

## YARP, a Thin Middleware for (Humanoid) Robots

Paul Fitzpatrick, Lorenzo Natale and Giorgio Metta

Italian Institute of Technology  
Via Morego, 30  
16163 Genova, Italy

### Abstract

YARP stands for “Yet Another Robot Platform.” It is a robot middleware that began life as a thin layer over the QNX real-time operating system to adapt it for use by humanoid robots. It is now used on all kinds of operating systems and robots around the world. It is free and open software, released under the LGPL. Over the past decade, YARP’s communication model has proven to be at a sweet spot that combines efficiency, flexibility, and ease of use.

YARP is used to build robot control systems as a collection of programs communicating in a peer-to-peer way, using an open-ended family of connection types (TCP, UDP, multicast, local, shared memory, MPI, mjpeg-over-http, XML/RPC, TCPROS, plain-text, etc.) that can be swapped in and out as need dictates. It also supports similarly flexible interfacing with hardware devices. The strategic goal of YARP’s developers is to increase the longevity of robot software projects (Fitzpatrick, Metta, and Natale 2008).

YARP’s first version coalesced in 2000. It was shaped by the problem of doing productive research despite constant flux in our robot platforms (hence the name). Since then, experience with incompatible architectures, frameworks, and middleware – which we like to call collectively “muddleware” – has taught us to make YARP a *reluctant* middleware, with no desire or expectation to be in control of a user’s system. Some see YARP’s restraint as a negative, but for others it is appealing; a typical user comment in a recent survey was: “It’s lightweight and easy to use. I like the multi-platform support, and its non-monopolistic philosophy.”

Communication in YARP generally follows the Observer design pattern (Gamma et al. 1995). `Port` objects deliver messages to any number of observers (other `Ports`), in any number of processes, distributed across any number of machines, using any of many underlying communication protocols. `Ports` can be connected on an individual basis or as topic-based groups. A message from a single `Port` may be sent simultaneously across multiple connections using distinct protocols. `Ports` can be connected to non-YARP network entities, such as IP cameras, web servers,

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.



Figure 1: YARP works on complex humanoids such as the iCub, on embedded systems, and on everything in between.

or ROS nodes (Quigley et al. 2009). All of this can require some fancy footwork by YARP, such as grouping multicast connections, or coordinating with a foreign middleware. YARP has a similarly flexible device interface. Individual YARP-using programs can be upgraded over time to deal with changes in hardware or networking without touching user source code, and without needing to make a “big bang” change of all programs at once.

### Stay out of my build

YARP was born in 2000 on an early humanoid robot (called Kismet) controlled by a set of Motorola 68332 processors, an Apple Mac, and a loose network of PCs running QNX, Linux, and Microsoft Windows (Metta, Fitzpatrick, and Natale 2006). Communication on this robot was an ad-hoc mixture of dual-port RAM polling, QNX message passing, CORBA, and raw sockets. From the beginning, YARP has been built with the implicit assumption that it is just one part of the users’ environment. So YARP restrains itself to behave like any other library, such as OpenCV, ODE, etc., rather than expecting any special treatment.

But there’s a good reason why most middleware are more than just regular libraries: marshalling/demarshalling. To

send user data between programs, that data must be translated to and from its format in memory to its format “on the wire.” It is commonplace to write a parser to convert a user’s structure description into generated code for performing this conversion. YARP has historically skirted around this approach, for two main reasons. First, it is invasive, since it insinuates the middleware into the user’s build toolchain (imagine how that much fun trying to use *two* such middleware is). Second, it can be inefficient, potentially leading to time wasted making copies of large data structures (YARP goes to great lengths to make sure that large structures such as images get transmitted with zero extra copies made). As computer scientists, it is hard to resist adding automation for this, but the cost of meeting that automation’s requirements should be borne in mind. A quote from a user:

“Compared to, e.g., CAVIAR and Psyclone, YARP looks like a fairly standard library - neither does it do its own message scheduling nor does it provide heavy-handed semantics for message definitions or networking. That may be its very strength.” (Stefánsson, Jonsson, and Thórisson 2009)

### The telnet test

We’ve found that a useful way to evaluate a middleware is this: can a user monitor and insert traffic between programs using just a telnet client? The telnet program can open (more or less) raw TCP sessions, and is commonly used to manually send messages to servers speaking SMTP, HTTP, IRC etc. We started applying the telnet test to YARP out of past frustration with other middleware, where simply passing a few numbers to a collaborator’s program required jumping through a dozen hoops. We therefore (following the “golden rule”) make it easy for others to send a few numbers to a YARP-based program without having to dig through protocol specifications, or link against our libraries, or use our build machinery – all of which can be a time-sink. This is similar to Google’s “data liberation front,” a subproject to ensure that it is easy for users to migrate their data from a service, avoiding lock-in.

A YARP network is designed to be usable without YARP. This doesn’t happen by accident. For example, we took care that YARP connections could be initiated by either the “sender” or “receiver,” with the logical flow of data being freely reversible. This is important for supporting a wide range of protocols, which may be *pull* or *push* in nature, but it also means a foreign program can both send input to and read from a YARP program without getting stuck writing a server for at least one of the directions of data flow. Examples of making YARP connections without using the YARP libraries are available in C, Python, and Tcl, and of course for telnet. Users seem to value the interoperability that YARP provides:

“YARP was chosen as the communication library with which all communication protocols were implemented as one of the goals of the design of the communication stack was to make it possible to interact with programs that are developed without using MeRMaID.” (Barbosa, Ramos, and Sequeira 2009)

## Conclusions

Why *not* use YARP? YARP’s commitment to portability slows its growth, since taking on any new dependency is complicated. A full native implementation of YARP exists only in C++; we rely on SWIG for wrappers in other languages. Lack of an interface definition language (IDL) and associated code generation can lead to some tedium implementing classic RPC-style code. YARP currently uses a central name server to do match-making, which can be problematic for applications such as modular robotics where there is no clear central hub<sup>1</sup>. The LGPL license that YARP is under is commercial-use friendly, but still could complicate certain proprietary uses of the library.

Why use YARP? If you’re looking for something portable, light-weight, and flexible, then YARP is it. YARP serves a truly diverse, interdisciplinary community. While YARP grew up on high-end humanoids with lots of resources, it works fine on embedded systems. It has a flexible and open model of connections that has stood the test of time. It places no constraints on the user’s build system (though we are big fans of CMake). And it passes the “telnet test,” a property of just about every successful, durable network protocol or format.

## References

- Barbosa, M.; Ramos, N.; and Sequeira, J. 2009. MeRMaID middleware for multiple robot intelligent decision-making. *Journal of Software Engineering for Robotics* 1(1):1–15.
- Fitzpatrick, P.; Metta, G.; and Natale, L. 2008. Towards long-lived robot genes. *Robotics and Autonomous Systems* 56(1):29–45.
- Fitzpatrick, P.; Metta, G.; and Natale, L. 2010. The CMaking of a humanoid. In *Kitware Source: Software Developer’s Quarterly*, number 13. Kitware. 7–9.
- Gamma, E.; Helm, R.; Johnson, R.; and Vlisside, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley.
- Huang, A. S.; Olson, E.; and Moore, D. 2010. LCM: Lightweight communications and marshalling. In *Int. Conf. on Intelligent Robots and Systems*.
- Metta, G.; Fitzpatrick, P.; and Natale, L. 2006. YARP: Yet Another Robot Platform. *International Journal of Advanced Robotic Systems*.
- Quigley, M.; Conley, K.; Gerkey, B. P.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. Y. 2009. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.
- Stefánsson, S. F.; Jonsson, B. T.; and Thórisson, K. R. 2009. A YARP-based architectural framework for robotic vision applications. In *Proc. of International Conference on Computer Vision Theory and Applications*, number 1, 65–68.

<sup>1</sup>LCM (Huang, Olson, and Moore 2010) has an interesting solution to this: it commits to UDP multicast for all messages, with broadcast messages filtered by clients; in this case, a central name server is not needed, reducing a point of failure.