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 dead ends, in that the software and hardware they produce disappear without trace. This is true for all sorts of projects, but it is particularly true in robotics since our work may be tied to uncommon hardware and a small group of users. Humanoid robotics is currently in exactly this condition, leading to a lot of "churn".

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 enable propagation of our hardware designs to interested parties.

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 network(s) and protocol(s) (thus allowing those details to change, as they often do). We develop a methodology for interfacing with devices (sensors, actuators, etc.) that again encourages loose coupling and can make changes in devices less disruptive. We discuss the problem of incompatible middleware, and how we are doing our part to avoid it. We emphasize the utility of the Free Software approach to software distribution and licensing, and specifically the benefit it brings to small communities with idiosyncratic requirements.

Of course, it would also be better if humanoid robot software had larger number of users with similar hardware, and we're doing our part to encourage this by releasing the design of our iCub humanoid under a free and open license, and funding development using this platform.

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

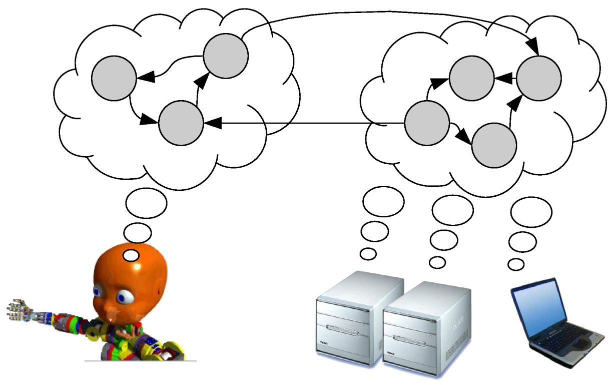

Figure 1. Basic robot model. We assume a set of processors, some of which may be on the robot, some of which may not be. We assume that the computers are diverse: they may have different devices, operating systems, processors, and the programs they run may use different languages, libraries, etc. We develop methods for interfacing with devices and communicating between modules that are useful on any heterogeneous system. We extend our reach by exploiting key Free Software projects dealing with and enabling diversity in operating systems, build systems, and programming languages. Our contribution, made as Free Software, is a system for dealing with and enabling diversity in transport mechanisms and novel devices. We use this software to support our open robot platform, the iCub humanoid, whose design will be available under free and open licensing.

1 Introduction

Software development is, in some ways, like natural evolution. 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 very large (for example, commercial PCs), some are tiny (a newly developed humanoid). Software evolves pretty quickly as new technologies get proposed and hardware changes; if trapped in a too narrow niche it soon tends to become obsolete and die, together with the efforts of the developers who have contributed to it. In the academia, software written for a robotic project suffer of this plague, because it is rarely written by professional developers, and often not the main goal of the efforts of the people who are working on it. For researchers software development is a time consuming and tedious task that takes away time and efforts that could be better spent doing research. Yet at the same time, software development in robotics is a delicate and crucial task that cannot be easily delegated to untrained persons. In research laboratories fast changing hardware and lack of human resources too often narrow the niche in which software lives.

Research to provide intelligence to these complicated robots, perhaps of humanoid shape, also suffers. 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 seems to be more baffling than what originally thought **??** and the year 2006 celebrated the 50^{th} anniversary of the workshop at Dartmouth College which is unanimously considered the birth of artificial intelligence at least in some modern sense **??**. A generation of researchers did not manage to make the progress that was certainly hoped for. Significant progress can certainly be made either because of a breakthrough in our understanding of the problem or through a slower accumulation of knowledge. Or it can be due to a combination of these two elements. Recently, the paradigmatic shift toward embodiment seems to require also the realization of appropriate robotic hardware **??**. 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. A PC of today is the modern version of the Theseus' ship, everything changed but the PC is still considered a PC. (((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").))) 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 creating operating systems and application software consisting of several millions of lines of code. Without a standard hardware things might have been more difficult. It is clearly difficult to foresee the future of humanoid robotics. It is easier though to imagine a scenario where common standards both in software and hardware will find the fertile soil to flourish when isolated breakthroughs will happen.

Research groups that all use a specific robot (Khepera, Pioneer, AIBO, ...) often form a natural software community. But each alone is a small subset of robotics.

Groups developing new robots face obstacles. There are big barriers to software collaboration: differences in sensors, actuators, and bodies; differences in processors, operating systems, libraries, frameworks, languages, compilers.

Computer science and PCs operate in a world that has been at least partially commodified; research groups in CS do not have to reinvent the operating system for every project they undertake.

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

The only viable solution to these problems is to facilitate code reuse both in time and between different people or institution. For projects of reasonable size this means following a *modular* approach, where software is divided in independent modules, 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 causing catastrophic effects. To take advantage of code written by other people in different contexts code must be as friendly as possible to other libraries avoiding to impose constraints and dependencies at all levels, from the hardware architecture to the development environment and programming language. Dependencies between modules should be minimized also from the point of view of performances; as long as resources are available the addition of new components should not clash with the existing ones. In particular the *timing* of signals should be preserved irrespective of the number of modules that are running at any given moment in the whole system.

The robotic platform can be seen as another factor on the equation of code reuse. A common hardware, common protocols, electrical standards, sensors, etc. can certainly make the life of the researchers easier. As for software, modularity can play a role in the hardware design too.

Following the Open Source Software 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.

Our motivation comes from the condition of humanoid robotics research, but most of this paper is not specific to that field.

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. There is clearly a tension here between providing a consolidated system and giving enough freedom to change every single part via upgrades and replacements. We would like to be able to do so both in software and hardware.

New devices come out all the time – needs to be easy to connect them to existing code. YARP needs a minimal "wrapper" class to match vendor-supplied library with relevant interfaces that capture common capabilities. YARP encourages separating configuration from source code – separating the "plumbing". Devices and communications remain distinct concerns. The goal is to allow collaboration between groups whose robots have different devices and make device changes less painful. We also want to have the ability to be able to switch from remote use of device to local use and vice versa without pain, without compromising the efficiency of local access.

Robots need an analogue of the operating system of a computer, or the nervous

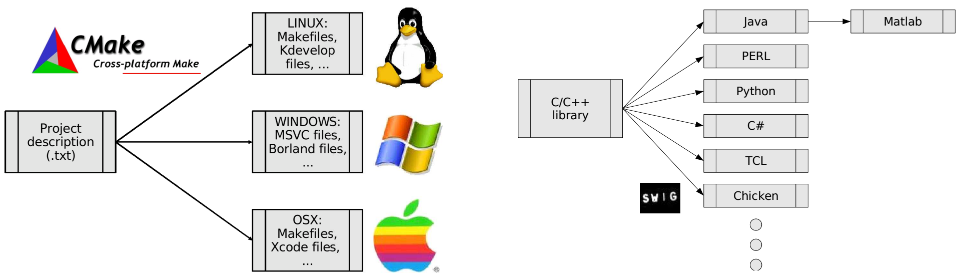
CMake
Cross-platform Make
Project description (.txt)
LINUX: Makefiles, Kdevelop files, ...
WINDOWS: MSVC files, Borland files, ...
OSX: Makefiles, Xcode files, ...
C/C++ library
SWIG
Java
Matlab
PERL
Python
C#
TCL
Chicken

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. SWIG (right) takes C/C++ source code and generates "wrappers" for it, usable from many different languages (including Matlab via Java).

system of an animal. We focus on the key issue of how information travels between processors, sensors, and actuators.

We use other key free and open source projects to make our libraries usable in as many environments as possible. We use the ACE operating system portability layer (but *not* the associated implementation of CORBA), the CMake build system, the SWIG wrapper generator.

2 A software ecology

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 many architectures/frameworks/... for modular robot systems. The prime concern for any such system should be that it is not a "black hole" – that once a piece of software has been adapted to a particular framework, it takes a lot of work to extricate it again and apply it to another. That would be a bit self-defeating.

So modularity alone is not a solution to software reuse, since different organizing frameworks or architectures may be mutually incompatible. It is import that modules developed can fit into a broader "ecology" than just the framework/architecture of the developers. By ecology, we mean the messy complicated collection of niches world-wide in which software development occurs.

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 a platform to another. On the other hand, following a modular approach, we would like our software to be as flexible as possible and adapt to needs of the user and the platform that he uses. In YARP, when possible, 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 gets compiled only when the supporting libraries (mainly GTK+) is installed in the system. Another example is the mathematical library which is built on top of the GNU Scientific Library. The idea is that only the modules that only and *all* the modules that can be compiled are actually built.

To avoid asking the user to go through a tedious and long process of manual configuration, a program automatically inspects the system and generates the files necessary to compile YARP on the host system (e.g. make files in Linux systems, or Visual Studio project files in Windows). This simplifies development, because it is no longer required to distribute build files for all possible platforms.

Among the available tools for automatic configuraton of software packages, we decided to use CMake (see Figure 2). CMake is cross-platform open-source, make 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. Through CMake the build process of YARP is robust, simple and flexible. CMake is free and open-source, with a healthy community of developers.

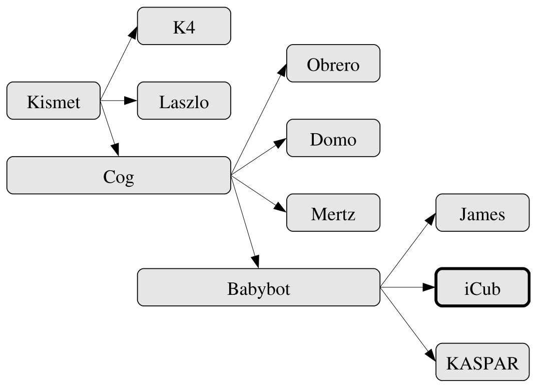

Figure 3. A potted history of YARP. YARP was born on Kismet (CITE), grew up on Cog and BabyBot (CITE), and serves as the software architecture for the iCub humanoid (CITE). Along the way other humanoids have also used the system. With iCub, we are trying to create some hardware "genes" that can travel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

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 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 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"-like 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 [9]. 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 [3].

2.3 Interoperating with other architectures/frameworks

The closest project in spirit to YARP is that of the Player project [8]. 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. 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 srobot simulator of that name from a YARP Network. The driver mechanism in both projects gives a very straightforward way to integrate quickly with what would otherwise be incompatible middleware.

Both projects are free and open source. They both have documented network protocols. Both have made an effort to allow different transports, and to be portable. Given all these similarities, it is reasonable to wonder whether the projects could be merged. From the YARP perspective, there doesn't seem any compelling reason to do so right now; individuals who've needed parts of both systems have been able to do so through the driver mechanisms. From the Player perspective, YARP's dependence on ACE is probably undesirable, and YARP uses more C++ features than the C-with-Classes style of Player (which is probably a better idea for portability). Also, we use different build machinery (autotools versus CMake). Due to recent changes in Player, communications could probably be adapted. Devices are different in YARP and Player. YARP starts with just a thin C++ wrapper which permits direct function calls; Player devices require message passing (even if the message passing is just internal rather than across a network).

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 the operating system defines a constant interface (threading, memory, network, filesystem). Unfortunately more specific harware (motor control boards and frame grabbers are just popu-

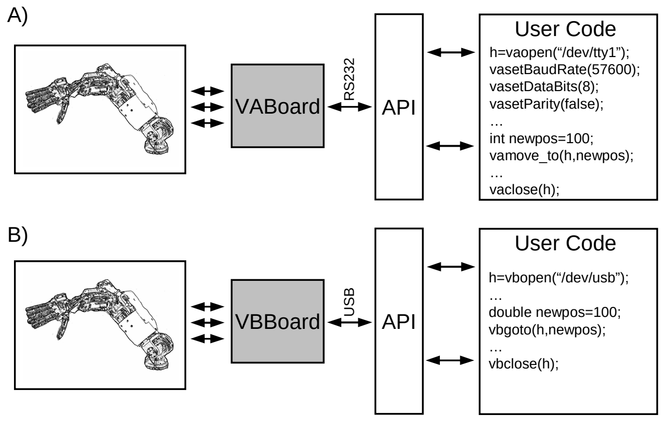

Figure 4. Example of code dependency. VA-Board is a motor control board which interface to the robot through serial port. Changes in the board (in the example above the VB-Board has s USB interface) are propagated to the user code.

lar examples) requires a more sophisticated mechanism. In these cases vendors provide device drivers and a set of API to simplify code development. The API comes in the form of a static or dynamic library which must be linked in the user 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 venodor 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. Often even similar devices have different "initialization" procedure. 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 recente 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)). When possible these (and other) differences must be hidden if we want that the same user code can effectively interface to the two devices.

YARP implements the separation between user and device specific code in three ways: i) definition of interfaces for families of devices ii) localization and separation of device initialization and creation ii) creation of network wrappers and separation between devices and communication.

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 ensamble of the functionalities a device must implement to be able to provide that interface. The implementation of a YARP device consists in a “wrapper” class which implements all methods specified by the interface. A single device can of course expose more than a single interface (in C++ this is implemented through multiple inheritance). All details specific of the hardware (vendor's API and library) are handled here and are hidden by its interfaces. The idea is that changes in the hardware are catched 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.

Another aspect that we have pointed out above, is the fact that initialization parameters may introduce annoying dependencies in the user 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;
```

and

```
virtual bool close()=0;
```

This *open* method initializes the device. Initialization parameters are passed to the function as a list of key-value entries (a *Searchable* object). A *Searchable* can contain all possible parameters 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 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 *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.

Interfaces to broad families of devices have been defined in YARP. For exam-

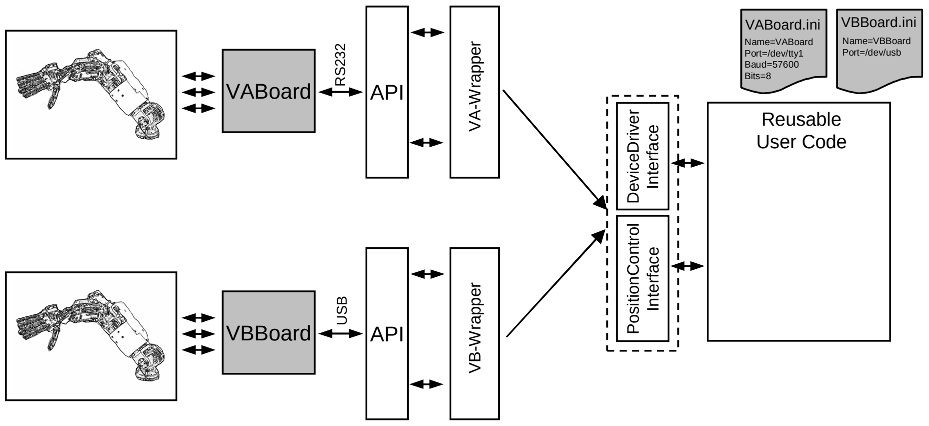

Figure 5. Interfaces allow code reuse. VA-Board and VB-Boards (see Figure (4)) now implement the same interfaces (through their respective wrapper classes). The user code access the hardware through these interfaces and is not aware of the details of how the methods are actually implemented. The different initialization parameters are listed in configuration files and are thus separated from the code.

ple this is a partial list of the interfaces that have been defined for devices belonging to the family of frame grabbers:

- *IFrameGrabberRgb*, which defines the interface to devices which generate a stream of color images. 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 a typical frame grabber device. Methods of this interface allows controlling the parameters of acquisition of the device, 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 stadard 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 type of functionalities and control algorithms. For this reason interface to control boards has 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 of velocity of each axis
- *IPositionControl*: methods to control motion of each axis by specifying its position (usually referred to as "position control")
- *IVelocityControl*: methods to contorl motion of each axis by specifying its velocity (usually referred to as "velocity control")
- *ITorqueControl*: defiens methods to control the amount of force/torque ex-

erted by each axis (usually referred to as "torque or force control").

These last interfaces are indipendent 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, like for example read or set the values of the gains.

As an example Figure (5)) shows how YARP interfaces allows code reuse in cases two different motor boards are used to control the same piece of hardware (the problem described in Figure (4).

3.2 A factory of devices

We have seen that forcing access devices only through interface achieve 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 that this introduces device dependent 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 forces all initialization procedures to be performed inside the *open* method. Device creation is instead 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 from a list of initialization parameters, creates the device, and initializes it through the *DeviceDriver* interface. If the process is successfull a valid pointer to the device is returned. This pointer is the only "access point" to the device and its interfaces. Other interfaces can be obtained by casting this pointer to the appropriate virtual class (in C++ this can be safely done through dynamic cast).

The whole process of creation, initialization and interface access is managed by the *PolyDriver* object. The user has only direct access to the *PolyDriver*, to which he asks the creation of a particolar device, through the *PolyDriver::open())* method. This request is forwarded to the *factory* which creats an instance of the particular device driver the user wants to use. Device drivers are uniquely identified through a symbolic name; the *factory* searches the list of device for an entry whose name matches the one that is requested and, if the match is found, it calls the appropriate constructor. 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*; the user can access only to copies of this pointer thourgh calls to the *PolyDriver::view()*

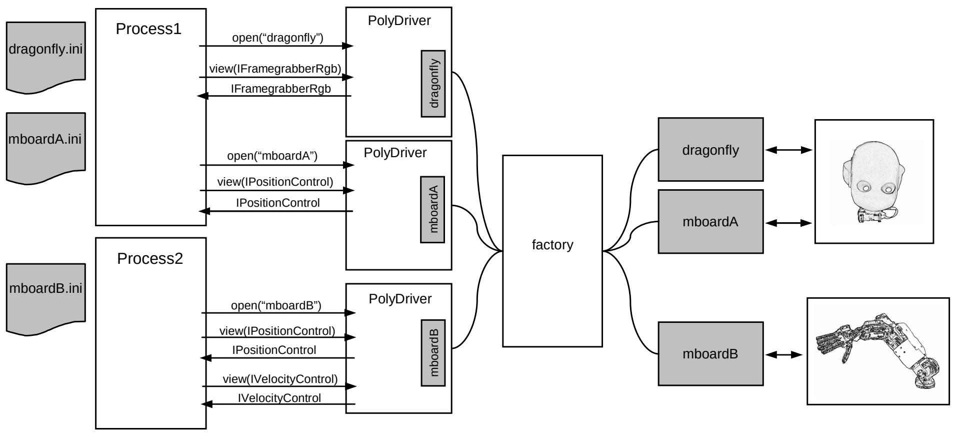

Figure 6. Device creation and initialization. Creation of devices in YARP in delegated to a *factory* object. Users access devices through instances of the *PolyDriver*. The Figure describes the following situation. Process1 controls a robotic head and needs to access to the robot frame grabber (whose symbolic name is "dragonfly") and to the control board connected to the motors of the head ("mboardA"). Process1 creates *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 acquires 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.

method, which performs a dynamic cast of the internal pointer to the device driver to expose the requested interface before returning it (this mechniam is sketched in Figure (6) *NOTE: not sure this picture is really needed, consider removing it*).

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 controlling 4 axes. This device supports the *IPositionControl* and *IVelocity* 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 (e.g. *test_ motor*):

```
PolyDriver device;
device.open(\"test\_motor\");
```

Now we can ask interfaces to *test_ motor* by calling the *PolyDriver::view()*method:

```
  IPositionControl *ipos=0;
  device.view(ipos);

  IVelocityControl *ivel=0;
  device.view(ivel);
```

Checking if ivel and ipos are not null assures that *test_ motor* really supports the respective interfaces.

We can now call methods of the *IPositionControl* interface to control joint 0 to move 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);
```

Or use the *IVelocity* interface to servo joint 1 at the velocity of $5\frac{deg}{s}$ with the acceleration of $100\frac{deg}{s^2}$:

```
ivel->setRefAcceleration(1,100);
ivel->velocityMove(1, 5);
```

3.4 Device Remotization: Binary Interfaces

A final level of separation is achieved by supporting device remotization. This feature is important because flexibility imposes that processes are written so that they can be easily moved across distinct machines, if needed. Knowledge of whether a process is accessing a given device locally or remotely would clearly limit this flexibility, because it would force modification in the code of the process itself. Remotization also facilitate portability across different platforms, as it naturally defines a "binary" 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 *add reference to port section* and on two *Network Wrappers* one acting as a *Server* and the other acting as a *Client*. Both *Network Wrappers* are devices implementing the very same set of interfaces as the device they wrap. Both devices act proxies and talk to each other using a predefined protocol, which involves one or more YARP Ports configured for RPC or streaming (see 7)

The *Server Wrapper* creates an instance of the wrapped device and forwards

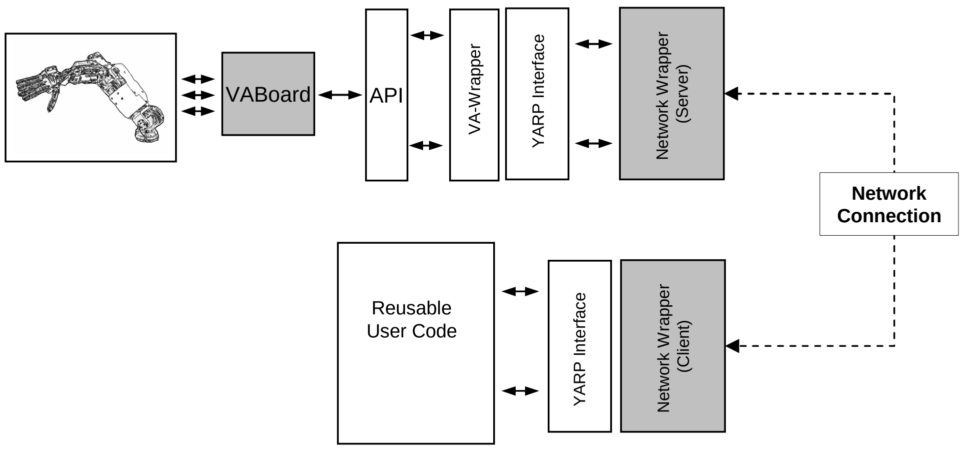

Figure 7. Network wrappers allow device remotization. A generic Network wrapper exports the YARP interfaces so that they can be accessed remotely by another machine through an RPC mechanism. At the other side of the communication the client wrapper exports the network interface back to the YARP interfaces from which the device is transparently accessed by the client code. The only difference between a local device and a remote one is in term of performances (the time it takes to execute a given function).

requests from the incoming connections to the device by calling its interfaces. If the request involves a reply this is sent back to the calling port so that it is received by the remote client. The mechanism used by the *Server Wrapper* to access the local device is the same we have described for the user code; as such the *Server Wrapper* is a total independent entity that can be reused for all devices implementing the same interface(s).

The process at the other side of the communication creates the *Client Wrapper*. The latter exports exactly the same interfaces as the device driver it wraps so the process is not aware that it is not talking to a real device. The job of the *Client Wrapper* is to convert calls from the user code in packets of bytes and send them to the other end of the communication, and, in case a reply is expected, waits for data and dispatch it to the calling process. The responsibility of the proxies among the other things is to perform the marshalling and de-marshalling of the information so that it is correctly interpreted by the distinct platforms (this is required for systems in which data is represented with different standards).

Orphan text:

The goal of YARP is to support humanoid robotics, in which a broad variety of hardware is often employed. In 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. In this sense device-level software development is a very time consuming and tedious task, so the possibility to share code in this context is extremely profitable. YARP devices consists in a set of *wrapper classes*, which usually link vendor's libraries....

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 [4] and the related work review in [1]). 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 based 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.

Communication in YARP generally follows the *Observer* design pattern. Special 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 several underlying communication protocols. Figure 8 is a very simple network of ports for a visual tracking application.

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.

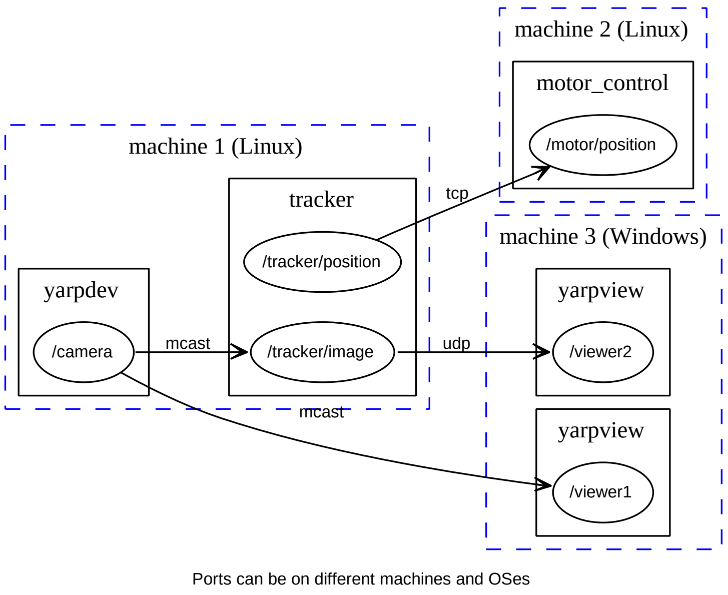

Figure 8. Port example. Images are transmitted from a camera ("/camera") port to a viewer ("/viewer1") port and the input of a visual tracker ("/tracker/image"). The tracker annotates the image, for example by placing a marker on a tracked point, and transmits that to another viewer ("/viewer2"). The 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. Every connection can take place using a different protocol and/or physical network.

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 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. In other words, the edges in the YARP Network are entirely mutable.

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 protocol 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.

The YARP name server is a server that tracks information about ports. It indexes this information by name, playing a role analogous to DNS on the internet. To communicate with a port, the properties of that port need to be known (the machine it is running on, the socket it is listening on, the carriers it supports). The YARP name server offers a convenient place to store these properties, so that only the name of the port is needed to recover them.

If messages follow YARP guidelines, then they can be automatically converted to and from a “text mode” connection, for easy human monitoring and intervention in the system.

Connections between ports in YARP can carry replies if desired, so conventional “RPC” (remote procedure call) style synchronous operation is possible. This is a trade-off though, because this model of communication can mske a network brittle by introducing strong coupling of timing between processes.

We’ve separated out most of the plumbing. We get to change it dynamically (handy). More importantly, we have better modularity. Programs can be moved around as load and OS/device/library dependencies dictate. Fundamental protocol for communication can be changed without affecting programs. Better chance that your code can be used by others (even just within your group).

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 much easier to work with and learn about experimentally without extensive study.

The YARP communications system is written in two parts. There are 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. There is no marshalling process at this stage.

The second part of the communication system is a standard data format. This standard is not enforced, 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. There is a general-purpose helper class in YARP called “Bottle” that reads and writes data in this format, but the format is also used by special purpose classes such as Vector and Image. Extensive examples are available on how to generate data in this form.

The bottle representation is based on a nested structure of certain primitive types. Here are some informal examples of bottles, expressed in text form (an equivalent binary form can be used interchangeably):

text-form bottle	**interpretation**
10 20 30	a list of three integers
10.0 20.5 31.4159	a list of three floating point numbers
action "go left" (10 20 30)	a list of two strings ("action" and "go left"), and a nested list of integers
[set] [pos] 1 30.3	a list of two vocabs, an integer, and a floating point number – perhaps to set the position of a specific axis.

The structure is basically that of an s-expression [7], except that the outermost parentheses are omitted. This makes the important and common case of messages without any nesting easier to write for those unfamiliar with s-expressions. The drawback is that this means that a bottle must always be a list, rather than any of the other primitive types, and (depending on the input mechanism) it may also need to be non-empty.

The primitives types available are lists, integers, floating point numbers, strings, blobs (uninterpreted sequence of bytes), and "vocabs". The vocab type is the one unfamiliar type in this list. It is motivated by the dual requirements of efficiency of machine interpretation binary representations, and ease of human reading and writing of text representations. For data with a tag in it upon which a dispatch will occur, it is simplest if that tag is a simple integer in binary mode rather than a piece of variable-length text. Vocabs are represented as simple integers in binary, and short (up to four ASCII character) strings in text, with a one-to-one mapping between the characters and bytes in the integer.

The important point is that under normal operation, ports can be sending fixed size 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 identifiers being used.

The basic constraints on the design of this format were as follows:

▷ Messages in ths format should have a convenient binary and text representation.
▷ The process of translating between binary and text representations should

be mechanical, and "round-trips" should not change the value significantly (except potentially with some round-off in floating point numbers).
- ▷ The text representation should be as easy as possible to express from the command-line of common shells.

The reason for the last item is to make YARP messaging compatible with typical UNIX pipe operation. Such operation does not really survive well when different character encodings may be in use, but still has uses.

The structure can be expressed in several representations. For messaging there are:

- ▷ Binary form – used for efficient messaging that is easily machine readable/writable.
- ▷ Text form – used for messaging that is easily human readable/writable (but less efficient for large messages).

It is also convenient to add two extra representations:

- ▷ Command-line form – arguments to a program.
- ▷ Configuration form – groups and lines in a configuration file.

This is useful when dealing with configuration of devices; since information could come from file, command line options, or across the network, it is convenient to have all the various represenations mapping to a homogeneous structure.

For robotics applications that are confined to a local area network with all source code available, changes to the data format are possible.

The default binary representation for integers on the wire for YARP is little-endian. Conversions happen for non big-endian systems (e.g. MAC PPC). For a network that is all big-endian, the binary representation can be flipped.

This could also be desired for XDR compatibility. The string and blob representations also would need to be updated to do padding which YARP does not do. These changes would be very narrowly localized.

Such a modified system would lose binary-mode compatibility with any modules using the standard YARP binary format, but connections could still be made in text mode.

In principle, evolution of the communication protocol in YARP can be relatively painless. Since new "carriers" can be added, modifications could be placed within a new carrier, while support for older carriers is continued for a generation or two. Ideally, something like today's text mode format should be honored for a long time, as a connection protocol of last resort. (NOTE: this is about more than just the bottle format; reorder).

4.3 Connection protocol

This 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 (think: tcp socket) 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.
- ▷ **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.

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
The first eight bytes of a connection in YARP specify the desired lower-level protocol to use.

▷ **Acknowledgement of payload**: The receiver may acknowledge transmission in some way. Carrier-dependent.

The message phase repeats as often as the user wants.

The description so far sounds 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).

Before sending each message, YARP also expects the endpoint to know whether the connection can be “escaped” – that is, it can accept port-to-port administrative communication that is not passed on as normal user data. The only time escaping is not desired on a carrier is in text-mode, when the bytes transmitted should follow the user data as slavishly as possible. In all other cases, some administrative data is currently packaged by YARP with user data and then extracted at the other end. In future carriers will be given more control over how this material (called the “port protocol”) is embedded.

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 system running, with a 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 code interface that deals with communication. But if for some reason we can't use the YARP codebase, what can we do?

YARP ports listen to incoming tcp connections, always ready to make new connections for input or output. Suppose we can discover that "/motors" is listening on port 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.

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.

user types	**system responds**
$ telnet 127.0.0.1 10022	*(telnet startup message)*
CONNECT foo	Welcome foo
r	do
	100.0 0.0
	do
	100.0 0.0
	(continues streaming motor state)
	...

The "do" is just like "d", to inform the recipient that user data follows. The "o" part means that replies are not expected by the sender, and should not be sent. This seems like a small detail, but in fact when replies are not expected the performance of streaming to multiple targets can be greatly improved in general, so it is worth mentioning.

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 logically it is all the same thing.

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?

To do that, we need to talk to the "yarp server", which is analogous to a standard DNS name server. If we're using YARP, our program would be able to discover where the server is by itself (by checking standard configuration files if available or sending broadcast messages if necessary). Or, if we don't want to use yarp at all, we can simply make a note of what machine and port number it is running on. Suppose it is on 127.0.0.1 listening to port number 10000. We can communicate with the name server exactly as with any regular port.

user types	**system responds**
$ telnet 127.0.0.1 10000	*(telnet startup message)*
CONNECT foo	Welcome foo
d	*(no response, waiting for data)*
help	*(a list of available name server commands)*
d	*(no response, waiting for data)*
query /motors	registration name /motors ip 127.0.0.1 port 10022 type tcp

So we now know how to reach /motors (tcp connection to 127.0.0.1 port 10022). So, with a running YARP system, we have seen how to 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.

5 RobotCub and iCub

a placeholder.

6 Conclusions

A survey of robot development environments [4] (p.s. rules YARP out, claim it has no coherent program, think they were seeing YARP1).

Nesnas [6] - “coping with hardware and software heterogeneity”. It is useful for the list of problems it works through (solutions are not that exciting). Mentions danger of overgeneralizing interfaces, argues for “multi-level abstraction models, object-oriented methodologies and design patterns”. A bit vacuous.

Vaughan [8] - this is a good player/stage paper.

Our previous comments on YARP1 [5] which describe its history and some usage information.

Gates thinks robotics could take off, Microsoft has released a robotics platform [2].

Missing infrastructure. Need a lot of software. Ideally (for researchers starting out) should be commoditized.

comparison with PC:

Difference: now we have the network. Go the player route, rather than the single IDE. Transform from hardware to open protocols as first step. Then whole ecology of computation is available.

Robots aren’t that special. webcam/microphone/games...

The resources available to us are generally lower. Smaller communities. Less software expertise. This could chage.

7 Acknowledgements

YARP makes heavy use of, and is itself, free sofware – thank you world. The authors were supported by European Union grant RobotCub (IST-2004-004370).

References

[1] 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.

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

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

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

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

[6] 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.

[7] Ronald Rivest. S-expressions, May 1997.

[8] 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.

[9] Georg von Krogh and Eric von Hippel. The Promise of Research on Open Source Software. *Management Science*, 52(7):975–983, 2006.