ROSlink: Interfacing legacy systems with ROS

Fabio Dalla Libera
JSPS Research Fellow at Osaka University
Osaka, Japan
Email: fabio.dl@is.sys.es.osaka-u.ac.jp

Hiroshi Ishiguro
Osaka University
Osaka, Japan
Email: ishiguro@sys.es.osaka-u.ac.jp

***Abstract*—This paper presents *ROSlink*, an open source project that aims at easing the integration of legacy systems with ROS (Robot Operating System). Its design principles provide a set of unique features that make it appealing for the interconnection of ROS with systems where ROS itself cannot be installed. First, ROSlink requires very limited changes to the legacy system. The project is self contained, bringing in no dependencies, which may be difficult to satisfy in a legacy system. Furthermore, with ROSlink any data type already in use in the legacy system can be employed for the communication of topics and service requests and responses. ROSlink allows run-time rerouting of the communication between the legacy system and ROS. Moreover, it empowers the legacy code with the ROS name remapping system, without enforcing any constraint on the command line parameters of legacy programs. Finally, by simply using a set of API that closely follow the ROS programming interface, ROSlink simplifies any successive porting of the code to a real ROS system. In this paper, the main design choices of ROSlink are discussed. A list of practical applications and tests where ROSlink was employed, as well as a short discussion on the project's future directions are then given.**

I. Introduction

Software running on robots is often very complex. A high variety of algorithms, from joint level control to AI reasoning methods must cooperate to achieve the desired goals. The complexity of this kind of control can be managed only by organizing the code as a set of subsystems, possibly operating at different layers, that interact with each other.

Over the years various middleware for the communication of these subsystems were proposed. Among the most well-known, we can certainly cite Player [1], URBI [2], YARP [3], OpenRTM [4], Miro [5], OpenRdk [6], Orca [7], MARIE [8], CARMEN [9], OPRoS [10], LCM [11] and the JAUS based SDKs such as OpenJAUS [12].

Recently, a new project, ROS [13], is gaining stronger and stronger attention in the robotics field. The simplicity of its API, good documentation and the support by an active community are just some of the features that makes it the middleware adopted for the integration of many new robotic projects. Launched by Willow Garage in 2007, ROS is currently officially supported by 28 robots, features over 2000 packages and has a wiki of over 14000 pages with over 2000 users. The scope of the software packages available is very wide, and ranges from hardware peripheral drivers to point cloud processing [14], from inverse kinematic libraries to Human Robot Interaction [15].

Reuse of this huge amount of publicly available software through ROS is very appealing for speeding up the development of robotic systems. It allows researcher to focus only on the parts of the systems directly correlated with their research. Problems arise, though, when it becomes necessary to deal with legacy systems, for which ROS is not available. Currently ROS supports only Ubuntu, and in an experimental way, other five Linux distributions (Debian, Fedora, Gentoo, OpenSuse and Arch Linux), OS X 10.5 or higher and Windows XP SP3 or higher. However, roboticists may be tied to use other, unsupported platforms for many reason. Examples are the availability of the driver of a peripheral only for an old OS, or the necessity of using proprietary programs compiled for a particular system and available only in binary form. Hard realtime requirements may impose the use of an unsupported, real-time operating system like QNX. Hardware constraints, such as very limited on-board flash memory mounted in an embedded device, may prevent the installation of a supported operating systems as well. Finally, there may be simply the desire of modifying working and debugged systems as little as possible.

The work presented in this paper aims at coping with all these cases in which constraints imposed by a legacy system clash with the need of interfacing legacy code with ROS, for making a legacy system available to new ROS packages, or for equipping a legacy system with functionalities provided by ROS packages. Related works and the design principles adopted will be described in the next section. Details of the actual solutions employed for achieving these goals will then be given. Successively, use cases will be briefly introduced. Finally, future work will be discussed.

II. Design Principles

ROSlink, the project presented in this paper, aims at allowing legacy software (or software running or legacy systems) to interact with ROS, without actually having to install ROS on the machine running the legacy code. During the software development, the following two fundamental goals were set:

1) Minimization of the API to be learned by the users.
2) Minimization of the changes required to the legacy system.

In order to achieve these objectives, it was decided to provide a C++ implementation that exposes to the legacy system a set of API essentially identical to *roscpp*, the C++ implementation of ROS. This allows users familiar with *roscpp* to start using ROSlink with minimal effort. In order to require as few software installations and as little code modification as possible, the code was split into two components. The first, named *helium*, is a lightweight (hence the name), self contained C++ library intended to be used by the legacy code. The second component, named *roslink*, is a library, distributed as a ROS package and depending just on ROS, that is used to perform marshalling between ROS and the legacy code. In the current implementation the two components communicate through TCP/IP, however switching to other types of communication, like I^2C, could be achieved by changing the implementation of few classes.

This subdivision into two components is similar to the one taken by rosbridge [16]. Indeed, rosbridge opens a TCP/IP server, which accepts commands, formatted using the JSON syntax, to subscribe/publish to ROS topics and to invoke ROS services. Clients can therefore interact with ROS through a TCP/IP (or WebSocket) connection by sending JSON objects having the fields specified by the rosbridge protocol. The original motivations of the two projects, however, reflect in different (and complementary) features. More in detail, rosbridge is intended for allowing non-ROS clients, like javascript in web-pages, to access existing ROS code. ROSlink, instead, aims at allowing the reuse of legacy systems in new projects. As a result, for instance, rosbridge does not allow clients to provide ROS services (but only to invoke them). Conversely, allowing ROS nodes to call services provided by the legacy code was set as one of the main goals since the start of ROSlink development. Similarly, while rosbridge delegates to its users the conversion of their data types into JSON messages, rosbridge focuses in providing its users with a set of simple API for speeding up the development process.

Along these lines, for achieving a fast and easy integration of legacy systems into ROS, the following distinctive design principles were set for ROSlink:

(A) minimal set of dependencies for the *helium* component
(B) compatibility between *helium* API and *roscpp* API
(C) ability of dynamically creating ROS publishers, subscribers, service servers and clients
(D) robustness to run-time changes of the network topology
(E) complete support of the ROS name remapping system
(F) marshalling between ROS and legacy code completely isolated from the legacy code

In the following, details for each of these design choices will be briefly provided.

A. Dependencies

The library to be linked with legacy code, *helium*, is self contained, apart from networking and multithreading. Networking is implemented by using Berkeley sockets when *helium* is compiled on POSIX systems, and by using the Winsock API when it is compiled on Windows. Similarly, multithreading is achieved using Pthreads on POSIX systems and native Windows threads when compiled on Windows. The OS-dependent code is isolated in few classes, and can be reimplemented in case neither Pthreads nor Windows API can be used. The choice of not using other libraries that increase code portability, and in particular *Boost*, comes from the desire of completely avoiding possible incompatibilities between the current *Boost* API and older versions that may be used by the legacy code. Similarly, the library comes with a cmake file, but manually written Visual Studio Projects, and Makefiles for Linux and QNX are provided for the systems in which cmake is not available. Users can also choose not to compile the code as a library, but simply to compile the *helium* code together with theirs.

B. API

The API exposed to the legacy system mirror as closely as possible the *roscpp* API. Each of the ROS objects, like *ros::NodeHandle* or *ros::Publisher*, find their counterpart in *helium*, in this case as *legacy::NodeHandle* and *legacy::Publisher*, with member functions that mirror their ROS equivalent. Peculiarities of ROS, like the possibility of publishing to a single subscriber in a *SubscriberStatusCallback*, are provided. When ROS throws an exception (for instance, because an invalid pathname is given in the construction of a *NodeHandle*) an exception is thrown in the legacy code as well. Setting, reading and searching parameters can be done in *helium* in exactly the same way it can be done in ROS. This brings a great advantage: in case an unsupported OS becomes supported by ROS at some point in time, it is sufficient to replace the *legacy* namespace with the *ros* namespace to use the native support.

To allow the communication with *roslink*, the API introduces a new object, called *legacy::Link*. Each *legacy::Link* object is identified by a name specified through its constructor. When a ROS node instantiates a *ros::Link* object (provided by the *roslink* package) with the same name, the communication is established. Conceptually, operating on a *legacy::NodeHandle* constructed in the legacy code is the same as operating on a *ros::NodeHandle* constructed in the node where the *ros::Link* object is declared. Services exported by the legacy code using the *advertiseService* member of a *legacy::NodeHandle* are seen by ROS as services of the ROS node where the *ros::Link* object is created. As will become clear in the next section, a single legacy program can declare multiple *legacy::Link* objects. For this reason, the link to be used by each *legacy::NodeHandle* (and thus, conceptually, the ROS node it belongs to), can be specified by passing a reference to the *legacy::Link* object to the *legacy::NodeHandle*'s constructor.

C. Dynamic creation and destruction

In ROS, when a *ros::Service* object is initialized, the service is advertised in the system, and when the last copy of that *ros::Service* is destructed, the service is automatically unadvertised. The same philosophy, valid for publishers, subscribers and clients, is maintained in *helium*. In particular, even if a ROS node containing a *ros::Link* aimed at exporting a legacy service is up and running, the corresponding ROS service becomes available in ROS only when the corresponding *legacy::Service* is advertised. In the same way, the service is unadvertised as soon as the last copy of the *legacy::Service* is destroyed, or when the connection between the *legacy::Link* and the *ros::Link* goes down.

D. Link network topology

Data exchange between legacy systems and ROS takes place through the communication between the *helium* library and ROS nodes that, using the *roslink* library, expose the legacy system functionalities to ROS and vice versa. More precisely, the communication takes place between a *legacy::Link* object declared in the legacy code and a *ros::Link* object with the same name created in a ROS node. ROSlink imposes no instantiation order, i.e. it is possible to launch either the ROS node or the legacy code first. The two components do not need to know the location (hostname and TCP port) of the paired entity either. Indeed, a program provided with *helium*, called *lmaster*, acts as a DNS server, in the same way *roscore* allows nodes to communicate without knowing each other's hostname and port beforehand. Specifically, when a *legacy::Link* or *ros::Link* objest is created, it automatically registers itself to the *lmaster* server, and when the paired entity becomes available, the object is notified the hostname and TCP port to connect to, so that a direct connection between the two objects can be established. This kind of approach allows a very flexible dynamic reconfiguration of the network topology. Fig. 1 provides example of four possible scenarios:

(a) Each legacy program uses an independent *legacy::Link* to connect to a corresponding ROS node that declares a single *ros::Link*.
(b) All legacy programs use the same *roslink*, provided by a ROS node responsible for all the legacy nodes.
(c) A legacy program uses multiple links, to provide conceptually different services in different ROS nodes.
(d) A mixed approach, where many simple legacy services are mapped to the same ROS node but a single legacy program, source of a high bandwidth data stream, is connected to a ROS node running on a different machine.

Thanks to the dynamic binding approach provided by *lmaster*, the configuration can be switched at run-time, for instance for balancing the load between multiple machines or for compensating a temporary failure of a machine.

E. Namespace remapping

ROS offers a very powerful system for remapping the names of topics and services. This allows "pushing down" a complete namespace, and thus easily integrate multiple systems from heterogeneous sources without name conflicts. Additionally, names of nodes, topic and parameters can be remapped from the command line or launch files to execute the same code under multiple configurations. Additional remappings can be specified in the creation of a *NodeHandle* object, allowing the remapping of the names declared in the subtree rooted at that particular *NodeHandle*. Finally, a particular namespace, called private namespace, is created for each node. The parameters in this namespace can be assigned very easily from the command line. All these features are maintained in ROSlink. In particular, remappings can be specified in the creation of *legacy::NodeHandle* objects. Furthermore, for the model introduced by ROSlink, nodes created in legacy code conceptually lie in the ROS node containing the corresponding *ros::Link*. For this reason, when a ROS node containing a *ros::Link* is launched with remapping arguments, the same remappings are applied to the legacy code linked through the corresponding *legacy::Link*. This allows ROSlink users to easily remap how the legacy system is seen from ROS (and the other way around) without restarting the legacy code. Furthermore, it enables remapping operations without passing command line arguments to the legacy code, which may parse the command line arguments in a way that is incompatible with the ROS remappings and parameter assignments.

F. Marshalling

The data types of topics and service requests and responses are usually defined in ROS using a very intuitive definition language. Scripts are then used to generate files for each particular programming language, header files in the C++ case. The generated code provides serialization and deserialization methods, which enable the actual transfer of the data over the network. When interfacing legacy code with ROS, one could generate the header files on a ROS equipped system, and include them in the legacy code. It would be then be necessary to replace the original data types with the ones generated by ROS, or to insert mapping functions between the original legacy data types and the ones generated by ROS when using ROS methods. Another option would be to manually equip the legacy data types with the serialization and deserialization methods required by ROS.

Following the philosophy of leaving the legacy code as untouched as possible, however, ROSlink takes an alternative approach. Topics and service messages used by the objects of *helium* (*legacy::Publisher*, *legacy::Subscriber*, etc.) accept any data type, hence legacy data types can be used directly. Serialization and deserialization between a *helium::Link* and the corresponding *ros::Link* of all data types default to the data type's $<<$ and $>>$ operators, but in case another serialization is desired, it is sufficient to specialize the *helium::write* and *helium::read* functions for that particular data type.

The conversion between the legacy data types and the ROS types is instead performed at the ROS node defining the *ros::Link*. Listings 1 and 2 provide a toy example. The legacy code (listing 1) makes its `addInts` function available as a service over the *legacy::Link*. The function accepts a pair of ints and returns their sum as an int. These types are mapped, respectively, to the request and the response of the service description reported in listing 3. As shown in listing 2, the mapping is specified using the *declareServiceServer* member function of the *legacy::Bridge* object. The first two template

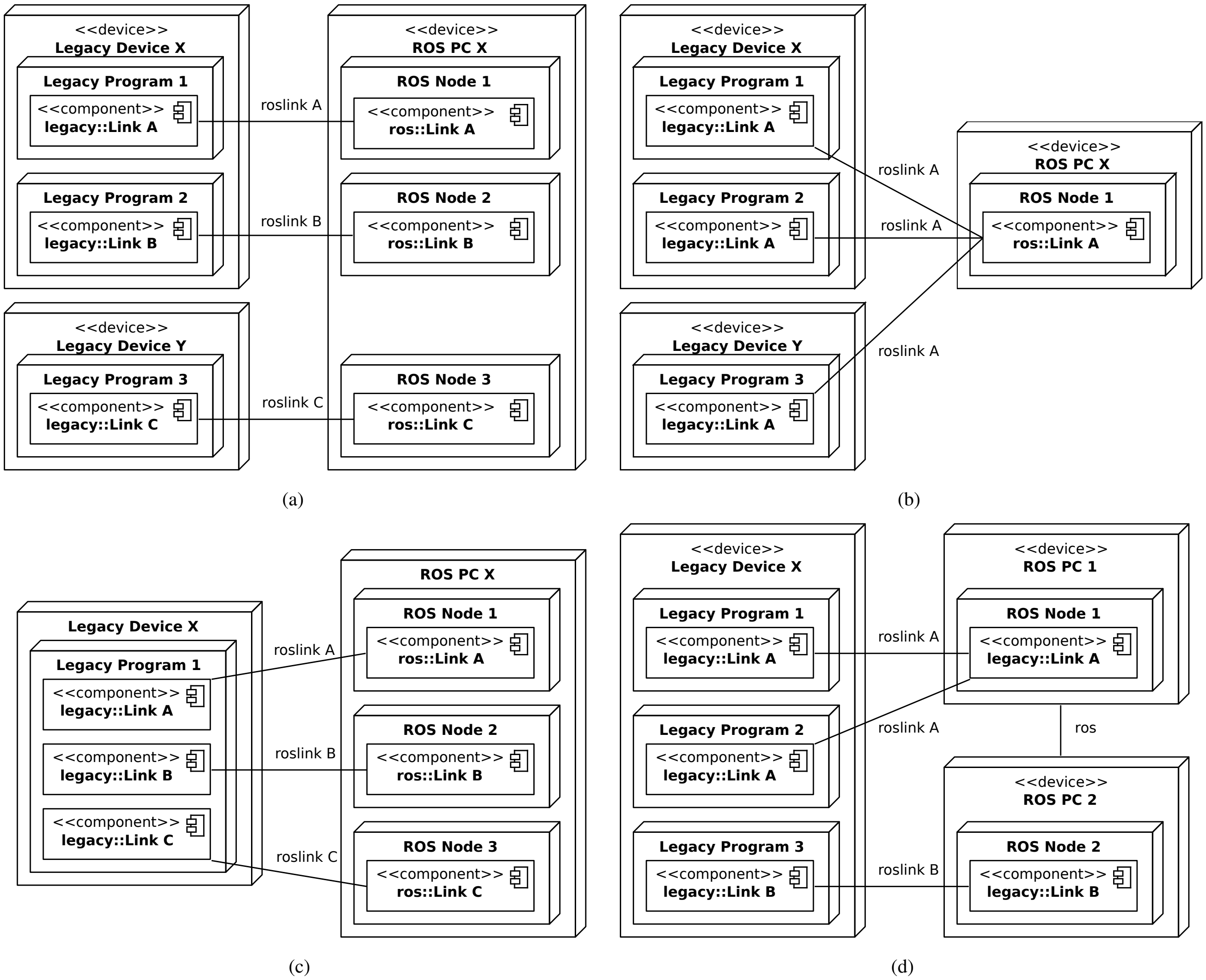

Fig. 1. A UML Deployment diagram of four examples of topology: (a) independent bridges for each legacy program (b) a bridge for all legacy applications (c) a legacy program provides conceptually independent services using different nodes in ROS (d) multiple legacy services are mapped to the same node, while a single bandwidth demanding connection is realized through an independent roslink to another machine

parameters indicate the legacy request and response types, and the following two parameters denote the corresponding ROS request and response types. These can be followed by additional parameters that indicate the classes to be used for the conversion. If left unspecified, the class used for the conversion defaults to the *roslink::DefaultMapper* template class. Listing 2 shows two specialization of this class to actually convert a ROS request into a legacy request, and to perform the opposite conversion for the response. Similar functions (*declareServiceClient*, *declarePublisher*, and *declareSubscriber*) can be used to declare other mappings. Besides letting specify the converted types through its template parameters, each function takes two parameters. The first indicates the topic (or service) to be mapped. The second, optional parameter, is a reference to a *ros::NodeHandle*. This can be used to create different mappings for topics (or services) that have the same name but are located at different locations of the namespace tree. Passing the "∼" node name allows mapping topics/services in the private namespace as well.

Listing 1. Legacy service

```
#include <helium/legacy/legacy.h>

//legacy function
bool addInts(const std::pair<int,int>& in, int& out){
  out=in.first+in.second;
  return true;
}

int main(){
  legacy::Link l("adder_link");
  legacy::NodeHandle n(l);
  legacy::ServiceServer server=
    n.advertiseService("add_two_ints",addInts);
  legacy::spin();
}
```

Listing 2. Mapping node for the legacy service

```
#include <roslink/roslink.h>
#include "roslink/AddTwoInts.h"

namespace roslink{
  //marshalling functions
  template<>
  struct DefaultMapper<roslink::AddTwoInts::Request,
                       std::pair<int,int>
                      >{

    static std::pair<int,int>
    get(const roslink::AddTwoInts::Request& r){
      return std::make_pair(r.a,r.b);
    }

  };

  template<>
  struct DefaultMapper<int,
                       roslink::AddTwoInts::Response
                      >{

    static roslink::AddTwoInts::Response
    get(const int& i){
      roslink::AddTwoInts::Response r;
      r.sum=i;
      return r;
    }

  };
}

int main(int argc,char** argv){
  ros::init(argc,argv,"adder");
  ros::Link l("adder_link");
  l.declareServiceServer
    <std::pair<int,int>,int,
     roslink::AddTwoInts::Request,
     roslink::AddTwoInts::Response
    > ("add_two_ints");
  l.notifyMaster(); // name server registration
  ros::spin();
}
```

Listing 3. Service definition (AddTwoInts.srv)

```
int64 a
int64 b
---
int64 sum
```

III. Test cases

The initial motivation for the development of ROSlink was making M3-Neony [17] control code and its GUI (shown in Fig. 2) available through ROS. Currently, the control code exports 13 services and 9 publishers, while the interface provides 11 publishers and subscribes to 5 topics, but this numbers are going to grow in the immediate future. The legacy code runs on PNM-SG3 from Pinon, a low power consumption (5W) 500 Mhz Geode based CPU board with 512Mb of RAM. The motherboard serial ports, used to communicate with the serial bus of the servomotors, are available only from Windows, and, given the reduced computational power, Windows 2000 was chosen as the OS intalled on the robot. The interface used for the robot's control, a Gtk based GUI, is currently used from Mac OS 10.6, Ubuntu 9.10, and Windows XP machines, and occasionally run inside the robot itself for quick demonstrations or inspections of the robot's state. The current version of ROSlink was proven successful in connecting both the robot's control code and the GUI to new ROS based software, running in ROS electric (on Ubuntu 11.10) and ROS fuerte (on Ubuntu 12.04).

To verify the code portability, ROSlink was then tested on QNX 6.5.0, Windows98 (*helium* compiled with MinGW gcc 4.6.2), and Windows 7 (*helium* compiled with Microsoft Visual C++ 2010 Express). In all these settings, test programs (a legacy publisher, a subscriber, a client, a server and interaction with the parameter server) were compiled and worked successfully. The spectrum of systems in which the ROSlink works is probably much broader. Code is now distributed under GPL license at *http://sourceforge.net/projects/roslink/* , and users are suggested to report successful compilation in other operating systems on the project's wiki or, conversely, to open a ticket if they find difficulties in compiling *helium* on a particular system.

The overhead introduced by ROSlink was then measured. In ROS, apart from the initial communication setup up through *roscore*, the communication between a publisher and its subscribers, or between a client and a server, is direct. In ROSlink, the communication takes place by two hops: from the legacy code to the mapping ROS node containing the *ros::link* object, and from such node to the actual subscriber or server. Roughly speaking, we can thus expect a doubling of the time required for the communication. Figure 3 reports the times measured for each of the ROS communication modalities for processes (legacy code and ROS nodes) running on the same machine, specifically a Ubuntu 12.04 machine powered by an Intel i7-2700K CPU at 3.50GHz and 8Gb of RAM. All the measurements were repeated 1000 times, at intervals of 1 second. Average times are reported with their standard deviation as notes in the figure.

In particular, Fig. 3 indicates how the time is divided among the phases necessary for the communication to take place. For instance, the first row reports what happens when a *legacy::Publisher* sends a message through its *legacy::link* connection to a ROS node that subscribes to the topic. The first 102.9 microseconds are spent for sending the message through the *legacy::link* to the matching *ros::link*. The following 191 microseconds are used by ROS for passing the message from the node containing the *ros::link* to the subscribing node. More precisely, the first portion accounts for the time elapsed from the *legacy::Publisher publish* function invocation to the invocation of the *publish* function of a corresponding *ros::Publisher* automatically created by the *ros::Link* object, while the second portion corresponds to the time spent from the invocation of the *ros::Publisher*'s *publish* function in the *ros::Link*'s node to the execution of the corresponding *ros::Subscriber* callback in the subscriber node.

We notice that the average overhead time is below our time doubling estimation, dropping as low as less than a 20%

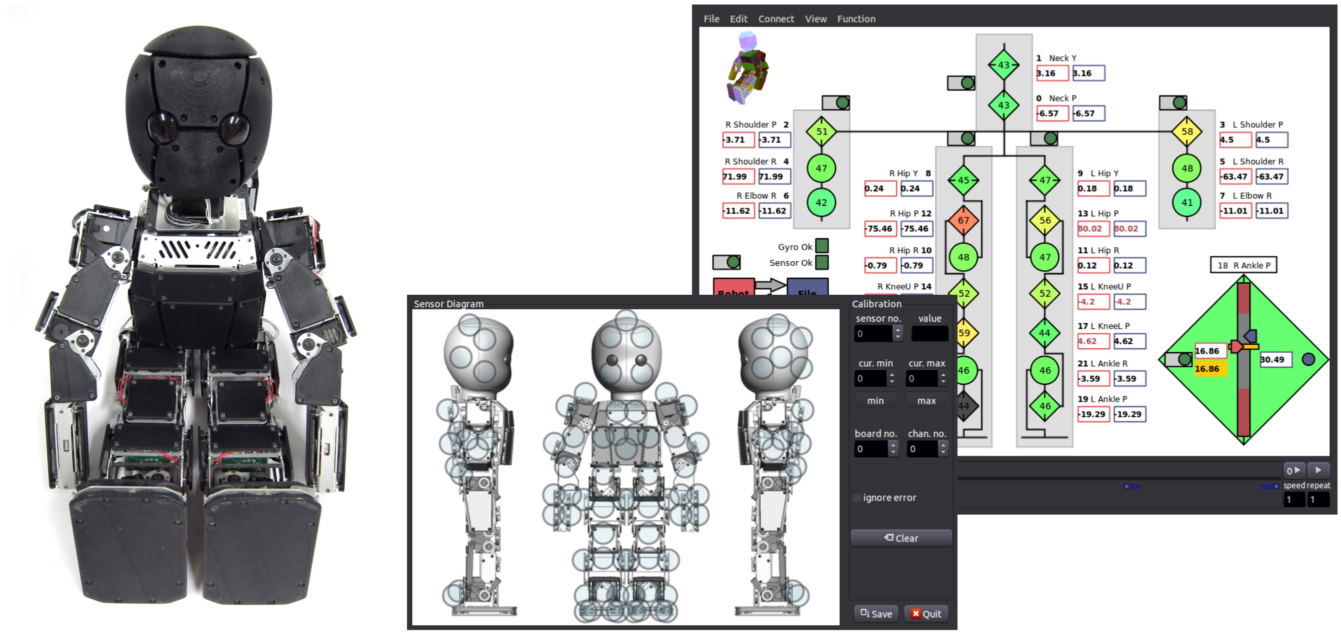

Fig. 2. M3-Neony (on the left), and its control GUI (on the right). Code running inside the robot is used to control its 22 servomotors, and to read the onboard sensors: 90 tactile sensors, 2 gyroscopes, 3 accelerometers, 2 cameras and 2 microphones.

increase for non persistent client-server service calls, where most of the time is spent for establishing the communication between the two ROS nodes.

IV. Conclusion and Future work

In this paper ROSlink, a project aimed at integrating legacy code into ROS systems, was described. Design policies that make it an interesting solution for integrating legacy code into ROS based systems were briefly discussed.

The basic concept underlying ROSlink, named links that bind *legacy::Link* to *ros::Link* objects with the same name, was presented. Two of the advantages of this kind of architecture were highlighted. First, it provides great flexibility in the organization of the data flow, and it allows runtime network topology changes. Second, it is conceptually very intuitive: all the *NodeHandles* (and the associated ROS names) created in the legacy code can be thought as constructed in the ROS node containing the matching *ros::Link*. This, in turns, eases the exploitation of the powerful ROS name remapping system. It is in fact sufficient to act on the ROS node containing a *ros::Link* object to remap the names of the legacy code including the corresponding *legacy::Link*.

The main technical solutions adopted for minimizing the changes required to the legacy system, and, at the same time, for making the API easy to use, were explained. In the development of ROSlink, in particular, it was chosen to provide the legacy system with an API that mirrors the *roscpp* ones. This allows ROS users to use ROSlink without difficulties. Additionally, if the code is moved to a native ROS environment at a second time, the porting process becomes trivial. Another choice taken in ROSlink is keeping the *helium* library, used by the legacy code, self contained, as to remove any possible dependency problem that may arise with the libraries installed in the legacy system. Finally, to minimize the changes required to the legacy code, ROSlink allows any legacy data type to be used as topic message or as service request/response. The conversion into and from data types streamable by ROS is performed outside the legacy system, in the ROS node containing the *ros::Link* object.

The next steps that will be taken for the project development deal with a simple extension of the current functionalities. For instance, it will be straightforward to introduce a set of *ROS_DEBUG*-like macros that, called on the legacy system, use the *ros::Link* to actually output to the ROS default logger, provided by *rosconsole*. Another step to be taken is extending the support of ROSlink to programming languages other than C++. In fact, even with the sole C++ implementation, ROSlink users may compile ROSlink as a shared library and write a thin wrapper for other programming languages, for instance, using SWIG [18]. However, providing native API in other broadly used languages like Python, Java or Lisp would be surely beneficial in speeding up the interconnection of legacy systems and ROS.

For this purpose, ROSlink could be made compatible with rosbridge. In particular, it would be possible to maintain the ROSlink interface exposed to the legacy code and the concept of named links, for keeping the advantages described in this paper. The communication with ROS, instead, could be easily reimplemented using a patched version of rosbridge. This solution was discarded for the initial implementation of ROSlink, as a direct, C++ implementation of both ends of the communication, using a simple custom protocol, allowed to keep the code much simpler and to minimize the overhead. Compatibility with rosbridge, possibly provided as a

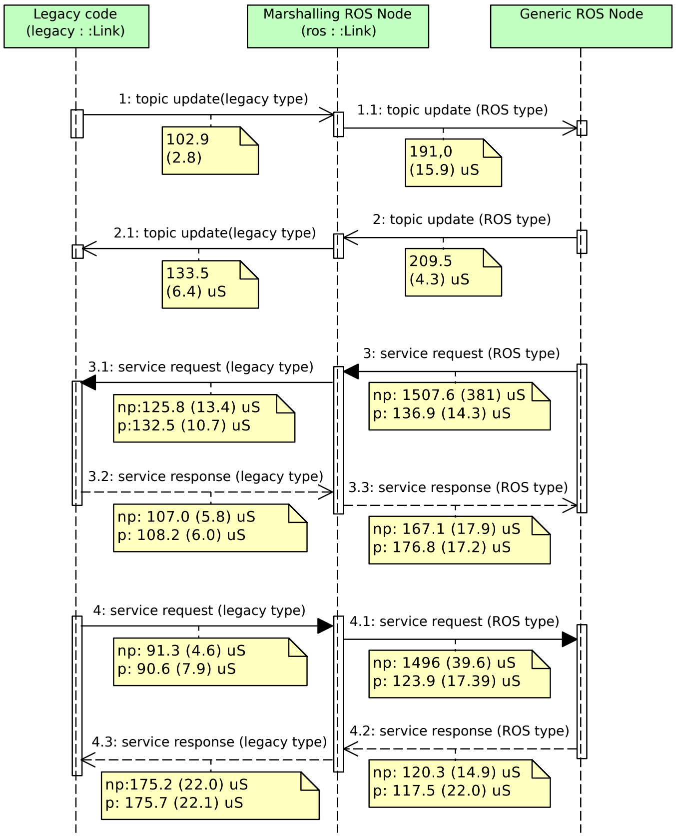

Fig. 3. UML sequence diagram of the communication between a legacy node and a generic ROS node. Notes indicate the average communication time, with its standard deviation in parentheses, obtained with 1000 measurements at 1 second time interval. All the times are given in microseconds. The four possible communication cases are reported: (1) A legacy publisher communicating with a ROS subscriber. (2) A legacy subscriber receiving updates from a ROS publisher. (3) A legacy server called by a ROS node. (4) A service provided by a ROS node called from a legacy program. In the case of service calls, ROS allows to declare the connection between the client and the server as persistent. For service invocations, therefore, the notes report two times: the one obtained with a non persistent connection (np) and the one obtained with a persistent connection (p). The time spent for marshalling, i.e. for the conversion between the ROS types and the legacy types or the other way around, is included in the communication time between the legacy code and the marshalling node for all the communication scenarios. Topics consist in four 64 bit ints, service requests in two 64 bit ints and service responses in eight 64 bit ints, that were progressively filled in with the times measured along the message path.

compilation time option, would however simplify the porting of rosbridge to the languages for which rosbridge already has a client, and would allow ROSlink to benefit from the community that is actively maintaining rosbridge.

REFERENCES

[1] B. P. Gerkey, R. T. Vaughan, and A. Howard, “The player/stage project:tools for multi-robot and distributed sensor systems,” in *Proc. of the 11th Int. Conf. on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal, 2003, pp. 317–323.

[2] J.-C. Baillie, “Urbi: towards a universal robotic low-level programming language,” in *2005 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2005)*, Sendai, Japan, 2005, pp. 820 – 825.

[3] P. Fitzpatrick, G. Metta, and L. Natale, “Towards long-lived robot genes,” *Robot. Auton. Syst.*, vol. 56, no. 1, pp. 29–45, 2008.

[4] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon, “Rt-middleware: distributed component middleware for rt (robot technology),” in *2005 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2005)*, Sendai, Japan, 2005, pp. 3933 – 3938.

[5] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar, “Miro - middleware for mobile robot applications,” *IEEE Transactions on Robotics and Automation*, vol. 18, no. 4, pp. 493 – 497, 2002.

[6] D. Calisi, A. Censi, L. Iocchi, and D. Nardi, “Openrdk: A modular framework for robotic software development,” in *2008 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2008)*, Nice, France, 2008, pp. 1872 –1877.

[7] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, “Towards component-based robotics,” in *2005 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2005)*, Sendai, Japan, 2005, pp. 163 – 168.

[8] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud, “Robotic software integration using marie,” *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 55–60, 2006.

[9] M. Montemerlo, N. Roy, and S. Thrun, “Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit,” in *2003 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2003)*, Las Vegas, NV, USA, 2003, pp. 2436–2441.

[10] H.-S. Park and S. Han, “Development of an open software platform for robotics services,” in *ICCAS-SICE, 2009*, 2009, pp. 4773 –4775.

[11] A. Huang, E. Olson, and D. Moore, “LCM: Lightweight Communications and Marshalling,” in *2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2009)*, Taipei, Taiwan, 2009, pp. 4057 –4062.

[12] T. Galluzzo and D. Kent, “The OpenJAUS Approach To Designing And Implementing The New Sae JAUS Standards,” in *AUVSI Unmanned Systems Conference*, 2010.

[13] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA Workshop on Open Source Software*, 2009.

[14] R. B. Rusu and S. Cousins, “3D is here: Point Cloud Library (PCL),” in *2011 IEEE Int. Conf. on Robotics and Automation (ICRA 2011)*, Shanghai, China, 2011, pp. 1–4.

[15] C. Rich, B. Ponsler, A. Holroyd, and C. Sidner, “Recognizing engagement in human-robot interaction,” in *2010 5th ACM/IEEE Int. Conf. on Human-Robot Interaction (HRI)*, Osaka, Japan, 2010, pp. 375 –382.

[16] C. Crick, G. Jay, S. Osentoski, B. Pitzer, and O. C. Jenkins., “Rosbridge: Ros for non-ros users,” in *Proc. of the 15th Int. Symp. on Robotics Research*, Flagstaff, AZ, USA, 2011.

[17] T. Minato, F. DallaLibera, S. Yokokawa, Y. Nakamura, H. Ishiguro, and E. Menegatti, “A baby robot platform for cognitive developmental robotics,” in *Workshop on ”Synergistic Intelligence” at the 2009 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS 2009)*, St. Louis, MO, USA, 2009.

[18] D. M. Beazley, “Swig: an easy to use tool for integrating scripting languages with c and c++,” in *Proc. of the 4th Conf. on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, Berkeley, CA, USA, 1996, pp. 15–15.