YARP

An Introduction

or ...

how to live in harmony with your (robotic) world
Overview of seminar

1. What is YARP?
2. How does it work?
3. Some examples
4. A (simple) demonstration
5. What can YARP do for me?
6. How to get started

Please feel free to ask questions as we go ...
Yet Another Robot Platform

- YARP is an open-source software library for humanoid robotics

History

- An MIT / Lira-Lab collaboration
  - Paul Fitzpatrick, Giorgio Metta, Lorenzo Natale
- Born on Kismet, grew on COG
- With a major overhaul, now used by RobotCub consortium,
- Used by the broader open-source community
- And of course, KASPAR, here at UH
What is YARP?

- YARP is an **open-source** software library for humanoid robotics
  - Network communication, device abstraction
- Designed to support and encourage:
  - **Collaboration** (code-sharing across space)
  - **Longevity** (code-sharing across time)
- YARP encourages **modular** development of robotics software
- Provides OS and build tool **independence**
  - Also some **language** independence
Modularity

• The opposite of a modular system is a coupled one.

• In a “coupled” system, changes in one part trigger changes in another.
  - Coupling leads to complexity
  - Complexity leads to confusion
  - Confusion leads to suffering

• This is the path to the Dark Side
Why Modularity for Robots?

- Robot code is notoriously *hardware-specific* and *task-specific*.
- But hardware and target tasks change quickly, even within the lifetime of one project.
- Our humanoid robots are far more complex than one person can build and maintain, both in terms of hardware and software.
- They need to be *modular*.
Modularity

- Modular approaches to robotics:
  - **Player/Stage** (mobile robotics)
    - Robot control (Khepera, Pioneer), simulator
  - **Orcos** (industrial robotics)
    - Real-time control, kinematics library, other libs
  - **YARP** (humanoid robotics)

*SOURCE: Chad Jenkins, June 11, 2005, Workshop Introduction Robotics 2005 Workshop on Modular Foundations for Control and Perception*
Escaping the Operating System

- We shield code from the details of the operating system it runs on
  - Then individual projects can use whichever OS we prefer or need (e.g. specific devices or libraries may only be supported on one OS)
- We shield software from the details of the “build tools” used
  - Visual Studio (Microsoft) people and emacs/g++ (Linux etc.) people can finally be friends
OS independence

- Start from ACE - the "Adaptive Communication Environment"
  - Free and Open Source
  - Widely used, widely tested

- YARP uses ACE in its implementation, but doesn't require YARP users to do so
  - ACE is big, complex, daunting, changing
  - You can understand and use YARP without understanding ACE
Build tool independence

- Start from CMake
- Free, Open Source
- CMake lets us describe our programs and libraries in a cross-platform way
- CMake takes care of creating the makefiles or workspaces needed by your preferred development environment
Build tool independence

Project description

LINUX: Makefiles, Kdevelop files, ...

WINDOWS: MSVC files, Borland files, ...

OSX: Makefiles, Xcode files, ...
Integrating other libraries

• With CMake, we can easily include other libraries in a cross-platform way
  - “OpenCV” computer vision library
  - “Boost” peer-reviewed libraries
  - “OpenGL” graphics library
  - “GTK” windowing library ...

• For YARP, we expect users will exploit such libraries, but minimize our own use of them (so as not to force their choice)
Beyond the Operating System

• ACE decouples source code from OS
• CMake decouples compilation from OS
• But, for humanoid robotics, our effective “OS” also includes:
  - Many **special hardware devices**
  - A (typically ever-changing) **network of computers**
• YARP tries to decouple our code from this “OS”
Beyond the Operating System

- YARP shields programs from the details of *how* they communicate
  - We can then reroute this “plumbing” as we wish, e.g. to send output to new programs
- YARP shields users from the details of the *devices* they control
  - The devices can then be replaced over time by comparable alternatives; user code may be useful to others
Communication independence: the Observer pattern

data source, or stream of events

Observer 1

Observer 2

Observer N
YARP Ports

• We follow the **Observer** design pattern.

• Special “Port” objects deliver data to:
  - Any number of observers (other “Port”s) ...
  - ... in any number of processes ...
  - ... distributed across any number of computers ...
  - using any of several underlying communication protocols with different technical advantages

• This is called the YARP Network
A simple example

- In this simple example the “yarp” command line utility is used to create yarp ports ...

  ```
yarp write /seminar/w               yarp read /seminar/r
```

  ```
  /seminar/w                      /seminar/r
  ```

  ```
  yarp connect /seminar/w /seminar/r
  ```

- ... and connect them together
- All output from the write port is sent to the read port
A simple example

- The output from /seminar/w could at the same time be sent to another process through another port
In code (C++)

- Here is some code that opens a port and writes to it

```cpp
#include <yarp/os/all.h>
#include <stdio.h>
using namespace yarp::os;

int main() {
    Network::init();

    BufferedPort<Bottle> in;
    BufferedPort<Bottle> out;
    in.open("/in");
    out.open("/out");

    // Connect the ports so that anything written from /out arrives to /in
    Network::connect("/out","/in");

    // Send one "Bottle" object.
    Bottle& outBot1 = out.prepare(); // Get the object
    outBot1.fromString("hello world"); // Set it up the way we want
    out.write(); // Now send it on its way

    // Read the object
    Bottle *inBot1 = in.read();
    printf("Bottle 1 is: %s\n", inBot1->toString().c_str());

    Network::fini();
    return 0;
}
```
Message in a bottle: an aside

- Messages in YARP are wrapped in objects called “bottles”
- From the YARP 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."

Day 267: After sending out that message in a bottle stating my location, I’ve been bombarded with junk mail.
Typical network of ports

- Connections can use different protocols
- Ports belong to processes
- Processes can be on different machines/os

machine 1: linux

machine 2: linux

camera

/motor/control

/motor/position

/tracker/position

/viewer1

/viewer2

/tracker/image

/motor/position

/motor/position

/motor/position

/motor/position
Beyond the Operating System

- YARP shields programs from the details of how they communicate
  - We can then reroute this “plumbing” as we wish, e.g. to send output to new programs

- YARP shields users from the details of the devices they control
  - The devices can then be replaced over time by comparable alternatives; user code may be useful to others
Another example 😊

- Create a (fake) frame grabber using `yarpdev` e.g.
  - `yarpdev -device test_grabber -framerate 20`
  - creates a device using a generic factory method
  - wraps the device in a generic network interface
- Open a viewer which accept images on its input port and displays them
  - `yarpview -name /viewer1`
- Connect the grabber and viewer
  - `yarp connect /grabber /viewer1 mcast`
  - the optional parameter selects the communication method
YARP Devices

- There are three separate concerns related to devices in YARP:
  - Implementing **specific drivers** for particular devices
  - Defining interfaces for **device families**
  - Implementing **network wrappers** for interfaces
1: implementing drivers

- The first step, creating drivers for particular devices, is obvious; every robotics project needs to interface with hardware somehow.
  - Cameras, microphones
  - Motors, encoders
  - ...
2: families of devices

- The second step, defining interfaces for families of devices, is important in the longer term.
- If you change your camera or your motor control board, how much of your code needs to change too?
- If you view your devices through well thought out interfaces, the impact of device change can be minimized.
Example: image sources

“Get an image” Interface (IFrameGrabberImage)

- Picolo framegrabber
- DragonFly fireware camera
- OpenCV Grabber library interface
- FFMPEG Grabber library interface
- Server/Remote network wrapper
- TestGrabber fake images

specific hardware
widely supported libraries for accessing image sources
any image source, on another machine
fake source for testing
Example: audio sources

“Get a sound” Interface (IAudioGrabberSound)

- Microphone (windows version)
- Microphone (linux version)
- PortAudio library interface
- FFMPEG Grabber library interface
- Server/Remote network wrapper

specific hardware

widely supported libraries for accessing audio sources

any image source, on another machine
3: network wrappers

- The third step, network wrappers, is important to give flexibility.
- You can scale up your computing cluster, or isolate hardware devices that don't play well together, or have specific OS dependencies etc.
Two Views

• YARP offers two views of a robot
  - A set of devices which you can control or query according to a choice of interfaces (device view)
    • If you are responsible for configuring and starting devices, this is the local device view
    • If configuration and starting-up/shutting-down is packaged with the robot, so you don't have to take care of it, this is the remote device view
  - A set of ports to which you can connect and get data or send commands (port view)
Devices

- Local and Remote devices
Modularity revisited

- A device driver implements the DeviceDriver interface at a minimum and also any other interfaces it is going to provide.

```cpp
... class FakeFrameGrabber : public yarp::dev::IFrameGrabberImage, 
    public yarp::dev::DeviceDriver {
...
```

- In code, you open a device like this:

```cpp
...
Property config.fromString("(device fake_grabber) (w 640) (h 480)");
PolyDriver dd(config);
IFrameGrabberImage *grabberInterface;
dd.view(grabberInterface);
```

- This starts and configures the device using a generic device factory method using the options you select.
- Then views the generic device as one that implements the generic IFrameGrabber interface.
Modularity revisited

- A device driver implements the DeviceDriver interface at a minimum and also any other interfaces it is going to provide

```cpp
... class FakeFrameGrabber : public yarp::dev::IFrameGrabberImage,
   public yarp::dev::DeviceDriver {

    ...code to implement open(), close() methods for DeviceDriver and
    getImage(), width() and height() methods for IFrameGrabberImage

- You can open this device and just use it without any bureaucracy:

```cpp
FakeFrameGrabber fakey;
fakey.open(640,480);
ImageOf<PixelRgb> img;
fakey.getImage(img);
...```
Modularity revisited

• But, if we're smart, we'd make as much of our code as possible depend just on the interface IFrameGrabberImage, so that we can reuse it or substitute in a different framegrabber later:

• This is a standard software engineering technique for minimizing unnecessary coupling between modules.

```cpp
// creation and configuration -- depends on specific device type
FakeFrameGrabber fakey;
fakey.open(640,480);
IFrameGrabberImage& genericGrabber = fakey;
// now we only care that our device implements IFrameGrabberImage
ImageOf<PixelRgb> img;
genericGrabber.getImage(img);
```
Modularity revisited

- But, we can go further:
- In order to open the device using the generic factory, we simply register it with YARP ...

\[
\text{DriverCreator } *\text{fakey\_factory} = \text{new DriverCreatorOf<FakeFrameGrabber>}("fakey","grabber","FakeFrameGrabber"); \\
\text{Drivers::factory().add(fakey\_factory); // hand factory over to YARP}
\]

- We can open the device directly with default parameters:

\[
\text{PolyDriver dd("fakey");}
\]

- With some configuration parameters

\[
\text{Property config("(device fakey) (w 640) (h 480)");} \\
\text{PolyDriver dd(config);}
\]

- Or even with a network grabber so that it is available on the network

\[
\text{Property config("(device grabber) (subdevice fakey) (w 640) (h 480)");} \\
\text{PolyDriver dd(config);}
\]
Port view

- Of course a process could start the device, grab frames from the device and make them available on a port.

```cpp
// code as above opens a port viewed through "grabber_interface"
...
BufferedPort< ImageOf<PixelRgb> > outPort;
outPort.open("/grabber/img");

if (grabberInterface != NULL) {
    ImageOf<PixelRgb> imgIn;

    while (grabber->getImage(imgIn)) {

        // Buffered ports require that you get the next outgoing object to put your data in
        ImageOf<PixelRgb>& imgOut = outPort.prepare();

        imgOut.copy(imgIn);

        // Actually send out the image on the port
        outPort.write();
    }
}
```
YARP Network

“Brain” (YARP processes on a robot)

External YARP processes (e.g. monitoring, logging)

Foreign “Edge” processes
The “Edge” of a YARP Network

- To participate in a YARP Network, it is not necessary to use C++
  - The YARP library can be “wrapped” for Java, Matlab (via Java), Python, Perl, C#, Chicken...

- It is also simple to communicate with Ports without using any YARP code
ACE+CMake+Libraries

• With ACE, CMake, and appropriate libraries, we are as portable as Java

• Why program in C/C++?
  - Flexible: as high-level or low-level as we need
  - And for robotics we often need to go quite low-level, e.g. to interface with devices

• YARP makes effort to support other languages via bindings and protocol documentation
The “Edge” of a YARP Network

- User can implement just enough to make a connection to a single Port

  - Easy! Ports support several protocols, so just use the simplest one - a trivial text-mode protocol
  - Don't get efficiencies of more complex protocols but that's often okay

- Called “Edge” of the Network since it is not a true Port, just a connection going “off the map”
“Edge” Example

foreign process (e.g. in Perl)

CONNECT foreigner

Welcome foreigner

d [set] [pos] 3 10.9

YARP Port tcp socket
What can YARP do for me?

• Help you write robot control code that will last and can be shared

• Let you easily spread processes over many machines
  - Audio processing on one, object detection on another, tight-loop control on a dedicated machine, etc.

• Even if you don’t want to control robots, the networking code could be useful in itself

• Free yourself from the tyranny of the operating system for which your control drivers were written

• Make the world a better, friendlier place … ;-)
How to get YARP

• Download:
  http://yarp0.sourceforge.net

• Or via CVS
  See the documentation ...

• Documentation:
  http://yarp0.sourceforge.net/specs/dox/user/html/

• More notes at the summer school site:
  http://eris.liralab.it/wiki/VVV06
Thank you all for your attention ...

Please come and ask me if you need any help with installing or using YARP