A middle way for robotics middleware

Paul Fitzpatrick Elena Ceseracciu Daniele E. Domenichelli Ali Paikan Giorgio Metta Lorenzo Natale
iCub Facility, Istituto Italiano di Tecnologia, Via Morego, 30, 16163 Genoa, Italy

Abstract—Robotics is changing. The amount of software available (and needed) is growing. For better or worse, the glue that holds that software together, the middleware, has a big impact on its viability. YARP is a middleware for robotics, with over a decade's continuous use on various humanoid robots. YARP was designed to help code survive changes, to easily experiment with new code and integrate with other systems. In a world of constant transition, with a steady stream of hardware and software upgrades, YARP helps code last long enough to make a real impact, and avoid premature loss of good code through middleware muddles. We review the features of YARP that support this flexibility, describing those situations in which they have been practically useful. We argue that there are practices that any middleware author can adopt that benefit users of *other* middleware and raise costs to those users when neglected. Altruism is not required: we also argue a middleware's own users will benefit from such practices right now (when collaborating) and in the future (when upgrading to a new version of the middleware). These concerns are not usually the first things on a user's mind when choosing a middleware, but a responsible middleware author will foresee them and prepare a happy ending rather than a trap.

Index Terms—Robotics, middleware, interoperability, component-based software.

1 Introduction

There is growing attention in robotics to software middleware, some specific to robotics (Player [1], ROS [2], YARP [3], [4], OROCOS [5], Urbi [6], MIRO [7], LCM [8] and MIRA [9]), others more general (ICE [10], CORBA [11], ØMQ [12]). There are several attempts in the literature to compare middleware based on the perspective of its users, looking at features and performance [13], [14], [9]. This is adequate for projects that operate on short time scales of a few months or years. For longer time scales, it is useful to also consider the middleware from an *external* perspective. When users enter in collaboration with a partner who favors a different middleware, and neither party is interested in changing, then suddenly a new set of desirable properties surface. When users of a middleware are faced with a breaking upgrade of that middleware, again there is a new set of desirable properties. Properties of a middleware that have no impact on a user of that middleware when evaluating it for performance may have an enormous impact when they find themselves trying to collaborate with others or trying to continue using their software in the future.

Middleware is a mixed blessing. In the short term it helps a system scale up, but in the long term it can hold it back. Like operating systems, middleware is "sticky" – developers have to make a special effort to make their software usable in other platforms. In the short term, for an individual developer, this is no big deal, since other platforms are irrelevant. However, in the longer term or on larger teams, this is a real cost, paid for through collaborations that never happen or that get bogged down due to platform fragmentation.

A possible solution is for everyone to get together and agree to use a common platform. Even with the best of intentions, a single platform can become a trap, as more and more components come to rely on fragile quirks of that platform, increasing the cost of any changes, even positive ones. This is a problem not just because it raises barriers to new middleware entering a community, but also raises barriers to future versions of the middleware itself. Either there will be a "big bang" change of infrastructure and all the pain that goes with that, or development of the platform will slow to a crawl.

There is a middle way between fragmentation and monoculture. We argue that fragmentation and monoculture are problems only if interoperation is made too expensive. As the cost (in terms of complexity) of interoperation is reduced, then fragmentation creates less waste, and monoculture creates fewer barriers. When thinking about interoperation, standardization and Request For Comments (RFCs) are the

Short paper – *Manuscript received November 11, 2013; revised September 10, 2014.*

- *The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7 ICT) under grant agreements No. 270273 (Xperience) and No. 611832 (WALK-MAN).*

first approaches that come to mind. Standardization definitely has a role to play, and should be encouraged. Nonetheless, the coordination it requires between middleware developers is most practical for "solved problems" rather than aspects of robotics that are still in flux (of which there are many). In this paper, we would like to demonstrate that there are many actions that middleware developers can take unilaterally to reduce the cost of interoperation. We would also like to encourage middleware evaluators to take these aspects into consideration, and ask proper questions. For example: does a middleware assume that the entity on the other side of a network socket is using the same middleware? This would be unfortunate, since sockets are a key opportunity for interoperability. Elaborating further, since this is actually a question of degree, it could be rephrased as: how complex are the steps an external program needs to take to communicate with a middleware-using program without using the middleware's own libraries or utilities? The simpler this is, the lower the cost of interoperation. Further examples: does the middeware lay claim to the main thread of a program? Does it dictate the tooling needed to build/compile a program? Two middleware making such claims will conflict fundamentally. And so on. Again, this is not a call for standardization and elaborate RFCs, just individual developers bearing the world outside their middleware in mind, thus benefiting their community today (by facilitating collaborations) and in the future (by reducing the pain of upgrades, and so making radical improvements more practical).

Our goal in this paper is to draw attention to opportunities and best practices for middleware developers to reduce interoperation costs, or to avoid unwittingly increasing interoperation costs. We focus on steps that can be taken unilaterally, without a standardization process. To be clear, we do not claim these steps are more important than standardization, or good practices by middleware users. They are simply useful steps that can be taken *as well*, and that we have seen omitted often enough to merit mentioning. To be concrete, we draw on examples from YARP ("Yet Another Robot Platform"), a middleware for robotics that has been in continuous use on humanoid robots for more than a decade. Those examples should not be read to imply that YARP is the only middleware to have ever implemented the example feature. The goal of this paper is not to promote YARP or any one middleware, but to encourage longer-term thinking when evaluating middleware amongst users and middleware developers themselves.

2 Diversity in protocols

Middleware typically plays a role in inter-process communication. This is a great opportunity for interoperability. Even when interoperability is not an immediate priority, it is worth taking some simple steps to avoid unnecessarily hobbling it.

For example, most every middleware has some kind of regular TCP/IP-based protocol. Suppose the other side of a socket is *not* controlled by your middleware. They may not know for sure what protocol we are speaking. Just as it is helpful for files lying around in a file system to have a "fingerprint" or "magic number" comprising a recognizable pattern of bytes early in the file, it is helpful if TCP connections do the same.

For example, HTTP responses begin with "HTTP/", and YARP's native protocols start with variants of "YA....RP" or other identifying byte sequences. This makes these types of messages easy to identify. In contrast, TCPROS begins with two 4-byte integers representing lengths, which in principle could have any value. A quick fingerprint up front would be very valuable for simplifying interoperation. If one is expecting a TCPROS connection from a native ROS source, it is possible to verify its nature by communicating with the `rosmaster` name server, so the lack of a fingerprint is not an insurmountable obstacle, but we can report that it did raise the cost of interoperation between YARP and ROS.

There are situations in which a middleware should plan to speak many protocols. Here are situations end-users hit that middleware developers do not see:

- You need to use a novel kind of network not broadly available.
- You badly need to interoperate at the network level with software from a different community.
- You have a problem with some aspect of the current behavior, and the middleware developers just shrug.
- You have a network-aware piece of hardware and you would like to talk to it without bridging.

Indeed, we are not arguing that every middleware should support every protocol present and future. We do however believe that middleware authors have a unique opportunity to foster collaboration without formal standardization, by treating network protocols as something their user communities will choose and, optionally, contribute rather than something defined by the middleware. Every middleware supports certain models of data flow, but it is worth being flexible about how that flow gets expressed concretely on the wire (or even on which kind of wire). Of course, it is reasonable for a middleware to provide default protocol implementations suitable for various scenarios. Yet, it is very helpful if new protocols can take their place easily and without disruption. Making this happen takes careful thought at the core of the middleware.

YARP originally used QNX message passing, so its own true "native" protocol is lost in the mists of time. Having a plugin mechanism for protocols allowed developers and users to extend the original set of protocols and adapt YARP depending on their needs.

In YARP there are basic protocol options operating over tcp, udp, multicast, and shared memory, suitable for different trade-offs between reliability, speed, and bandwidth. In addition YARP supports other standard protocols that are useful for talking to the outside world, such as XMLRPC, HTTP with JSON, MJPEG, and plain text.

Finally there are some notable examples in which specific users' needs have been elegantly and efficiently solved with the implementation of custom protocols.

Bayer carrier

A special purpose protocol plugin was made to carry raw Bayer pattern image streams and decode them on the receiver side, rather than decoding them closer to the hardware. This saves bandwidth without introducing compression artifacts, and it is fully backward compatible, since as far as user code is concerned it receives color images as normal. When there are multiple receivers, it does increase overall CPU usage, which is the main trade-off. As a side benefit, users have the freedom to choose how colors are reconstructed, and this choice can be made per-receiver. A protocol like this is very useful, but so specific to a class of camera that it would be unlikely to get supported in a middleware without a culture of protocol plugins.

ROS compatibility carriers

A set of plugins were added to support YARP-ROS interoperability, via XML/RPC and TCPROS. This is discussed further in Section 6.

Priority carrier

A new protocol was introduced to experiment with the concept of priority arbitration into YARP, so that messages from different origins arriving at the same target could inhibit each other in defined ways. This plugin made use of the fact that plugins can be chained (currently in a rather crude way), so that the new priority mechanism could be overlaid on existing protocols. This is discussed further in Section 9.

3 Diversity in naming

Middleware typically has some mechanism for converting symbolic names into detailed information on how to access a resource (for example, to convert the name "/camera" to "machine 192.168.1.15, port 10012, protocol mjpeg").

We found that naming can become a barrier preventing interoperability and forcing users to resort to inefficient bridging code. With time YARP extended naming support so that it could interface with other systems. At this time it now offers the following options:

- The YARP name server accepts arbitrary registrations, so (for example) a webcam at a random location can be named and read from directly without bridging.
- YARP clients can be directed to use a non-native name server, given an appropriate plugin. For example, there is a plugin for the `rosmaster` name server.
- YARP clients can be directed to use multiple name servers, to better support heterogeneous networks.
- YARP clients can be used without any name server for basic tcp connections where host names and socket port numbers are given directly.

A broad-minded nameserver is a boon for interoperation. Of course the *general* case of naming external resources is full of pitfalls, but most *particular* cases are quite simple.

4 Diversity in type systems

YARP historically has avoided any code generation step, specifically any compilation of an Interface Description Language (IDL) definition to source code. The reasons for this were twofold. Firstly, having an IDL introduces an extra step in the build process which we wanted to avoid to reduce the learning curve for inexpert users. This problem was eventually solved with the adoption of appropriate build tools and consequent automation of code generation and linking (i.e. CMake). The second problem is that IDLs are often a source of incompatibility (and lock-in) among systems. We did eventually find an approach to code generation for YARP that we were happy with, which was simply to support multiple IDLs. By avoiding reliance on a single blessed IDL, we hope to reduce the pain needed to support any future IDLs needed by our users for interoperation.

We assume that IDLs will come and go, and that we may need to support several at a time. So IDL support was added to YARP in the same pattern as for carrier plugins. There is no assumption of uniqueness. So far YARP supports types expressed with the following IDLs:

- Apache thrift. YARP uses the `.thrift` file format and syntax, with its own native serialization. Thrift is appropriate for expressing both structures and procedure calls. The source code for thrift has a plugin mechanism of its own for supporting code generation in different programming languages. We (ab)use this plugin mechanism to generate YARP-specific code.
- ROS `msg` (and `srv`) format and serialization method. This format is adequate for expressing structures, but awkward for expressing procedure calls. We wrote a parser and code generator for it. For convenience, we also wrote a server that can query a ROS installation for the `msg`/`srv` files relevant to a named type, and distribute type information to machines that are not running ROS.

In general some combination of IDL plugins, carrier plugins, name server plugins, and perhaps device plugins may need to work in concert to support comfortable interoperation with another middleware.

5 Example: interoperating with Hawaii

These days, many cameras can connect directly to the network and stream images natively, for example in Motion-JPEG-over-http format. Using such an image stream in a component not written with MJPEG in mind typically requires a bridge –

an intermediate program that does nothing but take in MJPEG images and spit out "native" format images. Bridges introduce latency, complicate the control network, and need to be figured out by each new user of the component. YARP takes a different approach of folding bridge-style code into plugins that hook directly into the source/sink where they are needed. This keeps the control network simple, avoids the latency of a non-local relay, and can be deployed in a more systematic way. For the example of IP cameras, YARP has an `mjpeg` plugin that makes this format a first-class wire protocol for YARP. So for example, if we wanted to hook up a random webcam in Hawaii to a YARP image viewer, we could enable that plugin and do:

```
# Start a YARP image viewer
yarpview /view &
# Make connection using mjpeg protocol
yarp connect /67.52.88.202:5159 /view \
  mjpeg+path.mjpg/video.mjpg?camera=1
```

And we immediately see Hawaii[1]. The `yarpview` program itself knows nothing about mjpeg, but a tcp connection is coming in from Hawaii, going directly to a component input expecting images, is accepted by middleware code as Motion-JPEG-over-http, and is decompressed appropriately.

Is this really so different to a bridge? Are we not just changing the level in the middleware at which "foreign" data is read or written? Yes, but this choice has real consequences. With bridging as a separate process, foreign protocols must pay a serialize-transport-deserialize tax. It is true that in many cases that tax will be small, especially on well-configured modern kernels. Still, there are cases where it will be large. Think "embedded" or "huge messages" or "non-TCP" or "encrypted" or any of many interesting directions in which users may be pushing the limits. The point is not that the tax can be large, it is that we (middleware designers) should not be forcing our users pay it. The logic is similar to operating system development: these days, many applications can get away without being optimized for size or speed, but responsible operating system developers make architectural choices with the demanding cases in mind, not the easy ones. Another downside to bridges as a separate process is that starting two processes instead of one inevitably introduces more ways for things to go wrong or be misconfigured. Again, this can be automated away, but is still a tax on every use of a foreign protocol.

6 Example: Interoperating with ROS

ROS, the Robot Operating System, is an extensive framework for robotics, popularized initially by Willow Garage and now stewarded by the Open Source Robotics Foundation.

ROS presents some specific difficulties for interoperation. It includes an elaborate packaging mechanism that has, at least to date, made installation of the basic middleware burdensome outside of the specific environments its creators support. So, when implementing methods for interoperation, we need to take care that we isolate any dependency on ROS packages carefully, to avoid burdening the entire YARP network (specifically, all the machines it runs on) with a requirement to have a working ROS installation.

Here are the steps we have taken, in chronological order, towards ROS interoperability:

- Support for the XML/RPC protocol was added to YARP, as a new carrier plugin. This is relevant for ROS since it is the protocol spoken by the `rosmaster` name server, and by an administrative server associated with each ROS "node."
- Support for the TCPROS protocol (spoken by publishers, subscribers, and services) was added as a new carrier plugin. Since ROS has a type system with side-information not included in connections, the carrier cannot automatically match types with existing YARP types – that part comes later, read on.
- An application for translating ROS .msg/.srv files into YARP-using serialization code was added.
- A helper utility to generate .msg files automatically by sniffing existing YARP connections was added.
- Support was added for `rosmaster` as an alternate name server to YARP's native name server. This made it possible for a YARP program to be completely interchangeable with a ROS program, if desired.
- An implementation of a "type server" was added to allow ROS type information to be distributed across the network to wherever it was needed, without requiring ROS installations on all machines.
- An analogue of ROS nodes was added to YARP, to bundle a set of ports together. For existing YARP code, node-port associations can be specified via a naming convention for ports, requiring no source-code modifications.
- A notion of data flow direction was added to YARP. YARP ports are not by default committed to any particular pattern of data flow. However, to know whether a port corresponds to a ROS subscriber, publisher, service provider or service user, we need to know what direction data flows and if there are replies. For compatibility with existing YARP code, port-dataflow associations can be set up via a naming convention.
- A notion of external typing was added to YARP. YARP ports are not by default committed to any particular type, and messages are JSON-like in content. To know what kind of data is flowing via a port, in order to present or interpret it correctly to/from ROS, we need to introduce type information.
- A method for mapping ROS wire formats to and from

1. The `yarp connect` command can be read as saying: connect from the given machine at port number 5159, to the local port called `/view`, speaking MJPEG. The MJPEG plugin requires an additional parameter, a path to complete the URL to the webcam, in this case `mjpg/video.mjpg?camera=1`

YARP messages was added, so that many (but not yet all) ROS topics could be connected to existing YARP programs.

Let's look at a concrete example[2]. Suppose we want to run YARP program within a ROS network. First, we tell YARP to locate and use ROS's `rosmaster` name server:

```
# find ROS name server, respecting ROS_MASTER_URI
yarp detect --ros --write
# give access to ROS type info to ROS-less machines
yarpidl_rosmsg --name /typ@/yarpros
```

The `yarpidl_rosmsg` program is a server[3] that gives remote access to ROS's type system, and should be run on a machine that can run the `rosmsg` and `rossrv` utilities.

In regular YARP usage, there is a utility called `yarp read` which reads data from the network and prints it to the console, invoked as follows:

```
yarp read /msg
```

YARP has port names that do not match ROS exactly, combining aspects of ROS nodes, topics, and services. We added an `@` syntax to model ROS node/topic naming. If we start `yarp read` as follows:

```
yarp read /msg@/test_node
```

We now have a node called `/test_node` and a topic called `/msg` which can be inspected with utilities such as `rosnode` and `rostopic`. As invoked, the `/msg` topic is not committed to any particular type. We can send it messages of various kinds from ROS:

```
rostopic pub /msg std_msgs/String "hello yarp"
rostopic pub /msg turtlesim/Pose 0 5 10 15 20
```

Basic types like `std_msgs/String` are recognized by YARP's TCPROS carrier plugin and can be translated without any external assistance. In general, there is not enough information "on the wire" to map arbitrary ROS messages onto YARP equivalents, and in these cases the type server will be consulted. So for `turtlesim/Pose` the type server will be consulted, and these two very different messages will print out just fine.

As an example of data flow in the opposite direction, suppose we run the standard iCub robot simulator and then try to view the simulated camera output in ROS's `image_view` utility:

```
iCub_SIM
rosrun image_view image_view image:=/icubSim/cam/left
```

2. The examples in this paper were tested with YARP version 2.3.63, available at https://github.com/robotology/yarp, and ROS Groovy.

3. The `--name` argument here is assigning the server a topic name of `/typ` (this is currently required), and a node name of `/yarpros` (this is arbitrary). The naming syntax is explained in detail in the remainder of this section.

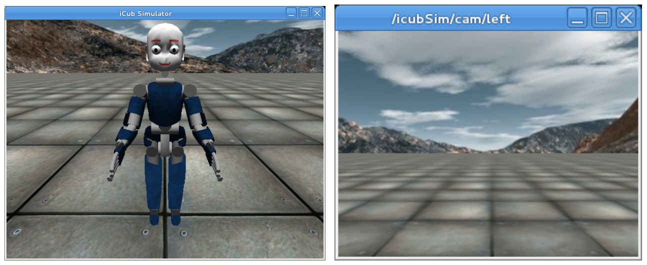


Fig. 1. An unmodified iCub simulator (left), streaming images directly to a ROS `image_pipeline image_view` instance (right). The images depart a TCP socket opened by the simulator in ROS `sensor_msgs/Image` format serialized to a `TCPROS` connection. No extra copy is made of the image, represented in the simulator in a non-ROS structure. The simulator source code is untouched by any of this, and no explicit bridging node is needed.

To identify a publisher/subscriber uniquely within ROS, we need a node name and a topic name. YARP ports are not natively named this way, but rather they have a single name and can be communicated with in many ways (not just publish/subscribe). For ROS compatibility, it is simplest for YARP ports to follow its naming scheme, which has been supported through an extension to the port name syntax. This is convenient since ports can be renamed in this way without touching source code. Specifically, in this case, we decorate names of camera ports (in `Sim_camera.ini`) with a node name and data flow direction for ROS's benefit, by adding `+@/icubSim` (the `+` gives data flow direction). Those ports will now show up as ROS publishers associated with a node called `/icubSim`. That is the only change needed (see Figure 1). The same method could be used to access the physical iCub robot.

7 Example: Talking to Web Apps

YARP ports can be contacted at any time in any of many protocols, including HTTP and JSON. Suppose the default YARP name server is listening on port 10000 on 192.168.1.2. All YARP ports are accessible via HTTP for browsing, AJAX requests, or streaming. For example, from a webpage we could request the name server to look up a port called `/icubSim/head/rpc:i` (this is the default name of a simulated controller for the robot head) via the following:

```
http://192.168.1.2:10000/?req=query+$port
                         &port=/icubSim/head/rpc:i
```

The `req` parameter determines the type of message we are sending (a name query in this case), and then any other parameters needed for that message are supplied separately (in this case the port name we want to look up). We will get back a result in JSON, easily consumed by a javascript application:

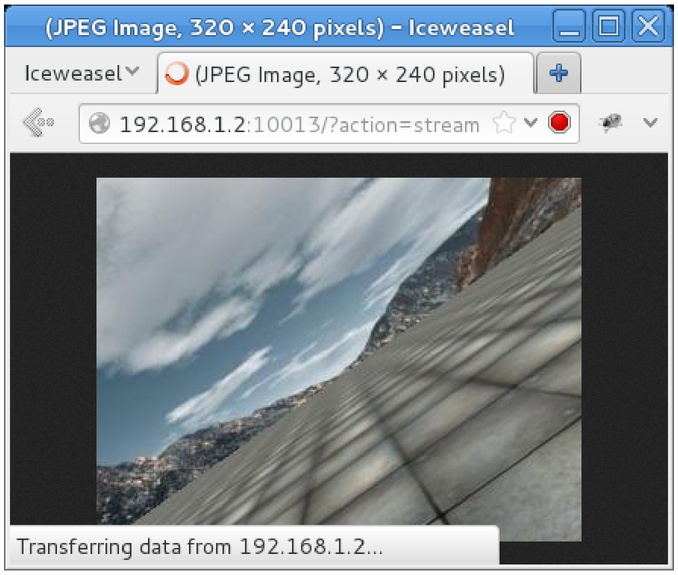

(JPEG Image, 320 × 240 pixels) - Iceweasel
Iceweasel
(JPEG Image, 320 × 240 pixels)
192.168.1.2:10013/?action=stream
Transferring data from 192.168.1.2...

Fig. 2. Viewing the simulator's camera output in a browser ("Iceweasel" is Firefox on Debian), via MJPEG (not all browsers support this). The image stream leaves a TCP socket opened by the simulator in Motion JPEG over HTTP format. The simulator source code is untouched by any of this, and no explicit bridging node is needed.

```
{
  "type": "port",
  "name": "/icubSim/head/rpc:i",
  "ip": "192.168.1.2",
  "port_number": 10044,
  "carrier": "tcp"
}
```

And now that we know how to contact the head controller, we can send it a request to tilt the head to one side:

```
http://192.168.1.2:10044/?req=set+pos+$axis+$angle
                          &axis=1&angle=45
```

We can check encoder values to make sure this did something:

```
http://192.168.1.2:10044/?req=get+encs
```

The result will be something like this:

```
[
  "is",
  "encs",
  [-0.000031, 45.000034, -0.0, 0.0, 0.0, -0.0],
  "ok"
]
```

(Note the 45° angle in the result). Alternatively, we could have asked the name server for an address of a camera port and then viewed the camera output via MJPEG over HTTP:

```
http://192.168.1.2:10013/?action=stream
```

The `?action=stream` parameter triggers YARP's MJPEG plugin. Figure 2 indeed shows a tilted head perspective.

8 Example: Talking to a Human

YARP is designed to have a low barrier to entry for programmers wishing to communicate with a YARP-using program *without using any YARP libraries or utilities*. If they can figure out how to open a socket, they will be able to read and write data to and from YARP ports within minutes. We certainly do not suggest that absolute beginners should be doing this: it is intended to lower costs to our users and the users of other middleware when they hit integration problems.

If a socket connection is opened to a YARP port, and garbage is sent, a helpful human-readable message is returned showing how to make a real connection. For the sake of example, suppose we are just using `telnet` or `netcat` from the command-line.

```
$ telnet localhost 10000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
wepfokwoefp
* Error. Protocol not found.
* Hello. You appear to be trying to communicate
  with a YARP Port. The first 8 bytes sent to a
  YARP Port are critical for identifying the
  protocol you wish to speak. The first 8 bytes
  you sent were not associated with any particular
  protocol. If you are a human, try typing
  "CONNECT foo" followed by a <RETURN>.
  The 8 bytes "CONNECT " correspond to a simple
  text-mode protocol.
* Goodbye.
Connection closed by foreign host.
```

Subsequent help shows how to start manipulating the port. All ports respond to a text protocol, which is easy for a human to read and write, and adequate for lower bandwidth applications. For example, here is how we could move and query the iCub simulator from the command-line (with `telnet` or `netcat`) without using any YARP-specific code:

```
{
  # ask for a text-mode connection
  echo CONNECT test
  # send a request to move axis 1 to 45 degrees
  echo d
  echo set pos 1 45
  # wait a second for robot to move
  sleep 1
  # ask for encoder readings
  echo d
  echo get encs
  # pause telnet to see results
  sleep 1
} | telnet 192.168.1.2 10044
```

A socket connection opened to a YARP port in its native protocols (including the text protocol) can be used to read *or* to write a stream. The direction of data flow is not tied to which side initiated the connection (as it is in many protocols, including TCPROS). This eliminates the problem of having to give the middleware a way to "call you back," bypassing the issue of naming external entities and leaving the user free

to implement a simple client rather than being compelled to make a server to listen for connections.
YARP is designed to be super-hacker-friendly. Users planning to use it on an open network should take extra sectuity steps. such as wrapping it up in a VPN.

9 Other carriers

Carrier plugins have proven handy for all sorts of reasons. For example, a YARP user[4] substituted YARP's regular point-to-point and broadcast protocols with alternatives using MPI. Another application that came up was distributing images from a firewire camera prior to debayering them, to save bandwidth (see Section 2). The plugin mechanism was also convenient while developing an upgrade for the shared-memory carrier: the two versions of the carrier could co-exist without disruption, thus facilitating the testing phase.

More recently, we have come to realize that carriers can serve much more general purposes than just fiddling around with wire formats and the like, especially once carrier chaining is permitted. We successfully added a "priority" carrier that can be attached to any other existing carrier, in order to allow incoming messages to be prioritized and to override each other. For example, we could take an instance of the `yarpview` image viewer program, and send it two image streams and have the priority carrier apply rules to favor one over the other when available. We could also set up a control signal that modifies those rules dynamically [15].

Again, all the logic takes place within the `yarpview` component's process space (although it is specified outside it), and all the network traffic goes directly to that component rather than to some intermediate bridge.

The priority carrier is a first step in a very interesting direction, where components can be adapted in sophisticated ways without paying too great a tax in extra latency, bandwidth, and complexity. We have recently developed a more generalized carrier modifier that can bring arbitrary transformations (in the Lua programming language) to ports, which may finally make it practical for users themselves to be creative in adapting components right at the source [16], [17].

10 Conclusions

Most literature on software middleware for robotics focuses on comparing performance, measured as speed, latency or bandwidth. This is useful, but we feel it misses an important point: support for interoperability. In the real-world middleware users rely on custom bridges to interconnect heterogeneous systems. The performance of bridges may well be fine for the application, but would not win any awards for latency or bandwidth. What is worse is that bridges solve specific problems, in that they perform the transformations and operations required for interoperating individual connections. For this reason they pose maintenance problems.

We hope with YARP to burn a lot of bridges, or at least make them unnecessary, and to find ways to let users transform inputs and outputs in more efficient and generic ways.

Who should care about interoperability between middleware? Everyone who works in robotics for more than a year or two. We need at least confidence that the middleware we use can interoperate with a future version of itself, so our users and their collaborators do not end up in a middleware muddle. In the fast and ever evolving world of robotics, a good middleware must be designed for change and, above all, interoperability.

4. Daniel Krieg, then at the Frankfurt Institute for Advanced Studies.

References

[1] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323. 1

[2] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009. 1

[3] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: Yet Another Robot Platform," *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 43–48, March 2006. 1

[4] P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robotics and Autonomous Systems*, vol. 56, no. 1, pp. 29–45, 2008. 1

[5] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3, 2001, pp. 2523 – 2528 vol.3. 1

[6] G. Technologies, "Urbi," http://http://www.urbiforge.org. 1

[7] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar, "Miro - middleware for mobile robot applications," *Robotics and Automation, IEEE Transactions on*, vol. 18, no. 4, pp. 493–497, Dec. 2002. [Online]. Available: http://dx.doi.org/10.1109/TRA.2002.802930 1

[8] A. Huang, E. Olson, and D. Moore, "LCM: Lightweight communications and marshalling," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, October 2010. 1

[9] E. Einhorn, T. Langner, R. Stricker, C. Martin, and H. Gross, "MIRA - middleware for robotic applications," in *International Conference on Intelligent Robots and Systems*, 2012. 1

[10] Z. Inc., "Internet communications engine," http://zeroc.com/ice.html. 1

[11] OMG, "Common Object Request Broker Architecture (CORBA/I-IOP).v3.1," OMG, Tech. Rep., Jan. 2008. [Online]. Available: http://www.omg.org/spec/CORBA/3.1/ 1

[12] "ØMQ, The Intelligent Transport Layer," http://www.zeromq.org/. 1

[13] J. F. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Auton. Robots*, vol. 22, no. 2, pp. 101–132, 2007. 1

[14] N. Mohamed, J. Al-Jaroodi, and I. Jawhar, "Middleware for robotics: A survey," in *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, sept. 2008, pp. 736 –742. 1

[15] A. Paikan, G. Metta, and L. Natale, "A port–arbitrated mechanism for behavior selection in humanoid robotics," in *The 16th International Conference on Advanced Robotics*, 2013, pp. 1–7. 9

[16] A. Paikan, P. Fitzpatrick, G. Metta, and L. Natale, "Data flow port monitoring and arbitration," *Journal of Software Engineering for Robotics*, vol. 5, no. 1, pp. 80–88, 2014. 9

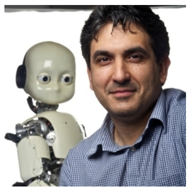
[17] A. Paikan, V. Tikhanoff, G. Metta, and L. Natale, "Enhancing software module reusability using port plug–ins: an experiment with the iCub robot," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014. 9

Paul Fitzpatrick has a background in artificial intelligence, robotics, and engineering, with a PhD in Artificial Intelligence from MIT, and a BEng in Computer Engineering from the University of Limerick, Ireland. He has worked as a software engineer in the fields of robotics, process control and automation, and static verification. Paul grew up on a small farm in Ireland. There were goats involved. He now lives in Montclair, New Jersey. There are fewer goats involved.

Elena Ceseracciu holds a MS with honors (2007) and PhD (2011) in Bioengineering, both from the University of Padova, Italy. She spent six months at Stanford University in 2007 as a visiting student. From 2012 to 2014 she was a post-doc at the Italian Institute of Technology, working on software integration for the Xperience European Project. Now she is a post-doc at the University of Padova within the BioMot European Project, working on improving exoskeleton technology for rehabilitation purposes.

Daniele E. Domenichelli holds a Laurea degree in Biomedical Engineering (in 2007) and a PhD in Informatics, Electronics, Robotics and Telecommunication Engineering (in 2011) both from University of Genoa, Italy. He worked at Nice S.R.L. from 2011 to 2012. He is currently a post-doc at Italian Institute of Technology, iCub Facility. His main interests include open source software architectures, and middleware for inter process communication, collaboration and tool integration.

Ali Paikan is currently a Postdoctoral researcher at the Istituto Italiano di Tecnologia (IIT) in the iCub Facility department. He accomplished his Ph.D. in Robotics at the IIT in 2014 and he received a double M.S. degree from the University of Genova, Italy in 2010, and from the Ecole Centrale de Nantes, France in 2009, within the joint European Master on Advanced Robotics (EMARO). During his research period (2005 - 2007) at the Mechatronics Research Laboratory (MRL), Iran, Ali was actively involved and awarded in various RoboCup competitions. He has an extensive background in robotics and his main research interests include software architectures for robotics, software reusability and real-time systems.



Giorgio Metta is the director of the iCub Facility, at the Istituto Italiano di Tecnologia. He holds a MSc cum laude (1994) and PhD (2000) in electronic engineering both from the University of Genoa. From 2001 to 2002 he was postdoctoral associate at the MIT AI-Lab . He is assistant professor at the University of Genoa since 2005 and with IIT since 2006. He is Professor of Cognitive Robotics at the University of Plymouth (UK) since 2012. Giorgio Metta's research activities are in the fields of biologically motivated and humanoid robotics and, in particular, in developing humanoid robots that can adapt and learn from experience. He is an author of approximately 250 publications. He has been working as principal investigator and research scientist in about a dozen international as well as national funded projects.

Lorenzo Natale received his degree in Electronic Engineering (with honours) in 2000 and Ph.D. in Robotics in 2004 from the University of Genoa. He was postdoctoral researcher at the MIT Computer Science and Artificial Intelligence Laboratory. He was later Team Leader at the Istituto Italiano di Tecnologia (IIT) where he currently holds a position as Researcher. In the past years Lorenzo Natale worked on various humanoid platforms. His research interests range from sensorimotor learning and perception to software architectures for robotics. He has been key investigator in several EU funded projects and hs is author of more than 70 papers in international peer-reviewed journals and conferences.