users to add configuration files to the ones already existing and how to package them in binary distributions. To support packaging, YARP defines a set of OS-dependent default locations for files and a set of rules that define the search priorities. To provide users with the freedom to install the software on custom directories, these default locations can be overridden or extended by modifying certain environment variables. YARP provides an helper class that allows organizing and locating configuration files, this class is called *ResourceFinder* because it allows managing all types of configuration files (called *resources* in YARP terminology) for a component.

The design of the *ResourceFinder* follows the rationale adopted in the Linux OS, i.e.,

- The software installation should be able to provide reasonable defaults for configuration files so that applications can run out-of-the-box;
- Installation directories are generally non-writable, users without root privileges cannot edit installed configuration files unless they first copy them inside their own private directories, the latter must take precedence and hide the others;
- Therefore it is normal that configuration files can be in multiple places, inside user-specific, private or shared installation directories;
- External packages can install files so that YARP can find them;
- Files are organized in families that are placed in specific sub-directories.

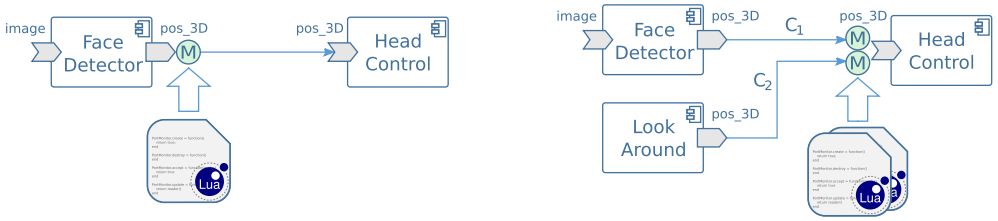


FIGURE 9 | Conceptual representation of *Port Monitor* (left) and its use for arbitration (right). Left: the output *Port* of Face-Detector is extended with a plug-in, which provides access to the outgoing data through scripting language (e.g., Lua). Right: at the input *Port*, the *Port Monitor* can arbitrate data from multiple connections.

Resource files belong to the following families: *modules, applications*, and *plugins*. *Modules* and *plugins* are text files describing modules and plugins (i.e., *manifest* files). *Applications* are files required to instantiate, configure, and connect components usually to achieve a certain task (called *application*). Configuration parameters for components are organized as key-value pairs and stored in one ore more configuration files. Different files, therefore, configure a component depending on the application. To easily switch configuration, YARP components support the parameter *--from* which allows reading configuration files from a well-defined directory (the *context* of execution). For example, the following commands:

```
myModule --from experiment1
myModule --from experiment2
```

execute *myModule* in two different ways, using files in the contexts *experiment1* and *experiment2*, respectively.

The directories *modules, applications, plugins*, and *contexts* are installed in the system directory *<prefix>/share/yarp*.[18] Users have read-only access to system directories, and they need to copy the files they want to edit in private directories in which they have write access.[19] To install and remove resource files, YARP provides the *yarp-config* tool.[20]

The *ResourceFinder* searches for configuration files in the following order of precedence:

- First, it looks in the current working directory;
- Then, it searches within *contexts* in the user private directory;
- Finally, it searches within *contexts* in the shared, installation directory(ies).

When searching for files and directories, the *ResourceFinder* follows the above order, so that files in the working directory or those modified by the user take precedence over installed ones. Search of files proceeds from the user private directories to shared installation directories. The same *context* directory can appear in multiple places and is likely to contain files with the same name. In this case, files that found first take precedence and hide those in other locations. This shadowing or masking mechanism is useful when the user needs to customize only a subset of the files for a specific context.

To allow users to modify where files are stored or to add other *contexts* to the existing ones, the *ResourceFinder* search path can be extended in two ways. Through the *YARP_DATA_DIRS* environment variables, a user can specify a list of locations, each used by the *ResourceFinder* when looking for shared installation directories. Third party package developers can add a text file that contains additional search directories in the directory *<prefix>/yarp/path.d*. This solution allows an installation package to extend YARP search path without requiring changes to the environment (to simplify this task YARP offers a set of CMake functions). For example, a user can install the YARP middleware and the iCub additional software without changing the environment. This is a useful feature for packaging and because we have found that modifying the environment is confusing and error prone for most users.

It is worth mentioning that the *ResourceFinder* follows the XDG Base Directory Specification for Linux systems.[21]

8. APPLICATION MANAGEMENT

To facilitate the application development and execution for the iCub robot, we developed a few graphical tools. The *yarpbuilder* (see **Figure 10**) enables users to easily develop an application by configuring and interconnecting the available modules. It makes use of a YARP module description in XML format and represents them as graphical entities. To build a new application, a developer can drag and drop modules, configure, and interconnect them. This tool also performs some simple model checking to ensure that some of the constraints, such as required input connections or parameters for a module, are satisfied for that specific application.

Using a resource description of the available machines in a cluster, the deployment information can be manually set for the execution of the modules or they can be configured to be deployed using the automated load balancer. Eventually, the application can be created and launched using the *yarpmanager*[22] deployment tool (see **Figure 11**). It has been developed using a multilayered software architecture, which abstracts the representation of modules, resources, and applications from their execution. The latter can employ different deployment methods (e.g., *yarprun* or *SSH*), which potentially allows executing components from different robotic middleware. The *yarpmanager* provides a rich set of functionalities such as module configuration, execution and monitoring, cluster resource discovery, load balancing, as well as establishing and checking connections. We are currently working toward integrating the *yarpbuilder* and *yarpmanager* in a single tool in which applications are developed, executed, and monitored using the same graphical representation.

9. SOFTWARE TESTING ON THE iCUB

Testing is an important topic in software engineering that has, however, received little attention in robotic research. Because robots in the future are expected to work in close interaction with humans, safety of robotic systems is going to become a fundamental issue. In this context, it is likely that software testing will play an important role. To test the iCub software we have adopted strategies for static as well as dynamic testing.

Static testing consists in checking the code without executing it. This approach includes code inspection, peer-to-peer reviews, and code verification using formal techniques. Code reviews allows increasing software quality by removing common problems and enforcing coding styles to improve readability, especially when development happen in a in distributed, open-source

[18] The actual value of *<prefix>* is system dependent, on Linux it is usually equal to */usr/share*, while on Windows it maps to *%PROGRAMFILES%/YARP,* i.e. usually *C:/Program Files/YARP*.

[19] On Linux systems this is *$HOME/.local/share/yarp* while on Windows it is *%APPDATA%/yarp*.

[20] Online documentation is available at: http://www.yarp.it/yarp_yarp-config.html

[21] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

[22] http://www.yarp.it/yarpmanager.html

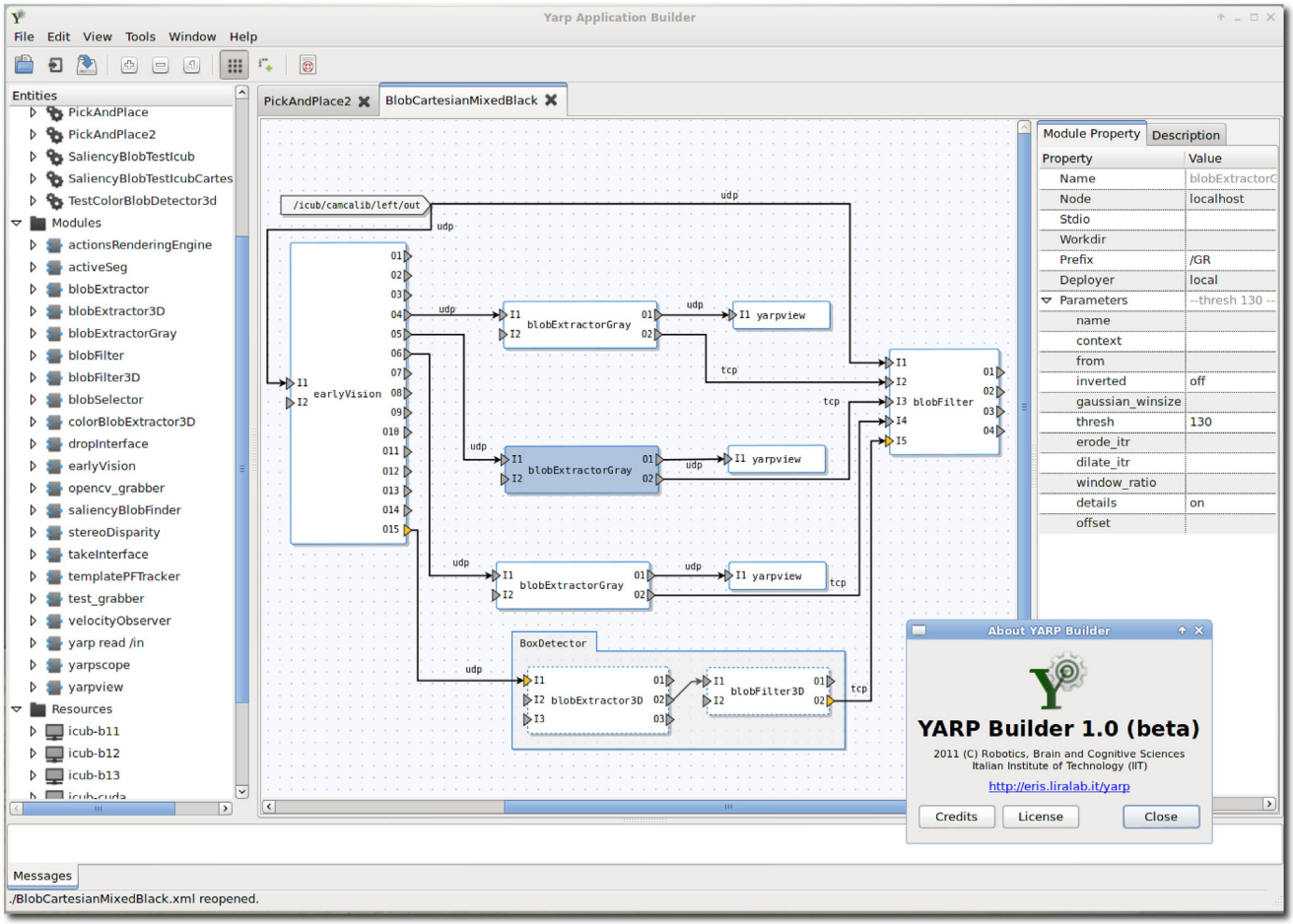


FIGURE 10 | A screenshot of yarpbuilder. The GUI allows to build *applications* by dragging and dropping *components* and by wiring the connections. The *applications* designed in the *yarpbuilder* can be executed by the *yarpmanager*.

community. On the iCub code reviews have been adopted only recently and so far mostly in the context of the development of the YARP middleware. At this aim, we rely on the functionalities offered by github,[23] the web service that hosts the majority of the repositories we manage. Non-trivial modifications to YARP are developed on a separate branch and are integrated in the master (the main development branch) only after other developers have revised and approved it. Common problems that have been identified through code reviews are: race conditions, memory leaks, and buffer overflows to mention a few. An important feature provided by github is the possibility to test software patches *before* they are integrated in the main branch (using Travis[24]). However, Travis supports only a specific distribution of Linux (at the time of writing Linux Ubuntu 12.04 LTS). Therefore, compilation tests are also executed for all the supported platforms on our compile farm. This happens periodically (nightly builds) and upon any commit (continuous builds) to the YARP and iCub main repositories.

Static code analysis can also be performed automatically using model checking techniques [see Baier and Katoen (2008) for a review]. These techniques allow ensuring that a piece of code satisfies given requirements (usually expressed in temporal logic). Model checking is preferable to other techniques because it explores systematically the behavior of a program and is completely automated. However, it requires that a model of the software is available – for example in the form of a finite state automata – and hardly scales to large systems. It has had practical applications in the verification of circuits and protocols (Kaufmann et al., 2000). In recent work, we have investigated the use of automatic techniques to derive a model of some of the components of the YARP middleware (namely *Ports* and *BufferedPorts*) and successfully applied model checking to verify properties of code that uses YARP (Khalili et al., 2014). Although promising, these techniques still require a certain degree of manual intervention from an expert and do not scale well to large programs. Further research is still

[23]https://github.com
[24]https://travis-ci.org/

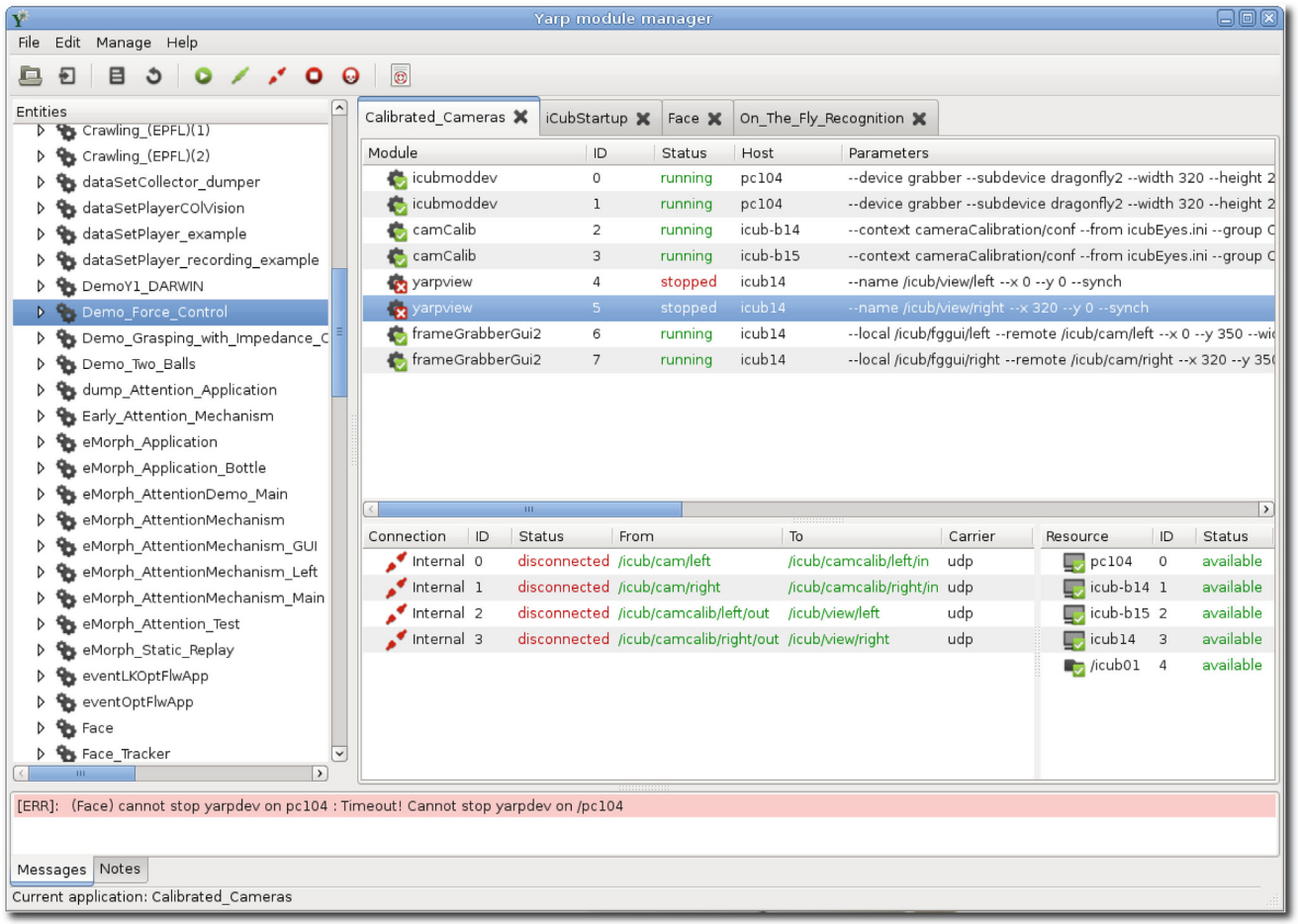


FIGURE 11 | A screenshot of *yarpmanager*. On the left, the GUI shows the list of *applications* that are available and can be loaded. Each tab in the center window contains one of the four *applications* that have been loaded. For each *application*, *the GUI shows the status of the components, the host in which it should be ran and the parameters. The bottom windows show the status of the* Port *and connections that are used by the current* application*. The bottom-right window shows the status of the nodes in the network.*

required before they can be integrated in the software development workflow.

Dynamic testing on the other hand consists of running and testing the dynamic behavior of the code. This involves writing specific pieces of code, specifically devised to exercise the functionalities of the software and verify that it complies with the specifications. Test-driven development has gained increasing attention in software engineering (Beck, 2003). Proper application of this technique requires (i) alternating writing tests and developing functional code in small and rapid iterations and (ii) executing tests automatically to ensure that modifications to existing code (new components, bug fixes, or new features) do not break existing functionalities. In the remainder of this section, we describe how unit-testing has been adopted for testing the software on the iCub robot.

9.1. The Robot Testing Framework

Unit-testing was adopted for the development of YARP since the beginning, whereas systematic testing of the software that controls the iCub was started only recently. This is because robot software cannot be tested in isolation, and it requires running other components, device drivers, or just the robot simulator (these resources are called *fixture*). Automating testing requires therefore that the testing framework is able to setup the required resources and monitor them to ensure that they remain functional during the execution of the tests, offering hooks to handle failure appropriately (i.e., restarting the robot, performing parking routines, etc.). To comply with these requirements, we developed the Robot Testing Framework (RTF) (Paikan et al., 2015d).

The RTF[25] is a generic testing framework for test-driven development of robotic systems. Its architecture is detailed in **Figure 12**. It is based on the well-known xUnit test patterns, which includes a test runner, test result formatters, and a test fixtures manager. In addition, it provides functionalities for

[25]The source code and documentation of the RTF can be accessed on-line at http://robotology.github.io/robot-testing/index.html.

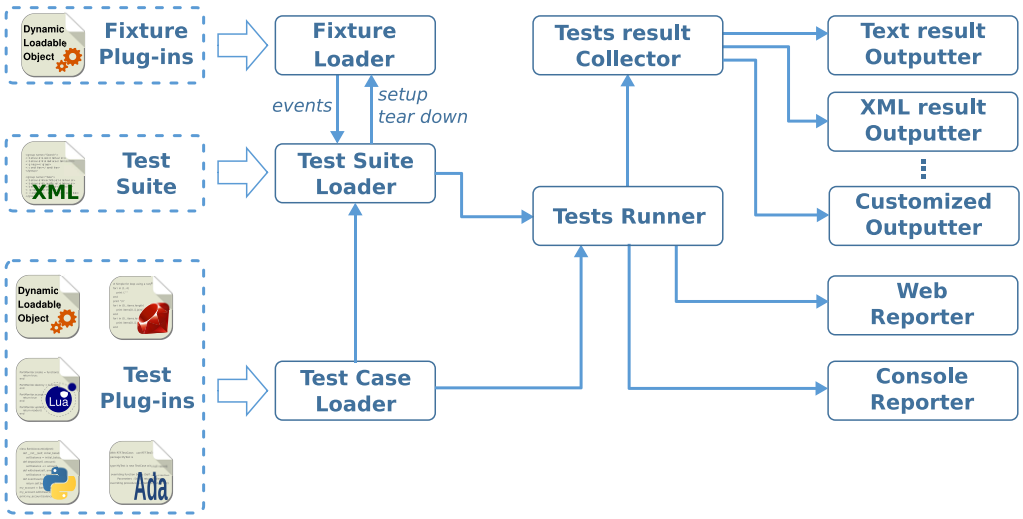


FIGURE 12 | The architecture of the *Robot Testing Framework*. Test cases can be developed as independent plug-ins using scripting languages or can be built as dynamically loadable libraries. The plug-ins are loaded by the *Test Case Loader* and are executed by the *Test Runner*. Test cases can also be grouped in different test suites, which are represented using XML. In the latter case, the *Test Suite Loader* parses the XML file and, using the *Test Case Loaders*, it loads the corresponding test plug-ins. Each test suit can optionally have a fixture manager, which is implemented as a separate plug-in (which is loaded by the *Fixture Loader*). This fixture plug-in is responsible for setting up the fixture and informing the *Test Suite Loader* when the fixture fails (e.g., crashes). In this case, the *Test Suite Loader* restarts the fixture and resumes execution of the remaining test cases. The result of the tests can be monitored from the console (through the *Console Reporter*) or remotely from a Web browser (through the *Web Reporter*). The *Test Result Collector* allows storing data in different formats. For example, this figure shows two components for storing output in text format or XML, (*Text result Outputter* and *XML Result Outputter*, respectively).

defining test cases (i.e., unit tests), suits, and assertions. RTF supports multiple middleware, languages, and operating systems. This is achieved by providing abstraction layers for the platform (i.e., operating system), the middleware, and the programing language. Moreover, RTF provides functionalities for managing complex fixtures, which support stress testing at the level of individual components (robot hardware like sensors or actuators) as well as integrated (sub) systems. It is worth pointing out that the RTF is not a tool for static analysis of code. As such it does not perform an exhaustive analysis of the code to ensure given properties of absence of deadlocks. The code is evaluated by running tests that call individual functions and verify the value of the return parameters or observe the internal status of the system under test by calling specific functions. Deadlocks can be detected indirectly by setting a timeout on the execution of individual tests.

A test suite is a set of test cases, which share the same test fixture (Meszaros, 2007). In RTF, a set of test cases (plug-ins) can be grouped as a test suite using an XML file and executed using the test runner. This allows the unit tests to be easily organized in different test suites, which are easy to maintain and extend.

The fixture manager is implemented as a separate plug-in so that it can be implemented using the deployment tools and policy of the middleware of choice. (for example OROCOS uses the component deployer, *deployer-corba*, ROS components use *roslaunch* toolset). For YARP, it is implemented using the *yarpmanager*.

Notably, RTF provides test results in different formats including Junit XML file. This allows the test results to be published and monitored using standard integration tools such as Jenkins. The next section describes the tests that are currently implemented for the iCub. The test cases can be run directly on the iCub robot or using a simulator. Some of the tests are executed automatically on the simulator using Jenkins upon any change in YARP middleware or the iCub software. These tests check that any new update in the software is compatible with the robot interfaces. There are also some tests that check the robot hardware; these tests are executed directly on the robot and under human supervision.

9.2. Testing on the iCub

We conclude this section with a description of the tests that have been implemented so far on the iCub robot. We can distinguish four categories of tests: (i) tests on the correctness of the robot configuration files, (ii) tests for specific hardware devices, (iii) tests for the compliance of low-level software and firmware with system specifications, and finally (iv), tests that are specifically written after a specific bug is identified and fixed.

The first category of tests may seem unusual. There exists about 30 iCub units, and, over the years, many of them underwent hardware customizations, revisions, or upgrades. Therefore, each unit has a specific set of configuration XML files, typically manually written and therefore subject to errors. Automatic tests attempt to minimize such errors by verifying the correct behavior of the robot.

JointLimits, for example, is a test which belongs to the first category. This test checks that range of motion written in the configuration files is achievable by the system by moving each joint to the maximum and minimum position. If a hardware

limit is unexpectedly encountered, the test fails, informing of a possible mistake in the robot configuration files. This kind of tests can be considered rather static over time: additional tests may be added if new configuration parameters are introduced, however, individual tests do not typically require maintenance.

The second category includes all the tests which validate the correct operation of a hardware device. For example, the *OpticalEncodersDrift* test moves a joint generating a sinusoidal trajectory. The test checks that, after a repeated number of cycles, the measurements of the optical encoder do not drift. A drift suggests a hardware defect such as the presence of dust on the encoder disk (for optical encoders) or electrical interference. This category of tests is dynamic: new devices are continuously introduced and new tests have to be designed for the new hardware, while existing tests have to be periodically reviewed as a result of a hardware revision of an existing device.

The third category includes all the tests which verify that the robot behavior complies with the specifications and requirements. The simplest test of this category is *MotorEncodersSignChecks*, which tests that individual encoders increase their value when the corresponding motor rotates with positive input. This is an important convention that determines the sign of the PID controllers but may be affected by incorrect mounting of the encoders, motor wiring, or firmware configuration. As previously mentioned, iCub control boards implements the concept of control mode. If a joint is controlled in position mode, for example, only position commands sent through the *IPosition* interface are accepted, while other commands (e.g., velocity, torque) must be rejected. *ControlModes* extensively verifies that all the possible states described in the specifications of control mode state machine are reachable and that the corresponding transitions are correct. Also prohibited transitions are tested. For example, if a hardware fault occurs (the test intentionally causes a hardware fault by sending an invalid command) the user has to set the joint in idle mode before switching to any other control mode. *MotorTest* exercises the interfaces for reading encoders and moving the joints in position mode. This category of tests typically evolves over time: for example, when a new control mode or interface is implemented, these tests may be extended in order to check the compatibility of the new features with existing ones.

The fourth category of tests is written when individual software defects are discovered. In this case, it is good practice to first write a test that triggers the specific defects and then ensure that a candidate software patch effectively passes the test. These tests remain in the system to ensure that future changes will not cause the same defect to reappear. An example of this test is *SensorsDuplicateRd*, which verifies that sensors values are not broadcast multiple times on a *Port*.

Table 2 lists all the tests that are currently implemented on the robot. These tests are executed periodically on the simulator and manually on the real robot. As the development is currently in progress, the list covers only partially the functionalities that could and should be tested.[26]

[26]The source code and description of the available tests for iCub can be accessed on-line at https://github.com/robotology/icub-tests.

TABLE 2 | Some of the test cases which has been developed for the iCub robot.

Test	Description
`CameraTest`	Checks the robot camera's frame rate and size
`ControlModes`	Checks control mode specifications, validates allowed, and forbidden transitions
`FtSensorTest`	Checks the robot force sensors against a predefined, known value
`JointLimits`	Checks the software joint limits configuration
`MotorTest`	Checks the IPositionControl and IEncoders interfaces, moves the motors individually or in groups, and verifies that the required position is actually obtained within a predefined amount of time
`MotorEncodersSignChecks`	Check that motor encoder readings increase when positive PWM is applied to a motor
`OpticalEncodersConst`	Checks the consistency between encoders at the motors and at the joints
`OpticalEncodersDrift`	Performs repetitive movements to verify that encoders do not drift
`PortFrequencyTest`	Checks the rate at which state information is published by the robot interfaces
`PositionDirect`	Checks the *IPositionDirect* control mode
`SensorsDuplicateRd`	Checks that a YARP *Port* publishes unique values at each update

10. LESSONS LEARNED

We report here a list of lessons we learned during the development of the YARP middleware and the iCub software architecture.

10.1. Freedom of Choice versus Freedom from Choice

The iCub is a research platform developed by and for researchers. We tried to accommodate as much as possible the need for the users of the robot giving maximum freedom in terms of development environment and tools. We support MacOS X, different flavors of Linux, and Windows. In several occasions, this turned out to be a good choice that allowed us to run the code on legacy hardware and to reach a wider group of users. It, however, required to keep support for various versions of compilers and libraries and resulted in considerable maintenance cost, sometimes slowing down the introduction of new features made available in new compilers or libraries. On the software development side, we also gave users freedom to code their components in the way the liked; for this reason, we did not introduce a rigid component model but only provided helper classes and best practices through documentation and on-line tutorials. This choice in the short term was beneficial because it reduced the learning curve, but in the long term may hinder standardization and actually slow down development. A somewhat opposite approach would be to rely on design tools and code generation to leverage the user from the task of implementing infrastructure code [this approach is followed for example by Smartsoft (Schlegel and Worz, 1999)]. Striking a good balance between the two approaches is a difficult design decision, which depends on background and expectations of the target users.

10.2. In-House Middleware

When we started the development of iCub back in 2004, only a few middleware existed in the robotics community and their use was quite fragmented. The first version of YARP was already mature and the consortium that designed the iCub included YARP's core developers in the team. This was one of the main reasons that has motivated the adoption and consequent development of YARP. Having in-house control of the middleware code-base has given us great flexibility to address the needs of the iCub community. Such examples have been described in this paper and include: the definition of the motor control interfaces and corresponding communication paradigm for remotization, the functionalities for remote execution and logging, and organization of parameters. Not less importantly it allowed us to come up with novel extensions like the *Port Monitor* and channel prioritization. YARP can be easily compiled on many Linux distributions, MacOS, and Windows. All these reasons still prevent us today from adopting ROS altogether and motivated us to add support for ROS interoperability, instead. Developing, debugging and maintaining the communication back-bone of the robot, however, has quite a high cost and should not be underestimated when developing a new robot. Our experience has shown that user code become quite entangled with the middleware data structures and build system (*middleware lock-in*). Considered that middleware technology is in constant evolution it may be a good idea to design the software architecture of the robot to reduce such dependency, so that changing middleware is possible and inexpensive.

10.3. Packet Management System

The build system of the iCub software is based on CMake. CMake offered a great degree of flexibility, allowing to customize and to automate the compilation process, including finding and configuring dependencies. However, the software ecosystem suffered from the lack of a sophisticated packet management system. The main stumbling block in this respect was the decision to support Windows, for which there are no mature systems for packet management and distribution. To partly cope with this problem, we developed custom scripts for packaging dependencies in binary distributions for Windows, Linux, and, hombrew recipes for MacOS X. Yet, the development of an iCub software ecosystem was slowed down from the lack of a powerful packet management system like the one that is available in many Linux distributions.

10.4. Lack of IDL

YARP was not born with an IDL language supporting the definition of data type and services. The reason for this choice was to facilitate adoption by reducing complexity and learning curve. Interfaces were developed using self-describing data types (i.e., the YARP *Bottle*), which allowed code to parse messages dynamically by inspecting their content. With the growth of the community, this became a limitation because it made it difficult to document modules interfaces, perform versioning, and verify compatibility between modules. In retrospect it would have been preferable to introduce the IDL much earlier in the development, enforcing its use like, for example, ROS. Finally, another advantage of the IDL is that it allows documenting service interfaces using Doxygen keeping services documentation and their code in the same files.

10.5. External Configuration of Ports

YARP provides different communication policies through the *Port* API (like buffering, streaming versus RPC). However, understanding how components communicate requires looking at the code; this increases the probability of introducing subtle bugs. It would be beneficial if the available polices were visible and configurable at run time. In Section 3, we showed initial steps in the direction for providing policies for channel prioritization; however, further development is required to give users access to the other configuration parameters.

10.6. Robot Interface Abstraction Layer

The robot interface abstraction layer had a positive impact during the development. It allowed to introduce new functionalities *via* new interfaces without affecting existing code and to execute code on-board, remotely on the real robot or simulators. These features gave a useful level of flexibility that facilitated debugging, code re-use, and fast-prototyping. Robot interfaces have evolved with time to accommodate the research requirements and to support new hardware as it became available; we tried to limit as much as possible the impact of these changes on the user code and introducing new functionalities in optional interfaces. The robot interface abstraction layer hides the user code from the details of the communication and the specific communication middleware. This allowed to extend the representation of the data structure that broadcasts the robot state without changes outside library code. In the future, it may even allow us to change the transport layer (i.e., the communication middleware) altogether with minimal impacts. Another improvement in the robot interface followed the refactoring of the plugin system, which was modified to load plug-ins at runtime using *dynamic libraries* (it was initially based on static linking). This decoupled all dependencies and allowed us to separate device drivers contributed by users in separate repositories, resulting in simpler packaging and maintenance.

10.7. Test-Driven Development

The development cycle of the YARP middleware adopted unit testing since the beginning. For practical reasons, this approach was extended only recently to the other layers of the iCub software architecture. The *Robot Testing Framework* is being used to test the low-level code that is developed for the new version of the iCub and although tests still cover a relatively small portion of the code, we already see its benefits: besides detection of bugs due to programmers errors, it allowed us to detect problems that were due to lacking or imprecise specifications. The latter is a problem that is particularly frequent in research environments.

11. CONCLUSION

In this paper, we described the software architecture of the iCub humanoid robot, including some of its recent development. We illustrated the design choices that have guided and constrained its development, including the lessons that we learned during

this endeavor. We do not claim these choices to be optimal and equally good in all cases. However, we hope that in the future, this paper may provide a useful guide for the design of other humanoid robots.

AUTHOR CONTRIBUTIONS

All authors have contributed to the conceptual design or development of the work described in this paper and have participated to drafting and revising its content. They also approve publication of the paper and agree to be accountable for all aspects of the work described therein.

ACKNOWLEDGMENTS

We acknowledge the contribution of the software development team and researchers in the iCub Facilty and Robotics Brain and Cognitive Sciences departments, as well as the contributions of the whole iCub community.

FUNDING

This project has received funding from the European Union's Seventh Framework Programme for research technological development and demonstration under grant agreement No. 270273 (Xperience), project No. 611832 (WALK-MAN).

REFERENCES

Almesberger, W., Salim, J. H., and Kuznetsov, A. (1999). "Differentiated services on linux," in *Proceedings of Globecom '99*, Vol. 1, (Rio de Janeiro: IEEE), 831–836.

Baier, C., and Katoen, J.-P. (2008). *Principles of Model Checking*, Vol. 26202649. Cambridge: MIT Press.

Beck, K. (2003). *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley Professional.

Brooks, A., and Kaupp, T. Makarenko, A., Williams, S., and Oreback, A. (2005). "Towards component-based robotics," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Alberta: IEEE), 163–168. doi:10.1109/IROS.2005.1545523

Brugali, D., and Scandurra, P. (2009). Component-based robotic engineering. Part I: reusable building blocks. *IEEE Robot. Autom. Mag.* 16, 84–96. doi:10.1109/MRA.2009.934837

Bruyninckx, H. (2001). "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, Vol. 3, (Seoul: IEEE), 2523–2528.

Bruyninckx, H., Klotzbücher, M., Hochgeschwender, N., Kraetzschmar, G., Gherardi, L., and Brugali, D. (2013). "The BRICS Component Model: A Model-Based Development Paradigm for Complex Robotics Software Systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, (Coimbra: ACM), 1758–1764.

Calisi, D., Censi, A., Iocchi, L., and Nardi, D. (2012). Design choices for modular and flexible robotic sofware development: the openrdk viewpoint. *J. Softw. Eng. Robot.* 3, 13–27.

Collett, T. H., MacDonald, B. A., and Gerkey, B. (2005). "Player 2.0: toward a practical robot programming framework," in *Proceedings of the Australian Conference on Robotics and Automation*, Sydney.

Dantam, N., Lofaro, D., Hereid, A., Oh, P., Ames, A., and Stilman, M. (2015). The ach library: a new framework for real-time communication. *IEEE Robot. Autom. Mag.* 22, 76–85. doi:10.1109/MRA.2014.2356937

Einhorn, E., Stricker, R., Gross, H., Langner, T., and Martin, C. (2012). "MIRA – Middleware for Robotic Applications," in *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Vilamoura: IEEE), 2591–2598.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.* 35, 114–131. doi:10.1145/857076.857078

Fitzpatrick, P., Ceseracciu, E., Domenichelli, D., Paikan, A., Metta, G., and Natale, L. (2014). A middle way for robotics middleware. *J. Softw. Eng. Robot.* 5, 42–49.

Fitzpatrick, P., Metta, G., and Natale, L. (2008). Towards long-lived robot genes. *Rob. Auton. Syst.* 56, 29–45. doi:10.1016/j.robot.2007.09.014

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc.

Habra, T., Dallali, H., Cardellino, A., Natale, L., Tsagarakis, N., Fisette, P., et al. (2015). "Robotran-YARP interface: a framework for real-time controller development based on multibody dynamics simulation," in *ECCOMAS Thematic Conference on Multibody Dynamics* (Barcelona).

Hammer, T., and Bäuml, B. (2014). The communication layer of the ardx software framework: highly performant and realtime deterministic. *J. Intell. Robot. Syst.* 77, 171–185. doi:10.1007/s10846-014-0095-9

Hoffman, E. M., Traversaro, S., Rocchi, A., Ferrati, M., Settimi, A., Romano, F., et al. (2014). "YARP based plugins for gazebo simulator," in *Modelling and Simulation for Autonomous Systems Workshop (MESAS)*, ed. J. Hodicky (Roma: Springer), 333–346.

Huang, A. S., Olson, E., and David, M. (2010). "LCM: Lightweight Communications and Marshalling," in *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Taipei: IEEE), 4057–4062.

Ierusalimschy, R., De Figueiredo, L. H., and Celes Filho, W. (1996). Lua an extensible extension language. *Softw. Pract. Exp.* 26, 635–652.

Kaufmann, M., Moore, J. S., and Manolios, P. (2000). *Computer-Aided Reasoning: An Approach*. Norwell, MA: Kluwer Academic Publishers.

Khalili, A., Natale, L., and Tacchella, A. (2014). *Reverse Engineering of Middleware for Verification of Robot Control Architectures, Volume 8810 of Lecture Notes in Computer Science, Book Section 27* (Cham: Springer International Publishing), 315–326.

Klotzbücher, M., and Bruyninckx, H. (2012). Coordinating robotic tasks and systems with rFSM statecharts. *Int. J. Softw. Eng.* 1, 28–56.

Lütkebohle, I., Philippsen, R., Pradeep, V., Marder-Eppstein, E., and Wachsmuth, S. (2011). Generic middleware support for coordinating robot software components: the task-state-pattern. *J. Softw. Eng. Robot.* 2, 20–39.

Maiolino, P., Maggiali, M., Cannata, G., Metta, G., and Natale, L. (2013). A flexible and robust large scale capacitive tactile system for robots. *IEEE Sens. J.* 13, 3910–3917. doi:10.1109/JSEN.2013.2258149

Mastrogiovanni, F., Paikan, A., and Sgorbissa, A. (2013). Semantic-aware real-time scheduling in robotics. *IEEE Trans. Robot.* 29, 118–135. doi:10.1109/TRO.2012.2222273

Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Boston, MA: Addison-Wesley.

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., et al. (2010). The iCub humanoid robot: an open-systems platform for research in cognitive development. *Neural Netw.* 23, 1125–1134. doi:10.1016/j.neunet.2010.08.010

Nori, F., Traversaro, S., Eljaik, J., Romano, F., Del Prete, A., and Pucci, D. (2015). iCub whole-body control through force regulation on rigid noncoplanar contacts. *Front. Robot. AI* 2:6. doi:10.3389/frobt.2015.00006

Paikan, A., Domenichelli, D., and Natale, L. (2015a). "Communication channel prioritization in a publish-subscribe architecture," in *Software Engineering and Architectures for Realtime Interactive Systems*, Arles.

Paikan, A., Pattacini, U., Domenichelli, D., Randazzo, M., Metta, G., and Natale, L. (2015b). "A best-effort approach for run-time channel prioritization in real-time robotic application," in *International Conference on Intelligent Robots and Systems (IROS)* (Hamburg: IEEE), 1799–1805.

Paikan, A., Schiebener, D., Wacther, M., Asfour, T., Metta, G., and Natale, L. (2015c). "Transferring object grasping knowledge and skill across different robotic platforms," in *IEEE International Conference on Advanced Robotics* (Instanbul: IEEE), 498–503.

Paikan, A., Traversaro, S., Nori, F., and Natale, L. (2015d). "A generic testing framework for test driven development of robotic systems," in *Modelling and*

Simulation for Autonomous Systems: Second International Workshop, MESAS 2015, Prague, Czech Republic, April 29-30, 2015, Revised Selected Papers (Cham: Springer International Publishing), 216–225.

Paikan, A., Fitzpatrick, P., Metta, G., and Natale, L. (2014a). Data flow port's monitoring and arbitration. *J. Softw. Eng. Robot* (Chicago: IEEE), 5, 80–88.

Paikan, A., Tikhanoff, V., Metta, G., and Natale, L. (2014b). "Enhancing software module reusability using port plug-ins: an experiment with the iCub robot," in *International Conference on Intelligent Robots and Systems (IROS)* (Montevideo: IEEE), 1555–1562.

Paikan, A., Metta, G., and Natale, L. (2013). "A port-arbitrated mechanism for behavior selection in humanoid robotics," in *16th International Conference on Advanced Robotics (ICAR), 2013*, Montevideo, 1–7.

Parmiggiani, A., Maggiali, M., Natale, L., Nori, F., Schmitz, A., Tsagarakis, N., et al. (2012). The design of the iCub humanoid robot. *Int. J. HR* 9, 1250027. doi:10.1142/S0219843612500272

Pattacini, U., Nori, F., Natale, L., Metta, G., and Sandini, G. (2010). "An experimental evaluation of a novel minimum-jerk cartesian controller for humanoid robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Taipei: IEEE) 1668–1674.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). *ROS: An Open-Source Robot Operating System*.

Randazzo, M. (2004). *iCub Control Modes Specifications*. Technical Report. Genova: iCub Facility, Istituto Italiano di Tecnologia.

Sametinger, J. (1997). *Software Engineering with Reusable Components*. Berlin: Springer.

Schlegel, C., and Worz, R. (1999). "The software framework smartsoft for implementing sensorimotor systems," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vol. 3 (Kyongju: IEEE), 1610–1616.

Stewart, D., Volpe, R., and Khosla, P. (1997). Design of dynamically reconfigurable real-time software using port-based objects. *IEEE Trans. Softw. Eng.* 23, 759–776. doi:10.1109/32.637390

Tikhanoff, V., Cangelosi, A., Fitzpatrick, P., Metta, G., Natale, L., and Nori, F. (2008). "An open-source simulator for cognitive robotics research: the prototype of the iCub humanoid robot simulator," in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems* (New York, NY: ACM), 57–61.

Tsagarakis, N. G., Morfey, S., Cerda, G. M., Zhibin, L., and Caldwell, D. G. (2013). "Compliant humanoid coman: optimal joint stiffness tuning for modal frequency control," in *IEEE International Conference on Robotics and Automation* (Karlsruhe: IEEE), 673–678.

Vernon, D., Billing, E., Hemeren, P., Thill, S., and Ziemke, T. (2015). An architecture-oriented approach to system integration in collaborative robotics research projects – an experience report. *J. Softw. Eng. Robot.* 6, 15–32.

Zenger, M. (2004). *Programming Language Abstractions for Extensible Software Components*. Ph.D. thesis, École Polytechnique Fédérale de Lausanne, Lausanne.

Conflict of Interest Statement: The work described in this paper was conducted in the absence of any commercial or financial relationship that can be construed as a potential conflict of interest.