Towards Long-Lived Robot Genes

Paul Fitzpatrick [a] Giorgio Metta [a,b] Lorenzo Natale [a]

[a] *Italian Institute of Technology*
Via Morego, 30
16163 Genova, Italy

[b] *LIRA-Lab, University of Genoa*
Viale F. Causa, 13
16145 Genova, Italy

Abstract

Robot projects are often evolutionary dead ends, with the software and hardware they produce disappearing without trace afterwards. Common causes include dependencies on uncommon or obsolete devices or libraries, and dispersion of an already small group of users. In humanoid robotics, a small field with an avid appetite for novel devices, we experience a great deal of "churn" of this nature. In this paper, we explore how best to connect our software with the mainstream, so that it can be more stable and long-lasting, without compromising our ability to constantly change our sensors, actuators, processors, and networks. We also look at how to encourage the propagation and evolution of hardware designs, so that we can start to build up a "gene-pool" of material to draw upon for new projects.

We advance on two fronts, software and hardware. Building on our robot software architecture YARP [11], we focus on how to organize communication between sensors, processors, and actuators so that loose coupling is encouraged, making gradual system evolution much easier. We develop a model of communication that is transport-neutral, so that data flow is decoupled from the details of the underlying networks and protocols in use (allowing several to be used simultaneously, key to smooth evolution). We develop a methodology for interfacing with devices (sensors, actuators, etc.) that again encourages loose coupling and can make changes in devices less disruptive. At the same time, we are concerned with the problem of incompatible architectures and frameworks, and discuss how we work around this.

We emphasize the strategic utility of the Free Software social contract [15] to software development for small communities with idiosyncratic requirements. We also work to expand our community by releasing the design of our ICub humanoid [23] under a free and open license, and funding development using this platform.

Key words:
humanoid robotics, free software, device drivers, ICub humanoid, YARP

1 Introduction

Robotics development is, in some ways, like natural evolution. Consider robot software. Every piece of software has its niche: the environmental conditions within which it can be used. Within this niche it will grow and change and perhaps expand to nearby niches. Some niches are large (standard PCs), some are medium-sized (for example robots like Khepera [19], Pioneer [17] and AIBO [21] to mention a few), and some are tiny (a newly developed humanoid). Software evolves quickly as new technologies get proposed and hardware changes; if trapped in too narrow a niche it tends to become obsolete and die, together with the efforts of the developers who have contributed to it. Robot hardware is subject in turn to the wider commercial and industrial environment. In academia, software and hardware designed for robotic projects are prone to obsolescence, because although graduate students may be talented developers they are rarely experienced and disciplined system engineers. Also, often the development of a robotic platform is not the main goal of the efforts of the people who are working on it but simply a means to an end. For such researchers hardware and software development are time consuming and tedious tasks that take away time and energy that could be better spent doing research. Yet at the same time, the design of a robotic platform is a delicate and crucial task that cannot be easily delegated to untrained personnel. In research laboratories fast changing hardware and lack of human resources too often narrow the niche in which robotic platforms live.

In this paper, we are concerned about how robotics researchers can avoid being caught in tiny niches, and how to prevent "genetic isolation" from setting in. We want to find a way to avoid this trap, without sacrificing the freedom to radically change hardware and software, a freedom that will be crucial in "bleeding-edge" research for years to come.

From the point of view of software development, the only viable solution to these problems is to facilitate code reuse both in time (from past to future) and space (between geographically dispersed people and institutions). For projects of a reasonable size this means following a *modular* approach, where software is ideally divided in independent components, that can be developed and maintained by different people so that efforts are shared among groups having distinct competences. A modular software platform is flexible. Obsolete modules are removed and substituted for newer ones without catastrophic effects. It is difficult to take advantage of code written by other people in different contexts unless that code avoids extraneous constraints and dependencies at all levels, from the hardware architecture to the development environment and programming language. In robotics, dependencies between modules need to be minimized also from the point of view of run-time performance; as long as resources are available the addition of new components should not clash

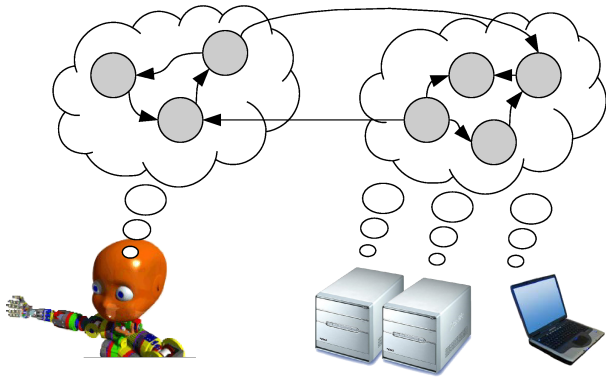

Figure 1. Our model of a humanoid robot's "brain". We assume a set of processors, some of which may be on the robot, some of which may not be. We draw no distinction; for research purposes, it makes sense to have off-board processing to do today what robots will be able to do on-board tomorrow. We assume diversity: different devices, operating systems, processors, different languages, libraries, etc. (of course, within our own project we have standards, but we don't expect everyone using our robot to agree). We exploit key Free Software tools for smoothing over differences in operating systems, build systems, and programming languages. We develop YARP, for smoothing over differences in networking, devices details, and libraries relevant to robotics. We release YARP as Free Software and use it to support our open robot platform, the ICub humanoid, whose design will be available under free and open licensing.

with the overall behavior of existing ones (in terms of throughput, latency, etc). And from the hardware development point of view, the robotic platform can be seen as another factor in the equation of code reuse. Common hardware, common protocols, electrical standards, sensors, etc. can certainly make the our life easier. As it does for software, modularity can play a role in the hardware design too.

In this paper we describe our efforts to build a modular humanoid robot platform (see Figure 1). We describe YARP [11], an open source library that we have developed to support software development on humanoid robotics. With YARP we try to facilitate code exchange between researchers, especially when this speeds up the time it takes to develop a platform and use it for research. We here report aspects of YARP that we hope will contribute to longevity and interoperability of software developed for robotics. Analogously for hardware, we describe our efforts to create an open robotic platform, the *ICub* that can be shared among several research groups worldwide.

Following the Open Source philosophy we make the code of our software and hardware available so that other researchers can better understand it and have the freedom to improve and better adapt it to their needs. We think it is relevant to any small research group, either academic or industrial, who wishes to develop novel robots (as opposed to build applications on third party robots). We want to maximize the reach of such research groups, being mindful of

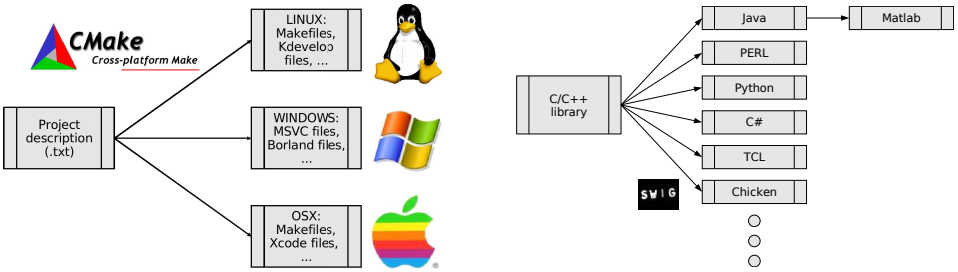

Figure 2. With the aid of a set of free and open source tools, a C/C++ based project like YARP can have a very wide reach. C/C++ source code is quite portable and widely supported, but the infrastructure needed to compile such programs varies a great deal. Tools like autoconf and automake have smoothed over the differences for UNIX-like systems. CMake (left) goes further and makes projects easy to compile within a wide range of integrated development environments (including UNIX-like systems, but also Microsoft Visual C++, Apple Xcode, Kdevelop, etc). For operating-system dependent functions, we use the free and open source ACE library [7]. SWIG (right) takes C/C++ source code and generates "wrappers" for it, usable from many different languages (including Matlab via Java).

the fundamental tension between providing a consolidated system and giving enough freedom to change every single part via upgrades and replacements.

2 A software ecology

Our initial motivation was that many robot projects are "black holes", in terms of software. A lot of software gets sucked in, but very little comes out. Once a piece of software has been adapted to a particular robot, it takes a lot of work to extricate it again and apply it to another. Obviously the answer to this problem is modularity. So there are now several architectures/middleware/frameworks for modular robot systems, YARP being one of them. The major concern for any such middleware (including YARP) should be that it not also become in turn a "black hole" – the danger is that once a piece of software has been adapted to a particular architecture/middleware/framework, it may take a lot of work to extricate it again and apply it to another. That would be a bit somewhat self-defeating. So modularity alone is not a solution to software reuse, since different organizing architectures, middleware, or frameworks may be mutually incompatible. It is important that modules developed can fit into a broader "ecology": the complicated, sometimes messy collection of niches world-wide in which software development occurs.

2.1 C/C++

We decided to use C++ as the main language for development. This is motivated by the fact that C++ is an object oriented language that is widely used by many developers in the world, and is well supported and portable on almost all the available platforms. Perhaps more importantly for robotics, C++ allows writing very efficient code and interfacing with the hardware at the lowest level. The drawback is that the compile process varies a lot depending on the platform and development environment. For example Linux and Cygwin developers use mostly Makefiles, whereas Microsoft Windows developers may prefer Visual Studio project files. Altough C++ has reached a fairly good level of portability which allows, with a reasonable effort, writing applications that compile on all platforms, it is still very common to have to wrestle to port code that was written for one platform onto another. On the other hand, following a modular approach, we would like our software to be as flexible as possible and be adaptable to the needs of users and the platform that they work on. In YARP, unavoidable dependencies have been made as localized as possible to modules that can be compiled or not depending on the underlying system and user choices. So for example applications that require a GUI get compiled only when the supporting libraries are installed on the system, and all the essential operations of YARP are independent of GUIs. We take similar care for dependencies on mathematical and image processing libraries.

Among the available tools for automatic configuraton of software packages, we decided to use CMake [18]. CMake is a cross-platform, open-source build system. It produces build files for the environment of choice (e.g. makefiles for Unix, Borland and MinGW and project files for all Microsoft compilers) starting from a language independent description. The language of CMake is powerful enough to support a flexible configuration process based on the packages that are available in the system and the preferences of the user (see Figure 2). Through CMake the build process of YARP is robust, simple and flexible. CMake is free and open-source, with a healthy community of developers. We use another free and open-source tool called SWIG to make YARP easy to use from many different languages. In all these choices, we are following the practices of large successful open-source projects.

2.2 Free Software

The ability to integrate software modules into a system depends not just on the technical constraints attached to their use, but also the cultural constraints (be they social, legal, or commercial) they carry. For example, whether two modules can be integrated can depend not just on their interfaces but also on

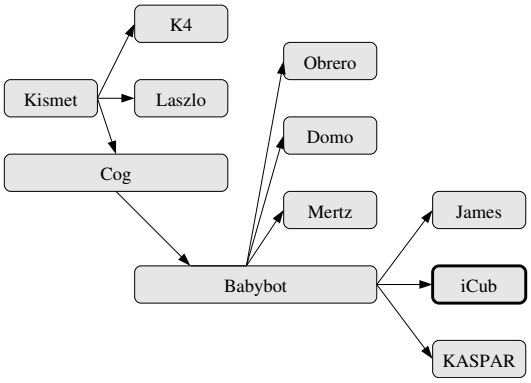

Figure 3. A potted history of YARP (for more details, see [11]). YARP was born on Kismet [2], grew up on Cog [3] and BabyBot [12], and serves as the software architecture for the iCub humanoid [23]. Along the way other humanoids have also used the system. With ICub, we are trying to create some hardware "genes" that can travel along too, so each robot does not need to be designed from scratch. There are currently 9 copies of the ICub head.

the conditions under which use of the modules is permitted by their respective creators, and what conditions the integrator wishes to apply to the aggregated system. This adds a great deal of complexity to the process of integration. In general, software produced under conditions where the creator has strong opinions about how it should be used, and enforces those opinions in licensing and other measures, does not make a good module to build on. It is possible, but painful.

The Free Software model is an alternative that strikes a different balance between creator and integrator. It proposes a set of standard freedoms which should be granted with software. Taken together these freedoms make the software actually useful as building blocks without excessive social/legal/commercial complexity. The freedoms are enforced using copyright law principles that apply to most of the world.

The Free Software model says nothing about the cost of software, although it does tend to contribute to commoditization, driving the cost of infrastructure-related software such as webservers and operating systems down. Free software should not be confused with "freeware". Freeware software is available without charge but may have complex social/legal/commercial terms attached, and may or may not grant the freedoms associated with free software (usually not).

The effectiveness of free and open software is becoming better understood from a business perspective [25]. The free and open model has had a crucial effect in the field of embedded devices, a large and growing market that overlaps with robotics, spurred by the existence of embedded Linux [6]. We release all our work under free and open licenses, in order to encourage their use as

building blocks. Historically, our "YARP" software grew and developed this way, principally through a collaboration between robotics groups at MIT and the University of Genoa 3.

2.3 Interoperating

The closest project in spirit to YARP is that of the Player project [24], and we take it as an example here of how YARP can interoperate with other architectures/frameworks. The Player/Stage software collection is widely used in the field of mobile robotics, and is the nucleus of a healthy, pragmatic community of developers. Rudimentary interoperability is possible between these projects at the device level. Player contains a "yarpimage" driver which can accept images from a YARP Network. A "stage" driver has been developed for YARP, which gives access to the 2D robot simulator of that name from a YARP Network. In fact devices have some similarity in structure between YARP and Player, but have the crucial difference (at least to our eyes) that YARP starts with just a thin C++ wrapper which permits direct function calls while Player devices must operate through a message passing framework (although that message passing can now be internal rather than across a network). In principle, YARP devices should be easy to wrap up systematically and efficiently for Player (since no assumptions are made about the communication model), but the other direction is not so easy.

The driver mechanism in both projects gives a very straightforward way to integrate quickly with what could otherwise be incompatible middleware. At a higher level of potential interoperation, both projects are free and open source, they both have documented network protocols, both have made an effort to allow different transports, and both are reasonably portable. So a determined individual will probably be able to make them work together on any given task. Neither YARP nor Player is aimed at users unwilling or unable to program. Such users benefit from these projects indirectly, through for example the ICub platform, which is being built using YARP to support users from different disciplines including neuroscience and experimental psychology.

3 Devices and Drivers

Code reuse becomes difficult at the level where algorithms communicate with the low-level hardware. The OS layer of YARP tries to minimize dependencies between algorithms and the hardware for which we define a constant interface (threading, memory, network, filesystem). Unfortunately more specific hardware (motor control boards and frame grabbers are popular examples) requires

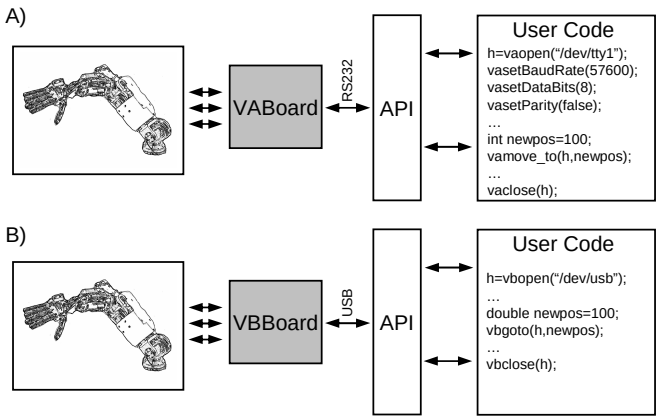

Figure 4. Example of code dependency. A) VABoard is a motor control board which interface to the robot through serial port. The user's code contains code to initialize the board and control the robot through the API library provided by the vendor. B) A new motor control board is connected to the robot; this new device has a USB interface and a different API. The differences are propagated to the user's code which must be rewritten.

a more sophisticated mechanism. In these cases vendors provide device drivers and a set of APIs to the user. The API comes in the form of a static or dynamic library which is linked to the user's code. Unfortunately APIs vary a lot even within devices that belong to the same family. Even worse the API of the same hardware may vary on different operating systems or change on future releases of the hardware. User code becomes dependent on the particular board for which it was initially developed and bound to the decisions and assumptions of the vendor. For example vendor A might decide to use integers to represent the position of a motor joint, wheras vendor B might decide to use a floating point variable. Otherwise interchangeable devices may have different "initialization" procedures. Consider for example a motor control board which has a serial interface to the host computer; the API of this board will probably require that some parameters (port number, baud rate, number of data bits, etc) are specified when the device is created. Suppose now that we obtain a more recent release of the same board that now has a USB interface. In this case the parameters to initialize the board are different and we are forced to rewrite all processes that use it (the situation is represented in Figure 4).

We call devices which can only be accessed using vendor supplied material "sticky devices" because they tend to make the particular set of assumptions chosen by the vendor stick to the user's code. A logical step in such a situation is to wrap the functionality supplied by the vendor in a facade, so that source code dependencies are reduced. In YARP wrappers can be made individually, compiled and built separately, and optionally used across the network. This mechanism produces a level of separation between device-specific code and user code that is effective for "quarantining" the sticky devices. This is achieved

in three ways: (i) definition of interfaces for families of devices (ii) localization and separation of device initialization and creation (iii) creation of network wrappers and separation between devices and communication.

Note that when we talk about "interfaces" here we do not refer to the interface description languages used in CORBA and other systems, but simply to a consistent API in C++. Concerns related to communication are addressed in point (iii), not (i). We keep communication and device interfaces separate, so that users can exploit one and not the other as they wish, and also code written to use a device remotely can later be made local with only a cost of a single extra virtual method call compared to calling the vendor's API directly. This is important so that users don't need to go through a painful porting process if they discover at some point that remote operation is too slow for their application – for example, an implementation of the vestibular-ocular reflex might require a very tight loop between sensors and motors.

3.1 Device Interfaces

An interface to a YARP device is the specification of the functionalities it provides. In practice in C++ an interface is a virtual base class, whose member functions define the ensemble of functionalities a device must implement in order to provide that interface. A YARP device is a "wrapper" class which implements all methods declared in its interface. A single device can of course expose more than a single interface (in C++ this is implemented through multiple inheritance). All details specific to the hardware (vendor's API and library) are handled in the wrapper class and are hidden behind its interfaces. The idea is that changes in the hardware are caught by the wrapper class and never propagated to the user code. As a result, if interfaces are well designed, the impact on the code due to hardware change is minimized. Of course, unique features of a device can be exposed in a new interface, but without much benefit over using the vendor's code directly for that specific feature. And any code written using that novel interface will need to be reworked if another device is substituted.

As discussed previously, initialization parameters may introduce annoying dependencies in the user's code. To solve this we have defined a common interface to all devices (the *DeviceDriver* interface) which normalizes how devices are initialized and un-initialized, and, more importantly, how initialization parameters are passed to them. In particular this interface defines two methods:

```
virtual bool open(yarp::os::Searchable& config)=0;
virtual bool close()=0;
```

This *open* method initializes the device. Initialization parameters are passed

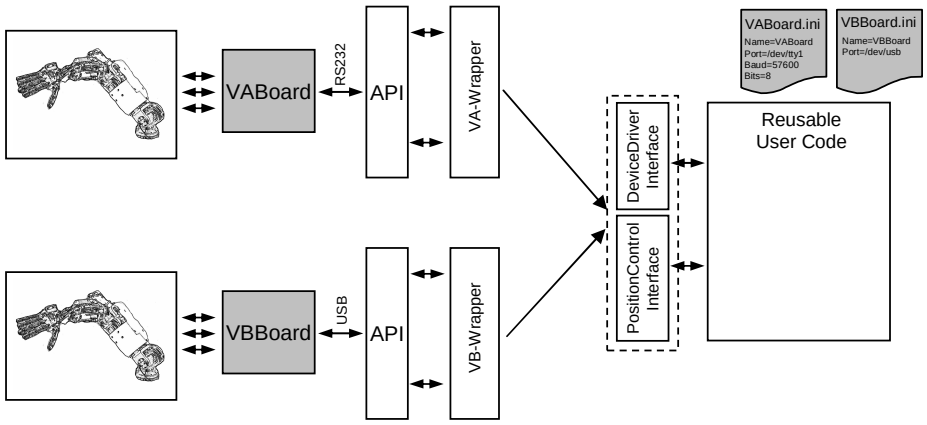

Figure 5. Interfaces allow code reuse. VABoard and VBBoard (see Figure (4)) now implement the same interfaces (through their respective wrapper classes). The user's code accesses the hardware through these interfaces and is not aware of the details of how the methods are actually implemented. The different initialization parameters are listed in configuration files and are thus separated. VABoard and VBBoard are now completely interchangable.

to the function as a (typically nested) list of key-value entries represented as a *Searchable* object. A *Searchable* can contain all possible parameters that devices might require for initialization. Initialization parameters for devices are stored in ".ini" files (again in the form of a list of key-value entries). A process that wants to open a device reads the file and transfers its content into a *Searchable* object. This class plays a role in YARP similar to that of the *ConfigFile* class in Player/Stage, except generalized to work for parameters expressed as command line arguments, or passed across the network, or created in a GUI, etc. – we abstract across all the possible sources of configuration settings. The configuration object is passed to the device through the *open* function. It is worth stressing that up to now this procedure is totally device independent, because the parameters are just copied and not interpreted by the process. It is only in the implementation of the *open* method (in the wrapper class of the device) where the *Searchable* object is parsed to extract the parameters that will be used to inizialize the device. The *Searchable* object is designed so that it can collect information about how it is used, yielding some basic documentation about the parameters relevant to a given device.

The *close* method performs all the operations required to shut down the device properly and release all the resources it was using. No parameters are required by this function.

YARP defines interfaces to broad families of devices. For example this is a partial list of the interfaces defined for generic devices that generate a stream of color images (frame grabbers):

- *IFrameGrabberRgb*, methods in this interface provide access to the most recent frame acquired by the device, and information about its size (number of columns and rows);
- *IFrameGrabberControls*, specifies a set of functionalities to control how the device performs the acquisition, like shutter speed, brightness and gain.

Interfaces to motor control devices are more difficult to define. Control boards designed for industrial applications have often a quite standard interface which provides a PID control algorithm and position or velocity control modes. Things become more complicated when we consider also programmable devices that can implement virtually an infinite set of functionalities and control algorithms. For this reason interfaces to control boards have been defined on the basis of the control paradigm they implement. Accordingly, YARP defines:

- *IEncoder*: group all methods providing access to the motor encoders, like methods for reading the current position and velocity of each axis;
- *IPositionControl*: methods to control each axis by specifying its position;
- *IVelocityControl*: methods to control each axis by specifying its velocity;
- *ITorqueControl*: methods to control the amount of force/torque exerted by each axis.

These last interfaces are independent of the particular algorithm the control board implements to realize the corresponding functionality. These details are delegated to specific interfaces. For example *IPidControl* includes methods to interface to a PID controller, such as for example to read or set the values of the gains.

To summarize, interfaces captures similarities among devices and allows separating device dependent code from user code. To the extent that user code uses interfaces shared by other devices, another device can be substituted later without change to that part. This includes devices with different initialization procedures, or different APIs (see Figure 5). Devices can also be nested or assembled into composite structures if necessary.

3.2 A factory of devices

Encouraging device access through interfaces achieves a good level of separation between vendor/device specific APIs and user level code. Interfaces alone, however, do not guarantee a complete level of separation. In practice users must still specify the type of device they want to create. Care must be taken to avoid this introducing unwanted coupling between device specific code and user code. A common software engineering practice is to *localize object creation* so to minimize the amount of code that is responsible for object creation and initialization. We have seen that in YARP part of this is realized

by the *DeviceDriver* interface, which encourages all initialization procedures to be performed inside a standard *open* method. We then go one step further, and encourage device creation to be delegated to a *factory*. The *factory* contains a list of all devices available in YARP and the corresponding functions to call to create them. It receives a list of initialization parameters, creates the device, and initializes it through the *DeviceDriver* interface (this is similar to the *DeviceTable* in Player). If the process is successful a valid pointer to the device is returned. This pointer is the only "access point" to the device and (via dynamic casts) its interfaces.

The whole process of creation, initialization and interface access is managed by the *PolyDriver* object. Under good "portable" usage in YARP, the user accesses devices via *PolyDriver*, asking for their creation through the *PolyDriver::open())* method. This works just the same as the *open()* method we talked about for specific devices, except now our configation object (read from file, command line, network, GUI, etc.) specifies which device we want as well as all its options. If the driver is successfully created the *factory* returns a valid pointer which is stored inside the *PolyDriver*. The lifecycle of the device is managed by the *PolyDriver*, and interfaces to the device are acquired via the *PolyDriver::view()* method (see Figure 6).

3.3 An example: accessing a motor control board

For example suppose we want to use the `test_motor` device. In YARP this is a fake device which simulates a control board for testing purposes. This device supports the *IPositionControl* and *IVelocityControl* interfaces. To begin with, we first create an instance of the *PolyDriver*. The actual device is create by calling the *PolyDriver::open()* method specifying the symbolic name of the device (`test_motor`):

```
PolyDriver device;
device.open("test_motor");
```

We could also be more specific and pass in configuration options, but we keep things simple here. Now we can get the interfaces we want from our `test_motor` by calling the *PolyDriver::view()* method:

```
IPositionControl *ipos=NULL;
device.view(ipos);

IVelocityControl *ivel=NULL;
device.view(ivel);
```

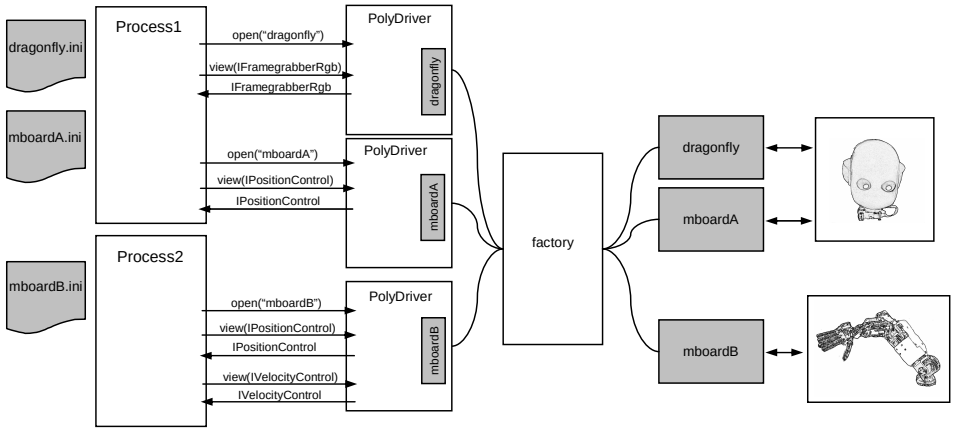

Figure 6. Device creation and initialization. Creation of devices in YARP is delegated to a *factory* object. Users access devices through instances of the *PolyDriver*. The Figure describes the following situation. Process1 controls a robotic head and needs to access the robot frame grabber (whose symbolic name is "dragonfly") and to the control board connected to the motors of the head ("mboardA"). Process1 creates an instance of *PolyDriver* and opens the device; the symbolic name of the device is passed as a parameter of the *open* function, together with initialization parameters read from a .ini file. The *PolyDriver* hands over these parameters to the *factory* which creates an instance of the device and returns it to the *PolyDriver*. Subsequent calls to the driver are entirely handled by the *PolyDriver* itself. Process1 calls *view* to acquire the appropriate interfaces to the device. A similar procedure is performed by the same process or other processes (Process 2 in Figure) to create instances of different devices. The important point is that calls to the YARP API for each specific device from the processes can be just one level of indirection away from vendor-supplied code.

Checking if `ivel` and `ipos` are non-NULL assures us that `test_motor` really supports the respective interfaces. This is reminiscent of the treatment of interfaces in something like DCOM, but is grossly simplified, implemented directly with the C++ type system so that we can be just one virtual call away from the native device API. We can now call methods of the *IPositionControl* interface to (for example) move joint number 0 to the angular position of $40deg$, with the velocity of $5\frac{deg}{s}$ and acceleration of $100\frac{deg}{s^2}$:

```
ipos->setRefAcceleration(0, 100);
ipos->setRefSpeed(0, 5)
ipos->positionMove(0, 40);
```

(The use of degrees rather than radians in YARP interfaces is part of a bias towards keeping all data as human-readable as possible.) Or we could use the *IVelocityControl* interface to move axis 1 at a smooth velocity of $5\frac{deg}{s}$, accelerating to that velocity at $100\frac{deg}{s^2}$:

```
ivel->setRefAcceleration(1,100);
```

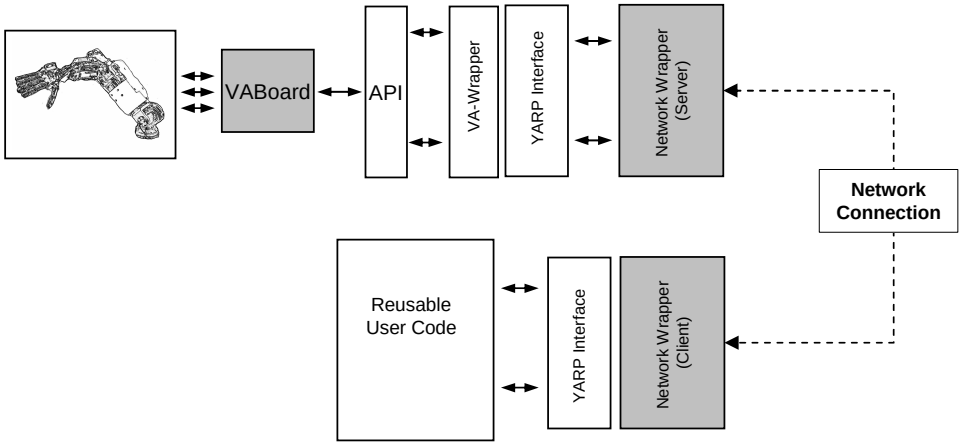

Figure 7. Network wrappers allow device remotization. A generic *Server Network Wrapper* exports the YARP interface of VABoard so that it can be accessed remotely by another machine. At the other side of the communication link the *Client Network Wrapper* exports the same interface of the remote device so that it can be transparently accessed by the client code. The local device and the *Client Network Wrapper* are totally interchangable, the only difference between the two is in term of performance (the time it takes to execute a function) and initial configuration.

```
ivel->velocityMove(1, 5);
```

The same code would do something for all of the motor devices we have that implement these interfaces. The details might vary a bit but in principle the overall behavior should be consistent.

3.4 Device Remotization: Network Interfaces

A final level of separation is achieved by supporting device remotization, or operation across the network via proxies. This feature is desirable for many reasons. It allows separate compilation and execution of different parts of the system, to avoid (for example) the existence of motor control libraries on just a single OS constraining you to also do image processing on that same OS. It makes distributed processing easier, letting you shift processing to extra machines when the load becomes high. Remotization is in practice one of the major benefits of offered by YARP and Player. They achieve that goal is somewhat different ways. In Player, devices are responsible for producing their own messages (represented as a C structure, with the Player library responsible for marshalling/de-marshalling). In YARP, message production is done at the level of device *families*. The use of standard YARP APIs for families of devices makes it straightforward to substitute in a proxy instead

of a local device. By using YARP ports for communication, with their defined protocols, remotization also gives us portability across different platforms, as it naturally defines a network interface that can be used to make resources available on one platform to processes compiled and running on a different one. This decouple the compilation, build environment, libraries, operating system and language dependencies of hardware and user software.

The remotization mechanism relies on the communication layer (see Section 4) and on two *Network Wrapper* devices, one acting as a *Server* and the other acting as a *Client*. Both network devices implement the very same interface of the device they wrap: the only difference is that they do not connect directly to the hardware but act as network proxies, talking to each other using a predefined protocol, which involves one or more YARP Ports configured for RPC and/or streaming as the nature of the device dictates (see Figure 7).

A process that wishes to connect to the remote device using the YARP code-base creates an instance of the *Client Network Wrapper* (the YARP code-base could be avoided by working with the network protocol directly, as described in Section 4). This wrapper exports exactly the same interface of the "wrapped" device so the process can pretent that it is connected to a real device. The *Client Network Wrapper* converts calls from the process into messages, sends them to the other end of the communication link, and, in case a reply is expected, waits for data and dispatches it to the calling process. The *Server Network Wrapper* waits for incoming connections from the network. In addition it creates an instance of the wrapped device to which it forwards requests from the network. If requests involve a reply theses are sent back to the calling port so that they are received by the remote client. The *Server Network Wrapper* gains access to the local device through its interface; as such it is a total independent entity that can be reused for devices of the same family.

4 Transporting data

A very basic problem that keeps cropping up in robotics projects is simply how to move data around between sensors, processors, and actuators. There's a universe of "middleware" solutions in existence for communication (see the survey in [10] and the related-work review in [4]). Our own preferred solution in YARP has the following features:

- ▷ We use an abstract model of communication that is transport-neutral and peer-to-peer.
- ▷ The underlying transport used for each individual connection between peers can be selected independently. Choices such as network versus shared memory, tcp versus udp, unicast versus multicast, text versus binary, which of

several networks to transmit on, etc can be made on a case by case basis. We encourage such details to be external configuration choices rather than properties embedded in programs.

- ▷ We are careful to have one text-mode transport that is extremely easy to implement, for those who wish to interact with a YARP system without using any of the YARP libraries or executables. We believe this is very important for supporting interoperability, and providing a gentle slope to integrating YARP into an existing system or vice versa.
- ▷ The model of communication is not intertwined with our ideas about how devices work or how processes should be started/stopped. Thus users can "cherry-pick" the parts that work for them.

Communication in YARP generally follows the *Observer* design pattern. Special port [1] objects deliver messages to any number of observers (other ports), in any number of processes, distributed across any number of machines, using any of several underlying communication protocols.

4.1 The YARP Network

For the purposes of YARP, communication takes place through "connections" between named entities called "ports". These form a directed graph, the "YARP Network", where ports are the nodes, and connections are the edges. Each port is assigned a unique name, such as "/icub/camera/left". Every port is registered by name with a "name server". The goal is to ensure that if you know the name of a port, that is all you need in order to be able to communicate with it from any machine. The YARP name server (YNS) is a generalization of DNS name service on the public internet for converting from domain names to IP addresses. It is not concerned just with machines but all the details necessary to make a connection with a specific resource. The YARP name server is designed to be easily used by clients who are not themselves using the YARP libraries or executables.

The purpose of ports is to move data from one thread to another (or several others) across process and machine boundaries. The flow of data can be manipulated and monitored externally (e.g. from the command-line) at run-time. It can also be accessed without using the YARP libraries or executables, since the relevant protocols are documented. If messages follow YARP guidelines, then they can be automatically converted to and from a "text mode" connection, enabling human monitoring and intervention in the system, and providing an easy way to experiment with integration with non-YARP modules.

[1] Don't confuse YARP ports with TCP/IP socket port numbers. We use the word "port" to refer to the former and "port number" to refer to the latter.

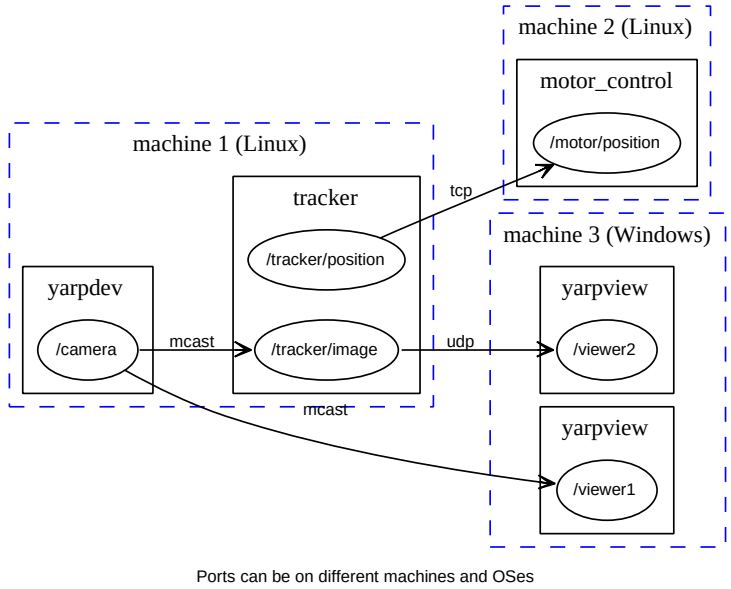

Figure 8. Example of a network of ports. Images are transmitted from a camera ("/camera") port to a viewer ("/viewer1") port and the input of a visual tracker ("/tracker/image"). The tracker annotates the image, for example by placing a marker on a tracked point, and transmits that to another viewer ("/viewer2"). The tracker also sends just the tracked position from a position output port ("/tracker/position") to a input controlling head position ("/motor/position"). Every port belongs to a process. They do not need to belong to the same process or be on the same machine as each other. Every individual connection can take place using a different protocol or physical network – in the figure multicast, udp, and tcp are shown.

A port can send data to any number of other ports. A port can receive data from any number of other ports. Connections between ports can be freely added or removed, and may use different underlying transports. The use of several different transports and protocols allows us to exploit their best characteristics. TCP is reliable, it can be used to guarantee the reception of a message. UDP can be faster than TCP, but without guarantees. Multicast is efficient for distributing the same information to large numbers of targets. Shared memory can be employed for local connections. Text-mode operation is much more human-friendly, and a good place to get started with external integration. Figure 8 shows a very simple network of ports for a visual tracking application.

Connections between ports in YARP can carry replies if desired (and if the underlying protocol supports that), so conventional "RPC" (remote procedure call) style synchronous operation is possible. We encourage streaming rather than RPC whenever possible, because RPC can make a network brittle by introducing strong coupling of timing between processes. For our implementation of ports, we have broken them down into several logically separable

parts:

- ▷ The carrier factory (called "Carriers"); carrier is our generic name for any different transport or protocol that can carry a connection. This factory maintains a list of managers for different kinds of connection. The user can extend this list with their own custom types of connection (for example, for a kind of network we've never considered, or a different implementation of an existing carrier).
- ▷ The core communications module (called "Port"). This will manage connect requests, disconnects, reading, writing, and various administrative details. It defers to the carrier factory to create specific connections and knows very little about their nature.
- ▷ Reader and writer buffers (called "PortReaderBuffer" and "PortWriterBuffer"). In order for communication to be efficient and avoid unnecessary copies, objects being transmitted generally need to be left untouched until communication is complete. With the variety of possible connections and options possible, the details of this can become complicated. YARP implements a certain set of policies we think are good in the reader/writer buffer classes. These are wrapped around the Port class to provide a BufferedPort class that gives both a simple interface and efficient implementation, while keeping buffering and communication separable for those with strong opinions about how one or the other should be done.
- ▷ The YARP network interface (called "Network"). Provides methods for manipulating parts of the network, such as creating or removing connections between ports.

The YARP name server is a simple program using a single ordinary port as its input; in the past, it had its own special protocol but now it is just like any other YARP program. This is possible because ports can operate without access to a name server if desired; it is another separable component.

4.2 Human readable/writable communication

There is a constant tension between using binary formats and human-readable formats. Binary formats can be much more efficient, but text mode formats can be easier to work with and study experimentally. The value of text formats and protocols has been seen time and time again in the short history of computing (postscript, http, html, xml, etc.). YARP is constructed so that both binary and text mode operation is possible.

The YARP communications system is written in two parts. The first part is a set of "carriers" which do the work of providing connections between ports, so that data can be faithfully transmitted from a source to a destination byte-for-

byte. The second (separable) part of the communication system is a standard data format. This standard is specified independently from the carriers, so that the carriers could be reused by someone with different opinions about data representation, but helper functions and classes make it easy to meet. This format is called the "bottle" format for historical reasons[2]. The bottle representation is based on a nested structure of certain familiar primitive types – lists, integers, floating point numbers, strings, binary blobs, and a special "vocab" type that is basically an integer in binary mode (for fast dispatching) and a string in text mode (for easy reading and writing). The important point is that binary and text representations are interchangeable. Under normal operation, ports can be sending easy-to-parse binary messages to each other, but then when a human eavesdrops on that data or tries to insert a message, they can still understand and generate the messages in text mode. Bottle-style messages can be expressed in several interchangeable representations: binary, text, command-line options, configuration files etc. We find that under various conditions sometimes we want the same kind of data coming from file, command line options, or across the network, so is convenient to have all the various represention mapping to a homogeneous structure.

In principle, evolution of communication protocols in YARP can be relatively painless. Since new "carriers" can be added freely, new and old versions could live side by side for a release or two. Ideally, something like today's text mode format should be honored for a long time, as a connection protocol of last resort.

4.3 Connection protocol

The connection protocol is the protocol used for a single connection from an output port to an input port. It has two main phases, the handshake phase, and the message phase. We begin once the sender has successfully opened a bidirectional streaming connection of some kind (presumably a tcp connection) to the receiver. First comes the **handshake phase**:

- **Transmission of protocol specifier**: Sender transmits 8 bytes that identify the "carrier" that will be used for the connection. The carrier can require switching to some other form of stream, or using a particular strategy for encoding data. The transmission of the initial 8 bytes is the only part of this protocol that is defined in terms of bytes sent.

[2] From YARP's online documentation: *The name of this class comes from the idea of throwing a "message in a bottle" into the network and hoping it will eventually wash ashore somewhere else. In the very early days of YARP, that is what communication felt like.*

- ▷ **Transmission of sender name**: Sender transmits the name of the port it is associated with. How this name is encoded and sent is the concern of the specific carrier.
- ▷ **Transmission of extra header material**: Sender and receiver may engage in further communication as needed for the specific carrier.

At the end of this phase, the sender and receiver are both "aware" of which carrier is in use (this may have involved discarding the original stream and switching to a new one – e.g. mcast, udp, shared memory) and both are aware of the identifier of the port at the other end of the connection. This can be important for collaboration during connection shutdown.

Next comes the **message phase**. After the handshake, the connection is (as far as YARP is concerned) quiescent until either the sender decides to send a message across it. It is technically possible for the receiver to initiate activity – we'll return to this issue.

- ▷ **Transmission of index**: Carrier-dependent. Some carriers will require statistics about the message (such as its length) to be given at the beginning. YARP binary carriers have a fairly elaborate index, due historically to a limitation of the QNX message-passing API. Text-mode carriers have no index at all, since it is unreasonable to expect a human to be able to generate one.
- ▷ **Transmission of payload**: The actual message is transmitted in a carrier-dependent way.
- ▷ **Acknowledgement of payload**: The receiver may acknowledge transmission in some way. Carrier-dependent.

The message phase repeats as often as the user wants. Note that the description so far is very loose – just about every aspect of a connection is carrier-dependent. What does YARP actually expect of connections, in order to build on them?

- ▷ After the handshaking phase, both sides of a connection know the names of the other side. This is important for housekeeping.
- ▷ After the handshaking phase, a connection endpoint must have certain knowledge about the connection that it can report to YARP. It will be connection-based or connectionless. It will be text mode or binary. It will deliver acknowledgements or not. It will be able to deliver replies or not. It will be active or "fake" (in multicast, many logical connections can be serviced with a single active connection – these details are taken care of at the carrier level).

The important point about the communication protocol is that is *polymorphic* and allows *heterogeneous* use – the protocol on each connection between two ports can be controlled independently. This allows for system evolution, where

8-byte magic number	protocol
‘Y’ ‘A’ 0xE4 0x1E 0 0 ‘R’ ‘P’	tcp
‘Y’ ‘A’ 0x61 0x1E 0 0 ‘R’ ‘P’	udp
‘Y’ ‘A’ 0x62 0x1E 0 0 ‘R’ ‘P’	multicast
‘Y’ ‘A’ 0x63 0x1E 0 0 ‘R’ ‘P’	shared memory
‘C’ ‘O’ ‘N’ ‘N’ ‘E’ ‘C’ ‘T’ ‘␣’	text
‘C’ ‘O’ ‘N’ ‘N’ ‘A’ ‘C’ ‘K’ ‘␣’	text-with-ack
‘G’ ‘E’ ‘T’ ‘␣’ ‘/’ . . .	http
. . .	. . .

Table 1
Partial list of YARP connection “magic numbers” – the eight initial bytes of a connection in YARP specify the desired protocol to use from that point on. Magic numbers are commonly used in all sorts of file formats intended for interchange. YARP’s rather strange magic numbers for binary formats (“YAnnnnRP”) evolved in order to be compatible with earlier versions of itself.

new protocols are introduced, potentially mapping onto radically different physical networks, virtual networks, or external middleware.

4.4 YARP without YARP

Suppose some YARP programs are running and we want to send or receive data from them. For example, suppose there is a YARP port called “/motors” which will accept commands to move a motor. For concreteness, let’s imagine we have started the following standard YARP programs (on the same or different machines):

```
yarp server
yarpdev --device test_motor --axes 2 --single_port --name /motors
```

The “yarpdev” program here creates a port called “/motors” that can accept command to a fake set of motors (2 axes or degrees of freedom), and report on their state. Normally we would interact with the motor through a device API that takes care of communication details. But if for some reason we can’t use the YARP codebase, what can we do?

YARP ports listen to incoming connections of a certain default initial carrier (tcp), always ready to make new connections for input or output. Suppose we can discover that “/motors” is listening on port number 10022 of our current machine (we could discover that using netstat on Linux, or by querying the YARP server as we’ll see shortly). We can then connect manually to the port

as follows:

user types	**system responds**
$ telnet 127.0.0.1 10022	*(telnet startup message)*
CONNECT foo	Welcome foo
help	*(an explanation of available commands)*
*	This is /demo at tcp://127.0.0.1:10032 ...

Everything so far would be basically the same for any YARP port. For people who have used MUDs, IRC, or serial interfaces to hardware, it should all seem vaguely familiar. Of course we don't suggest actually using telnet, it is just a placeholder for socket communications in the user's language of choice. So far all our communications have been "administrative" – we have communicated with the port but not really with the program that owns it. To do that, we send payload data. For the text-mode carrier we've chosen (determined by the 8 initial bytes we sent, "CONNECT␣" in this case), this is done by typing "d", hitting return, then writing a text-mode representation of the data we want to send. Let's try it:

user types	**system responds**
d	*(no response, waiting for data)*
help	*(a list of available yarpdev commands)*
d	*(no response, waiting for data)*
[get] [axes]	[is] [axes] 2 [ok]
d	*(no response, waiting for data)*
[set] [pos] 0 100.0	[ok]

We have determined that there are indeed two axes available as we requested when starting yarpdev, and have set the position target for the first axis to 100.0 units. We could go on to query positions, use other interfaces, etc. We can disconnect by closing our connection (or, more politely, sending the message "q").

By default, the motor port will stream encoder readings from the motors to any reader that connects. To subscribe to this stream, we simply connect as above and then type "r" to reverse the connection. Reversing means to invert which side should take the initiative in sending data.

Suppose we wanted to send messages more efficiently? We start out the same way, connecting via TCP, and then give the "magic number" of the carrier we

want to use (tcp binary, udp, mcast, shmem, etc). Understanding these carriers is a bit harder than basic text-mode operation, but they are documented.

One part we skipped at the start was how to discover how to access ports in ths first place. If we know the port we want is called "/motors", how do we discover where it is? We can in fact talk to the yarp name server using exactly the same protocol that we have described here. What socket port the name server listens to is reported when it starts (and can be configured, or discovered using a broadcast protocol).

So, with a running YARP system, we can discover and communicate with running programs, sending commands and reading data, without using any YARP libraries or executables. All the steps we've gone through are trivial in any language with a basic socket library (we're not using any special features of telnet, it is just for demonstration purposes). It is important to remember, though, that while we've been communicating with the "/motors" port using text across tcp, at the same time the same port could be communicating with other programs via binary messages over udp or multicast etc. We believe the existence of the bottle format for communicating with YARP processes makes it much easier to experiment with and build bridges to in text-mode (like http for the web), while gracefully supporting switching to binary-mode communication when the situation demands it.

5 RobotCub and ICub

RobotCub is a collaborative project funded by the European Commission under the Framework 6 program and it is part of the Cognitive Systems effort coordinated by the Unit E5 [27]. One of the goals of RobotCub is that of creating an open platform where many other projects could thrive by exploiting a common hardware and software infrastructure. RobotCub has also the goal of making the *ICub* (this is the name of the robot) the platform of choice for several other research groups worldwide and, simultaneously, to advance our knowledge of natural and artificial cognitive systems.

One of the tenets of the RobotCub stance on cognition is that manipulation plays a key role in the development of cognitive capability. Consequently, the design is aimed at maximizing the number of degrees of freedom of the upper part of the body (head, torso, arms, and hands). The lower body (legs) is made to support crawling on the four limbs and sitting on the ground in a stable position with smooth autonomous transition from crawling to sitting. This allows exploration of the environment, grasping and manipulation of objects lying on the floor. The total height is estimated to be around 105cm. The total number of degrees of freedom (DOF) is 53 of which 41 in the upper body (7

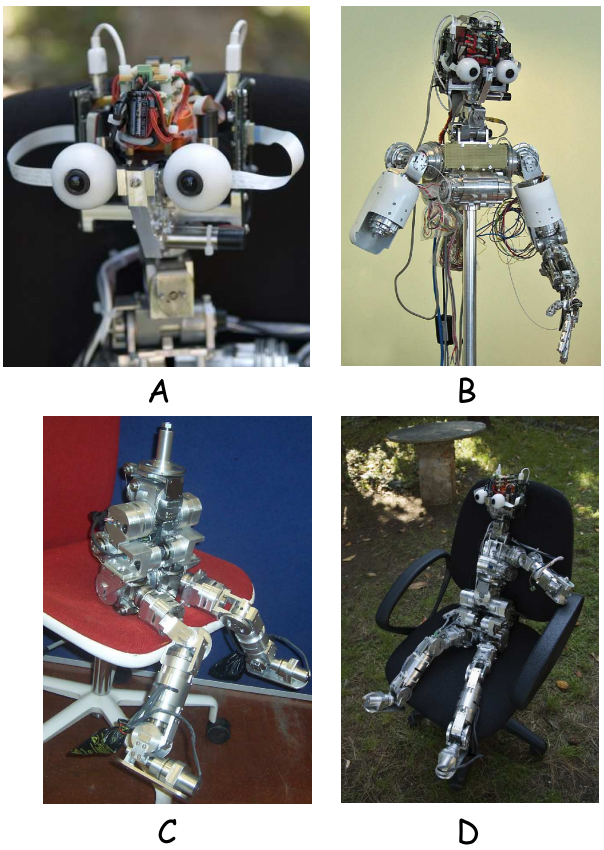

Figure 9. The ICub at various stages of construction. Panel A) shows the head of the robot with part of the embedded electronics and sensors; in B) the upper torso showing also the left hand; C) shows a first realization of the legs (now improved); D) a tentative assembly of the entire robot.

for each arm, 9 for each hand, 6 for the head and 3 for the torso and spine). Each leg consists of six additional degrees of freedom. The sensory system will include binocular vision and haptic, cutaneous, aural, and vestibular sensors. Functionally, the system should be able to coordinate the movement of the eyes and hands, grasp and manipulate lightweight objects of reasonable size and appearance, crawl using its arms and legs, and sit up. This allows the system to explore and interact with the environment not only by manipulating objects but also through locomotion.

The philosophy adopted by RobotCub is that of the free software movement, as codified by the General Public License (GPL). On the software side, the RobotCub project adopted YARP and contributed to the development of some new specific features. For the hardware, we selected the GPL license for the sources and FDL for documentation and drawings. While it is clear how to apply these licensing schemes to source code (e.g. C++), we need to clarify how to apply them also to hardware designs.

The phrase "Open Source hardware" might sound strange, but in fact it is a plain transfer of the open source philosophy to the entire design of the RobotCub platform. The design of the robot started from the preparation of specifications (e.g. estimation of torque, speed, etc.), a typical 3D CAD modeling, and eventually in the preparation of the executive files which are used to fabricate parts and for assembly. Without good documentation it is very complicated to build and assemble a full robot. This means that documentation (as for software) is particularly important.

The CAD files, in some sense, can be seen as the source code, since they are the "preferred form of the work for making modifications to it", in the language of the GPL. They get "compiled" into 2D drawings which represent the executive drawings that can be used by any professional and reasonably well-equipped machine shop either to program CNC machines or to manually prepare the mechanical parts. This compilation process is not fully automated and requires substantial human intervention. There is a clear dependency of the 2D drawings on the original 3D CAD model. To enable the same type of virtuous development cycle as occurs in open source software, the 3D CAD is required, since changes happen in 3D first and get propagated to 2D later. In addition, assembly diagrams, part lists, and all the material produced during the design stage should be included to guarantee that the same information is available to new developers.

One difference between software and the hardware design is that there are currently no effective formats for interchange of 3D models. Proprietary systems such as SolidWorks and Pro/E can import and export a range of formats, but going from one to another is lossy, destroying information needed for production and leaving just the basic geometrical shape. So in practice, designs are tied to tools produced by a particular vendor, and interoperability between hardware design tools is limited. In RobotCub we were forced to choose a specific set of tools for mechanical and electronic CAD and future upgrades will have to strictly adhere to these standards. Due to the absence of open source professional design tools, RobotCub uses proprietary products. This is an unfortunate situation, but there is no practical alternative at the moment. The "C++" and "gcc" of CAD do not yet exist.

As a practical matter, the simple duplication of RobotCub parts does not require the use of any of these tools since we provide all executive drawings and production files (e.g. Gerber files for the PCBs). For modification, the design tools are somewhat expensive (educational discounts or educational releases exist). Free of charge *viewers* are currently available for all file types in question.

For RobotCub, we decided to license all the CAD sources under the GPL which seems appropriate given their nature. Associated documentation will be licensed under the FDL. These will be made available through the usual source code distribution channels (e.g. repositories, websites).

5.2 The design process

The design process of RobotCub has been a distributed effort as for many open source projects. Various groups developed various subcomponents and contributed in different ways to the design of the robot including mechanics, electronics, sensors, etc. In particular, a whole design cycle was carried out for the subparts (e.g. head, hand, legs) and prototypes built and debugged. The final CAD and 2D drawings were discussed and then moved to the integration stage. Clearly, communication was crucial at the initial design stage to guarantee a uniform design and a global optimization.

The distributed design broke down at the integration stage where the industrial partner [3] stepped in to carry out integration, verification and consistency checks. The design and fabrication of the control electronics was also subcontracted to a specialized company. It is important to stress the collaboration with industry for a project of this size and with these goals and requirements. For many reasons building a complete platform involves techniques and management that is better executed by applying industrial standards. One example that applies to RobotCub is the standardization of the documentation.

A further strategy used in RobotCub is that of building early. Each subsystem was built as soon as possible and copied also as soon as possible. In several cases debugging happened because the copies of the robot did not work as expected or easy to fix problems were spotted. Sometimes the documentation had to be improved. Unfortunately, this strategy was applied less extensively to some of the subparts which are or were still under design and debugging.

The design stage will be completed by the realization of ten copies of the ICub. This will further test the documentation and in general the reliability of the overall platform including software, debugging tools, electronics, etc. The first release of the ICub will be consolidated after this final fabrication stage.

The actual design of the robot had to incorporate manipulation by providing sophisticated hands, a flexible oculomotor system, and a reasonable bimanual workspace. On top of this, the robot has to support global body movements such as crawling, sitting, etc. These many constraints were considered in preparing the specifications of the robot and later on during the whole

[3] Telerobot Srl, Genoa

design process.

The behaviors we set forward for representing the robot's skills generate two types of constraints:

- kinematics: about the geometrical construction of the robot;
- dynamics: about the forces and torques we require from the robot.

The possibility of achieving certain tasks is favored by a suitable kinematics, and in particular this translates into the determination of the range of movement and the number of controllable joints (where clearly replicating the human body in detail is impossible with current technology). Kinematics is also influenced by the overall size of the robot. We decided *a priori* to target the size of a three and a half year old child (approximately $1m$ tall). Actual dimensions were taken from studies in ergonomics and x-ray images [22]. This size can be achieved with current technology. QRIO [26] is an example of a robot similar in size although with less degrees of freedom. In particular, our specifications had to consider hands and moving eyes. Also, we wanted to consider the workspace and dexterity of the arms and thus a three degree of freedom shoulder was a requirement.

Considering dynamics, the most demanding requirements appear in the interaction with the environment. Impact forces, for instance, had to be considered for the crawling behavior, but also and more importantly, developing cognitive behaviors such as manipulation might require exploring the environment erratically. As a consequence, it is likely that impacts will occur with various parts of the robot structure. This turns out to require strong joints, gearboxes, and powerful actuators or alternatively passive compliance and soft materials. In order to evaluate the scale (order of magnitude) of the required forces we ran simulations of various behaviors in a reasonable model of the robot. These dynamic simulations provided data for starting the design of the robot.

At a more general level we had then to evaluate the available technology, compared to the experience of the RobotCub consortium and the targeted size of the robot: it was decided that electric motors represent the most suitable technology for the ICub, given also that it has to be ready according to a very tight schedule in the span of the RobotCub project. Other technologies (e.g. hydraulic) are left for a "technology watch" activity and they were not considered further.

In addition, given the size of the robot, and given the power density available, considerations of speed for certain joints lack significance: i.e. given the power and the torques required, speed is a consequence rather than a design parameter. In certain cases, in comparing to human data, clearly also the power density is much lower than desired (e.g. the wrists cannot possibly support the weight of the robot).

Finally, the ICub is not only about motors, sensors are equally important. Also in this case, we had to deal with and exploit the available technology as best we could. The robot has vision, audition, joint sensors, force sensors, tactile sensors - where possible - and temperature sensors in many of the motors. The robot can give feedback through a speaker. ICub will thus include a plethora of sensors as cameras, microphones, gyroscopes, linear accelerometers, encoders (or other positional sensors), temperature and current consumption sensors, force/torque, and tactile sensors. The choice of these components is clearly related to the robot specifications.

To recapitulate, the constraints of size and available technology determine a good part of the design choices - i.e. our freedom in deciding which components to use. In parallel, we simulated some of the robot's behaviors to determine the required joint torques. These two pieces of information were then used in selecting the best available motors compatible in size, torque, and strength. As we mentioned earlier, speed is a consequence rather than a design parameters here, although, in simulation we examined the dependency of speed to torque for crawling.

Other design choices are related to the embedded electronics and the structure of the software. The ICub will have many sensors and actuators working in parallel. We would like to exploit this parallelism also at the computational level and, consequently, the ICub API was mapped one-to-one onto YARP.

5.3 Modularity

The ICub design is modular across two dimensions, namely, the mechanical hardware and the control structure. Mechanically, the robot has a certain degree of modularity which allows for improvements without a full-blown re-design activity. The controller is modular in the sense that it is made of several layers. Each layer can be replaced with a different technology and/or implementation without much suffering.

When we consider hardware modularity, we need to strike a balance between the desirability of a global optimization and the advantages of modular and dependable design. The current design probably reflects more the desire to achieve certain functionalities, within a given size, in a constrained setting of three years dedicated to design rather than the search for the quality and maintainability of the robot in the long term. In essence, the ICub is and will remain a research platform. It cannot be considered akin to the AIBO, nor a more industrial realization like the HRP2.

In spite of these stringent requirements the ICub shows modularity and macro-subgroups can be identified in the hand plus forearm, in the arm (entire arm),

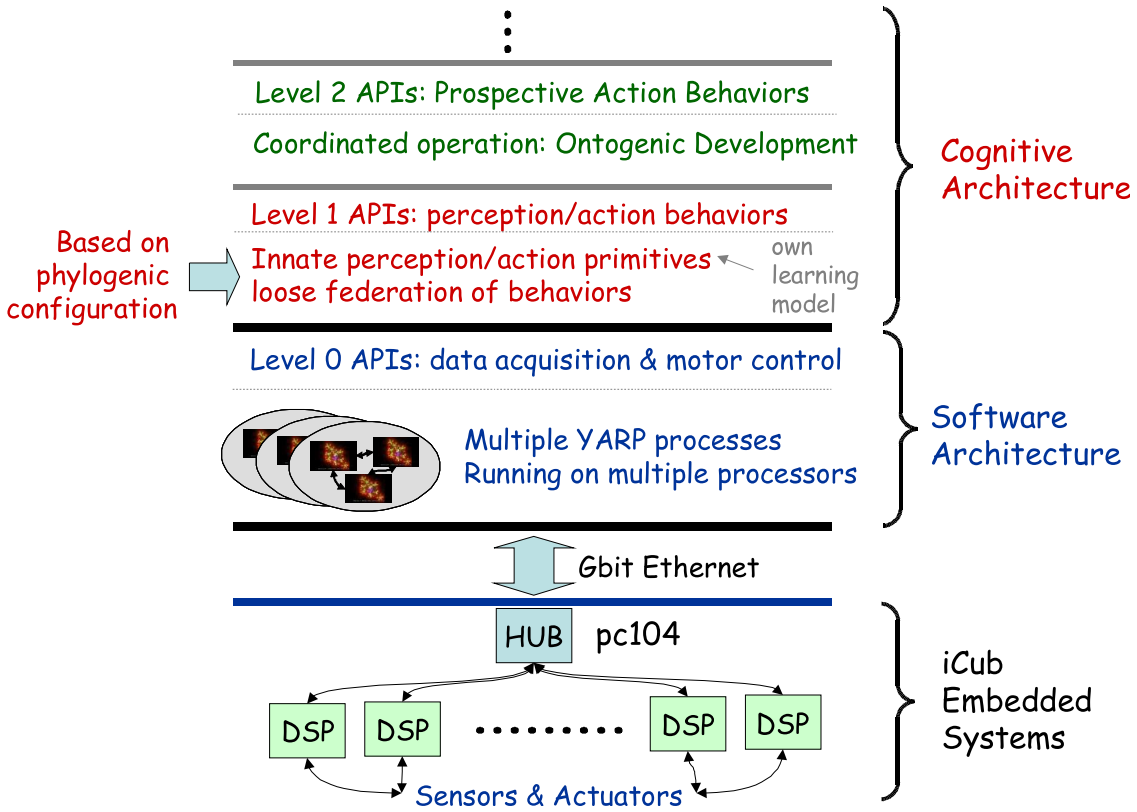

Figure 10. The layered structure of the ICub. The lowest level is the DSP layer which directly connects to the motors and/or sensors. The next hardware level is represented by the PC104 HUB which interfaces on one side to all data sources and controllers and on the other to the GBit Ethernet network. The next layer is a distributed computation engine made of a set of standard PCs which communicate through YARP. On top of this the RobotCub partners will develop a cognitive architecture. Communication is defined by protocols as for example, from the DSP to the motors, from the PC104 to the DSP, and from the YARP processes to the PC104. Standardization at this level favors reusability and dependability of the system.

in the head, the torso, and the legs. These parts can be built and maintained, developed, and assembled separately. Finer grained modularity is not possible because of the placement of the motors and the routing of the tendons. The electronics represent another element of complication since the control cards for certain groups are not localized within the groups (e.g. the hand controllers are in the upper arm section).

Assembly techniques have been considered for mechanical parts and details have been optimized to favor mechanical realization (e.g. tendon routing has been considered and the assembly sequence optimized whenever possible). Nonetheless the realization of the hands (the most complicated parts of the robot) requires considerable time and effort.

At the controller level, modularity is described by at least three layers:

- the DSP-controller level;
- the HUB-coordination level (interface);
- the control architecture.

The DSP level consists of a set of controller cards that can drive the motors directly but also by virtue of programmability enable the preparation of local sophisticated control algorithms. These controller cards were specifically designed for the ICub. They communicate through a set of four CAN bus backbones to a Pentium-based HUB card which can do both synchronization of sensorial and motoric data and run simple control loops in case they are needed to be local to the hardware (for very tight timing). The Pentium, a PC104 format CPU card, is interfaced to YARP processes through a Gbit Ethernet cable. The interface at this level is fully YARP-compatible and specified at the level of ports or device drivers. The YARP processes form the control architecture and can implement complex cognitive behaviors (as indicated in Figure 10).

Protocols are specified at each level. Electrical between the controllers and the motors (determined by the motor specifications), software and electrical (CAN) between the DSP and the PC104 HUB, also software at the level of the YARP packets that travel on the GBit Ethernet cable, and clearly software between the modules of the cognitive architecture. Replacement of components, as long as the protocols remain unchanged, is likely to require only the redesign of the appropriate layer. For example, the obsolescence of the DSP microcontroller currently in use may lead to a new version that can be made compatible with the current CAN bus specification.

6 Conclusions

In recent years we have seen the beginning of many new and ambitious robotic projects [8, 9, 1, 28]. However research to provide intelligence to these complicated robots is advancing at a snail's pace. Accumulating knowledge in the form of working demonstrable systems is plagued by the difficulty of forming teams, on agreeing on standards, and in general by the lack of a critical mass in any existing laboratory no matter the size or funding.

The problem of artificial intelligence is a deep one, and since it began to be investigated, each generation of researchers has grossly underestimated the problems. For example, the Summer Vision Project of 1966 at the MIT AI Lab planned to implement figure/ground separation and object recognition on a set of objects such as balls and cylinders in the month of July, and then

extend that to cigarette packs, batteries, tools and cups in August [14]. As it turns out, if they had written decades rather than months, it would still have been over-ambitious. Significant progress can certainly be made either because of a breakthrough in our understanding of the problems or through a slower accumulation of knowledge. Or it can be due to a combination of these two elements. We, like many others, are drawn to robotics as a way to confront the "real" problems of intelligence head on. This has the advantage of exposing unforeseen opportunities that embodiment brings with it [16], but the downside that it requires a lot of time spent building hardware. It would be beneficial to build a community that can accumulate knowledge and make effective progress, and to expand the niche of humanoid robotics and artificial intelligence to the point where it is healthy and self-sustaining.

In this respect, the parallel with the commercial PC is easily made. The success of the PC was determined, among other factors, by the definition of hardware standards that everybody could understand, copy, and reimplement. From time to time new standards were required (e.g. the ISA bus slowly left space to PCI slots) but the system flourished. Under the hood, the PC is a few orders of magnitude faster and of larger storage capacity. On the software side, the benefit of a common architecture allowed the creation of operating systems and application software consisting of several millions of lines of code. Without a standard hardware things might have been more difficult. A PC of today is the modern version of the Ship of Theseus [4], everything changed but the PC is still considered a PC.

Is robotics really facing the same challenges as the computer industry three decades ago [5]? It is clearly difficult to foresee the future of humanoid robotics. However a few dedicated software platforms are appearing as either commercial [20] or academic [24] products (see also [10] for a survey). It is easier to imagine a scenario where common standards both in software and hardware will find the fertile soil to flourish when isolated breakthroughs will happen.

The problem of dealing with diverse hardware and software in robotics is a complicated one – see [13] for a good description of the many and various problems. The key insight from the Free Software community is the value a common social contract, granting mutually beneficial rights that greatly reduce both the direct and organizational cost of software integration. Regardless of the technical measures we pursue, adopting such a social contract in at least a part of the humanoid robotics community would be a key advance. We believe that this will occur naturally "bottom-up" through pseudo-evolutionary forces: models of software development that are long-lived and fertile will sur-

[4] The Ship of Theseus – the mast gets replaced, the planks get replaced, over time everything may get replaced, but it is still in some important sense the same ship ("paradox of identity")

vive, other forms will die off. The rate at which this will occur is hard to say, and could be influenced by education. For example, a common fear is that such approaches are incompatible with commercial exploitation; in fact they are not, as has been learned in many other fields including embedded devices [6]. They do change the rules of the game though, which is disruptive. We should welcome that disruption.

7 Acknowledgments

YARP and ICub make heavy use of software released under free and open licenses – thank you world. The author would like to gratefully acknowledge contributions to YARP from Radu Bogdan Rusu, Alexis Maldonado, Eric Mislivec, Christopher Prince, Charles C. Kemp, Francesco Nori, Julio Gomes, Alexandre Bernardino, Carlos Beltran, Jonas Ruesch, Assif Mirza, Hatice Kose-Bagci, Jose Gaspar, Claudio Castellini, Michael Bucko, Nelson Gonçalves, Marco Barbosa, Tomassino Ferrauto, Boris Duran, Mattia Castelnovi, and Alessandro Scalzo (if we missed anyone, plese let us know). We'd also like to thank contributors to the ICub platform amongst the RobotCub Partners and the RobotCub Consortium. The authors were supported by European Union grant RobotCub (IST-2004-004370).

References

[1] Bryan Adams, Cynthia Breazeal, Rodney A. Brooks, and Brian Scassellati. Humanoid robots: A new kind of tool. *IEEE Intelligent Systems*, 15(4):25–31, 2000.

[2] Cynthia Breazeal, Aaron Edsinger, Paul Fitzpatrick, and Brian Scassellati. Active vision for sociable robots. *IEEE Transactions on Systems, Man, and Cybernetics, A*, 31(5):443–453, September 2001.

[3] R. A. Brooks, C. Breazeal, M. Marjanovic, and B. Scassellati. The Cog project: Building a humanoid robot. *Lecture Notes in Computer Science*, 1562:52–87, 1999.

[4] Toby H.J. Collett, Bruce A. MacDonald, and Brian P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation*, Sydney, Australia, December 2005.

[5] Bill Gates. A robot in every home. *Scientific American*, pages 58–65, January 2007.

[6] J. Henkel. Selective revealing in open innovation processes: the case of embedded Linux. *Research Policy*, 35(7):953–969, 2006.

[7] Stephen D. Huston, James C. E. Johnson, and Umar Syyid. *The ACE Programmer's Guide*. Addison-Wesley, 2003.

[8] K. Hirai, M. Hirose, Y. Haikawa, and T. Takenaka. The development of honda humanod robot. In *IEEE International Conference on Robotics and Automation*, pages 1321–1326, 1998.

[9] K. Kaneko, F. Kanehiro, S. Kajita, and H. Hirukawa. Humanoid robot HRP-2. In *IEEE International Conference on Robotics and Automation*, pages 1083–1090, 2004.

[10] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Autonomous Robots*, 22(2):101–132, 2007.

[11] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. YARP: Yet Another Robot Platform. *International Journal on Advanced Robotics Systems*, 3(1):43–48, 2006.

[12] Lorenzo Natale. *Linking Action to Perception in a Humanoid Robot: A Developmental Approach to Grasping*. PhD thesis, DIST, University of Genoa, Italy, February 2005.

[13] Issa A. D. Nesnas. The CLARAty project: Coping with hardware and software heterogeneity. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, pages 31–70. Springer-Verlag, Berlin, 2006.

[14] S. Papert. The summer vision project. Memo AIM-100, MIT AI Lab, July 1966.

[15] B. Perens. The open source definition. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open Sources: Voices from the Open Source Revolution*. O'Reilly and Associates, Cambridge, Massachusetts, 1999.

[16] Rolf Pfeifer and Josh C. Bongard. *How the Body Shapes the Way We Think: A New View of Intelligence (Bradford Books)*. The MIT Press, 2006.

[17] Internet Site. Activmedia robotics. www.activrobots.com.

[18] Internet Site. CMake cross platform make. http://www.cmake.com.

[19] Internet Site. K-Team Corporation. www.k-team.com.

[20] Internet Site. Microsoft robotics studio. http://msdn.microsoft.com/robotics.

[21] Internet Site. Sony AIBO Europe - official website. www.sonydigital-link.com/aibo.

[22] A.R. Tilley. *The Measure of Man and Woman: Human Factors in Design*. Wiley, book&cd-rom edition, December 2001.

[23] N. G. Tsagarakis, G. Metta, G. Sandini, D. Vernon, R. Beira, F. Becchi, L. Righetti, J. Santos-Victor, A. J. Ijspeert, M. C. Carrozza, and D. G. Caldwell. iCub - the design and realization of an open humanoid platform for cognitive and neuroscience research. *Journal of Advanced Robotics*, 21(10), 2007. in press.

[24] Richard T. Vaughan and Brian P. Gerkey. Reusable robot code and the player/stage project. In Davide Brugali, editor, *Software Engineering for Experimental Robotics*, pages 267–289. Springer-Verlag, Berlin, 2006.

[25] Georg von Krogh and Eric von Hippel. The promise of research on open source software. *Management Science*, 52(7):975–983, 2006.

[26] Sony global - QRIO. http://www.sony.net/SonyInfo/QRIO/, 2005.

[27] European commission, unit e5, home page. http://cordis.europa.eu/ist/cognition/index.html, 2007.

[28] Y.Sakagami, R.Watanabe, C.Aoyama, S.Matsunaga, N.Higaki, and K.Fujimura. The intelligent ASIMO: system overview and integration. In *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'02)*, pages 2478–2483, 2002.