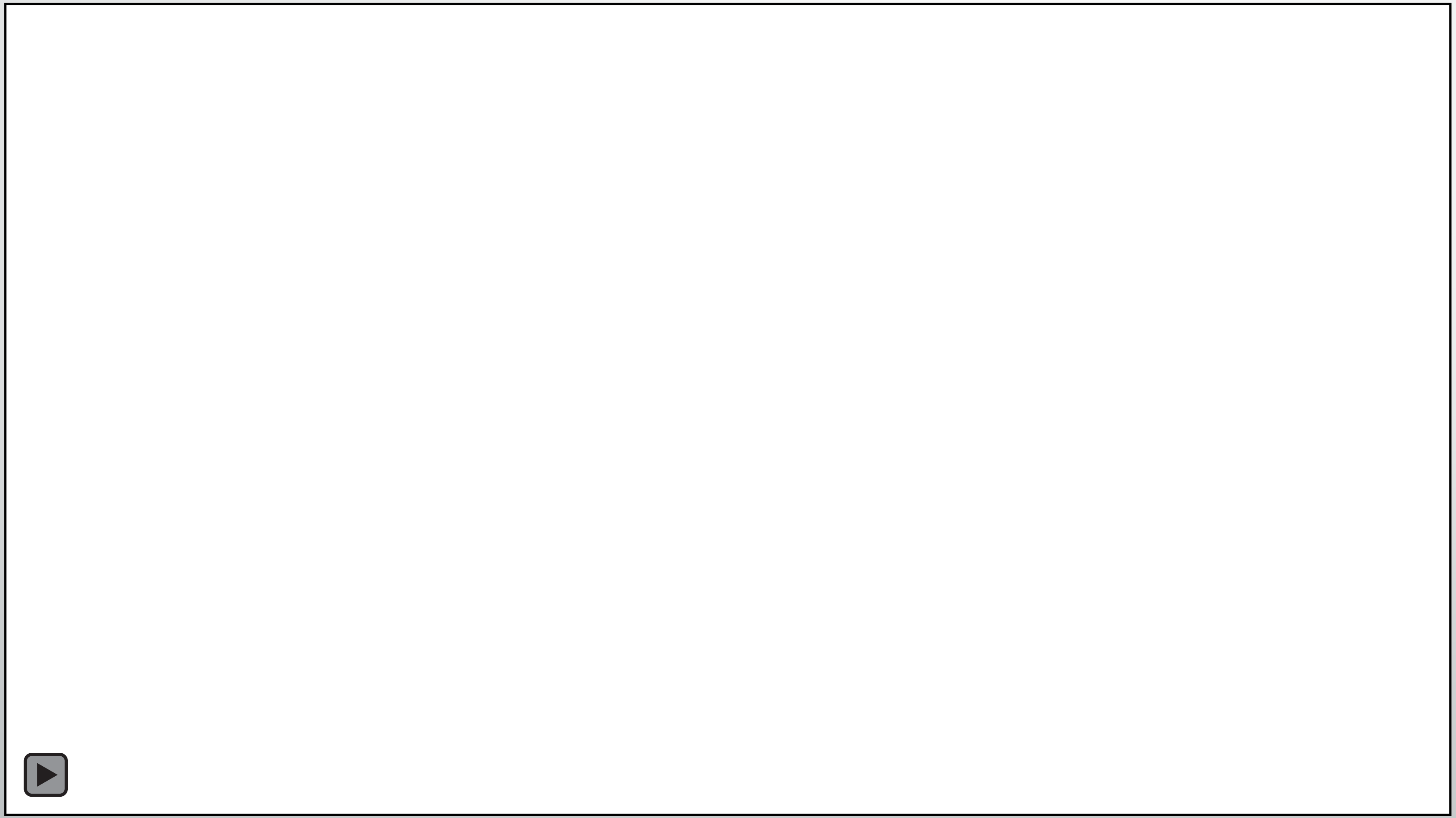
YARP

Yet Another Robot Platform

Summary

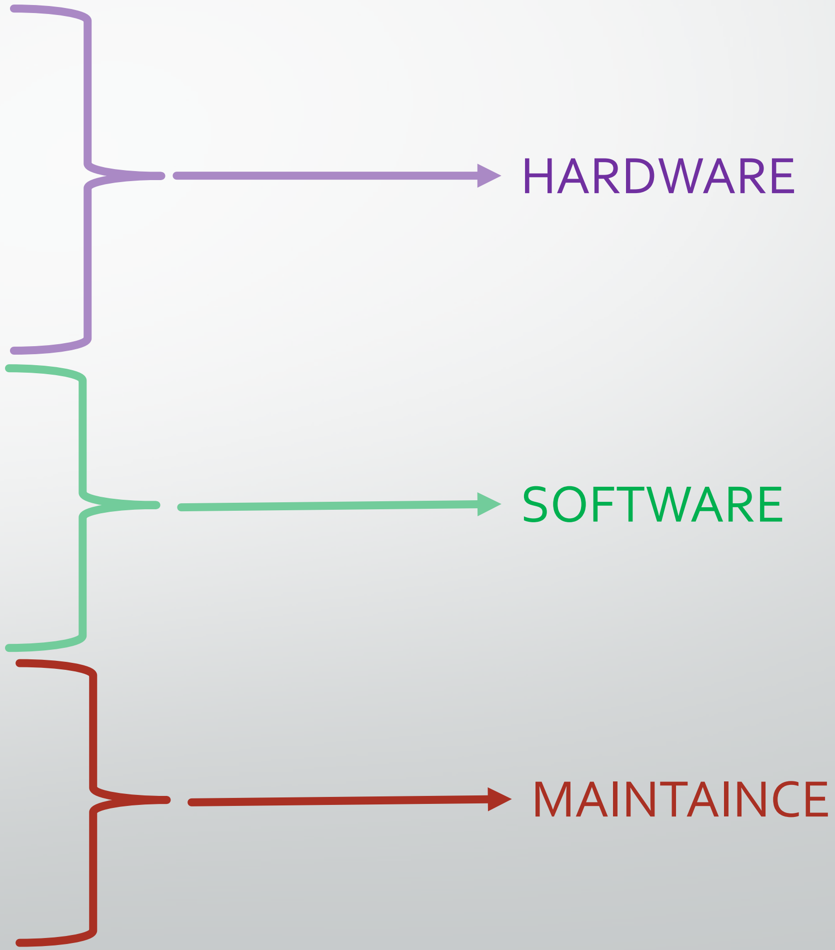
- What is YARP?
- Who uses YARP?
- How to use it?
- And ... Why?

Let’s start from the end – Why?



Why do we need a framework?

- Various scenarios and platforms
- Hardware changes in time
- Lots of different sensors
- Lack of standards
- Distributed processing
- Real-time friendly
- Algorithms/libraries/code changes in time
- Inherent complexity
- Distributed development
- Short life span of projects



HARDWARE

SOFTWARE

MAINTAINCE

What is YARP?

"If data is the bloodstream of your robot, then YARP is the circulatory system."

[Paul Fitzpatrick]

What is YARP?

“We're **not** out for world domination.”

[Paul Fitzpatrick]

What is YARP?

YARP is a middleware aimed to ease the development of high level application for robots with a strong focus on modularity, code re-usage, flexibility and hw/sw abstraction.

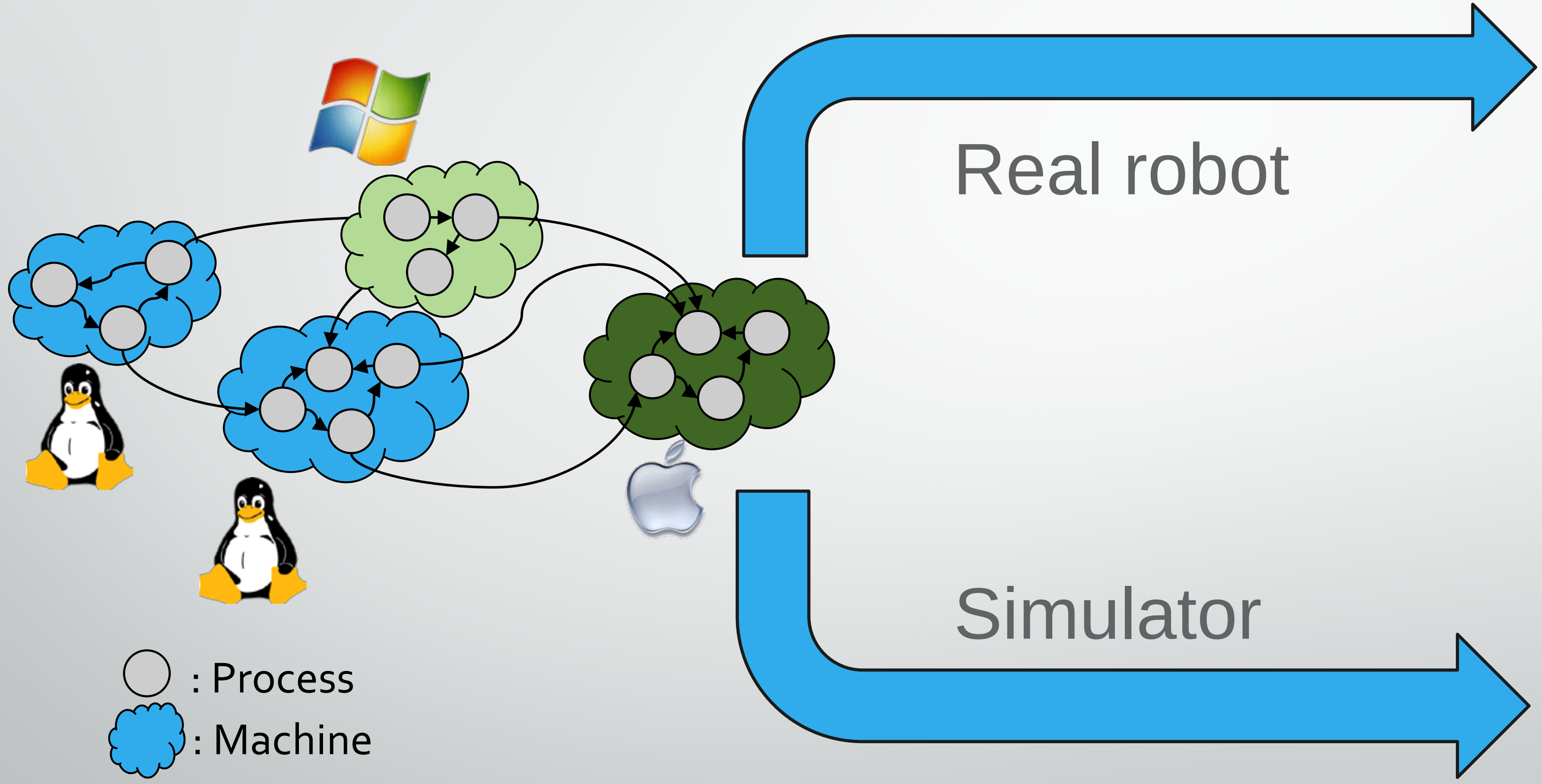
Homogeneous set of libraries, GUIs, tools, debug and run facilities

What is YARP?

YARP is a middleware aimed to ease the development of high level application for robots with a strong focus on modularity, code re-usage, flexibility and hw/sw abstraction.

YARP has been designed to support building robot control systems as collection of executables communicating in a peer-to-peer way, with an extensible types of connections (tcp, udp, multicast, local, MPI, mjpg, XML/RPC, tcpros, ...).
The strategic goal of this kind of design is to increase the longevity of robot software projects.

Typical application



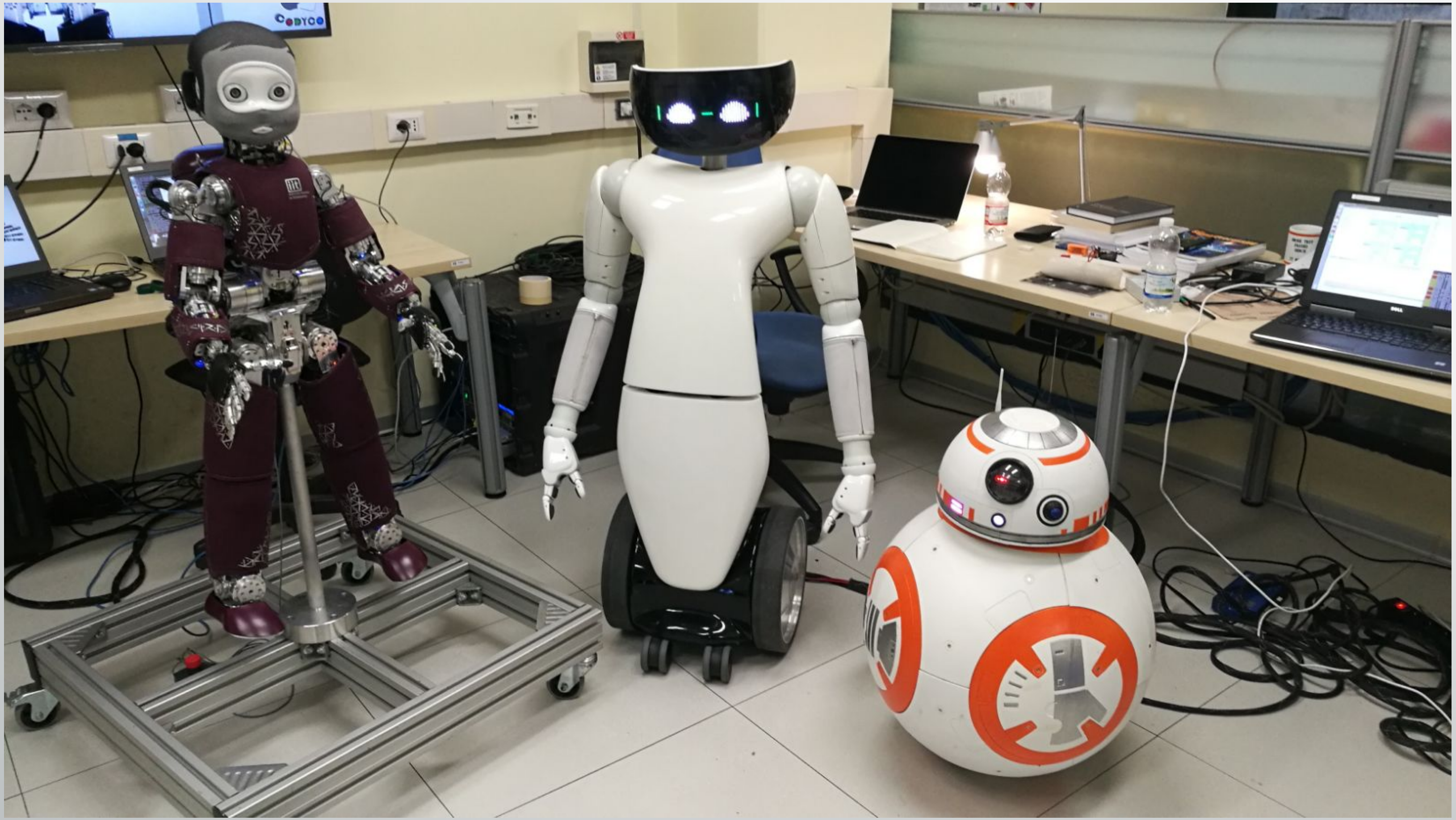
○ : Process

☁ : Machine

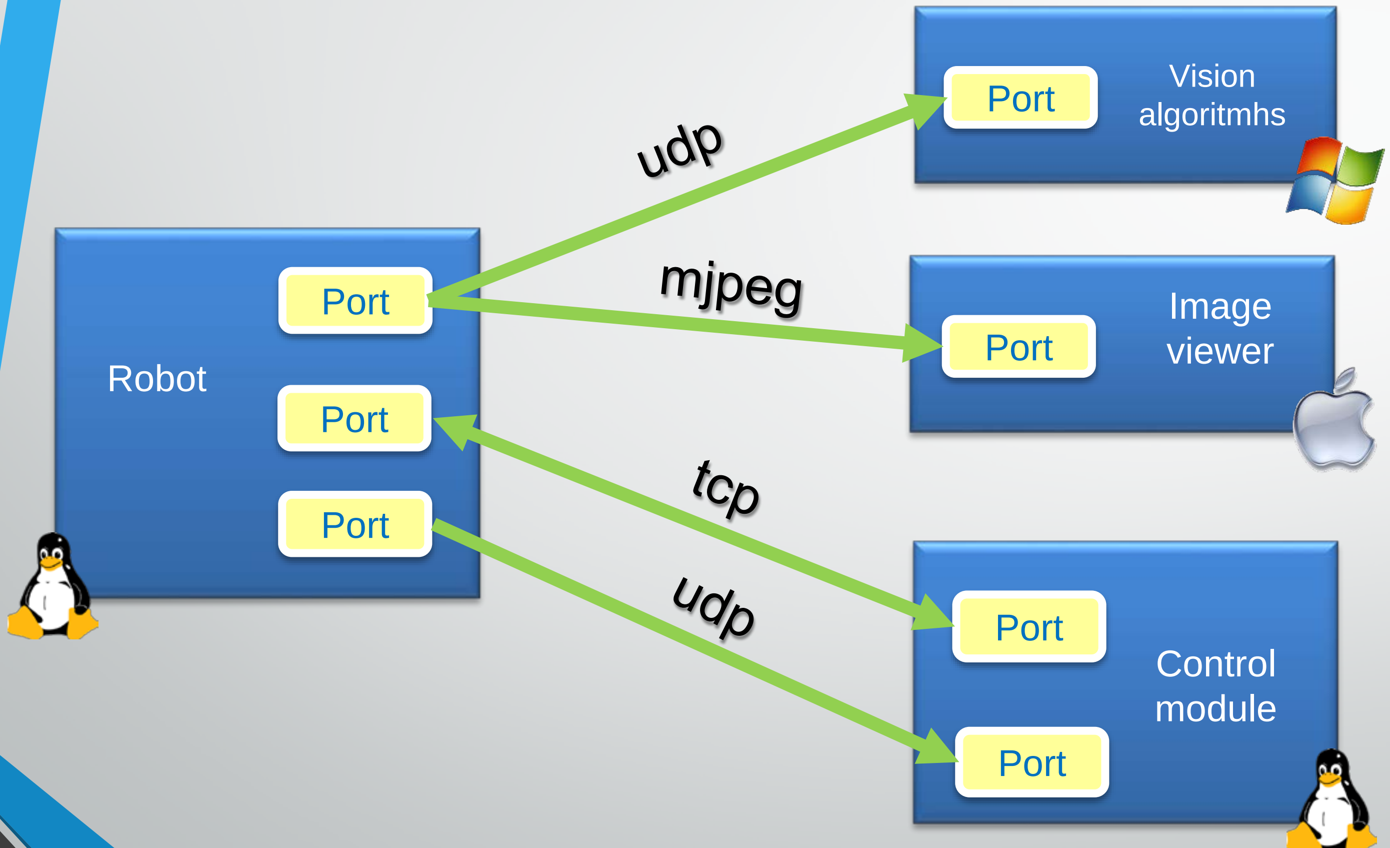
Real robot

Simulator

Who uses YARP

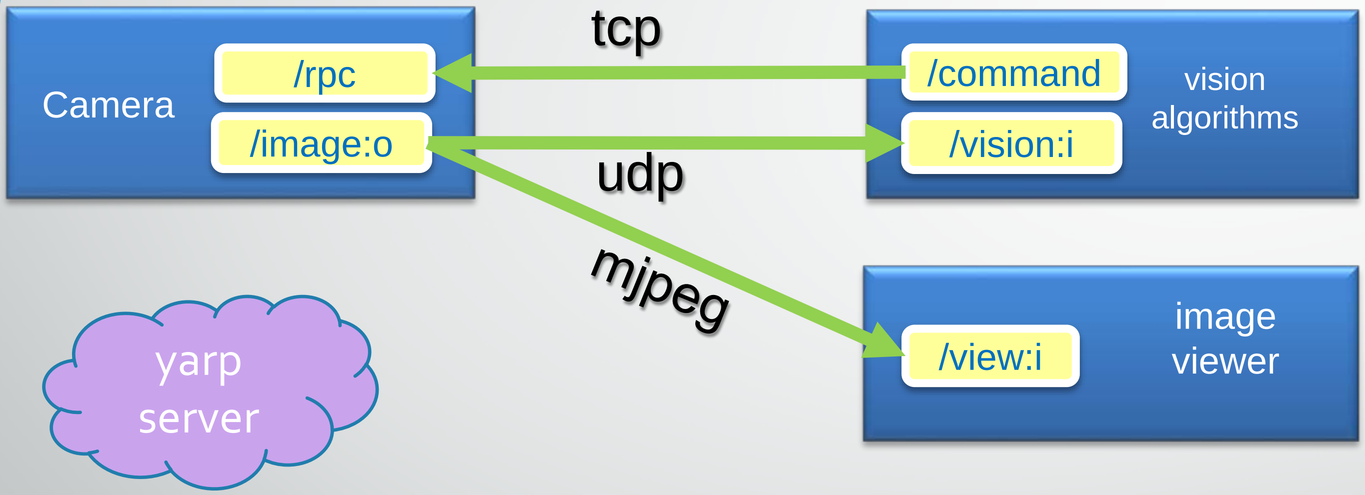


Ports: How YARP communicates



- YARP ports are the communication entry point.
- A port is a bi-directional communication entity.
- Many clients can connect to a port.
- Each connection can use different protocols or custom carrier to manipulate data on the fly.

Ports: How YARP communicates



Camera

/rpc

/image:o

tcp

udp

mjpeg

/command

/vision:i

vision algorithms

/view:i

image viewer

yarp server

YARP server acts as a DNS, resolving yarp port names into system sockets

```
$ yarp name list

/image:o    192.168.1.1:10001
/vision:i   192.168.1.2:10002
/view:i     192.168.1.3:10003
/command    192.168.1.2:10004
/rpc        192.168.1.3:10005
```

`yarp connect <source> <receiver> <carrier>(tcp)`

```
$ yarp connect  /command   /rpc
$ yarp connect  /image:o   /vision:i  udp
$ yarp connect  /image:o   /view:i    mjpeg
```

Data types

Data in YARP are Portable classes with read and write capabilities.

```
class MyData : public yarp::os::Portable
{
   // Portable interface toward YARP
   read(…);
   write(…);

   // Custom user methods for data handling
   fill_me();
   getData();

   // Usually for readability
   toString();
}
```

yarp::os::Value

Value is a container able to store in a uniform way a single instance of different basic data types.

Value can be queried to know its data type.

Data can be extracted in its native format with asXXX function.

```
class yarp::os::Value : public Portable
{
    Value(int x);                  // Create an integer data.
    Value(double x);               // Create a floating point data.
    Value(const std::string& str); // Create a string data.
    Value(void *data, int len);    // Create a binary data.

    bool isInt32();
    bool isFloat64();
    bool isString();
    bool isBlob();

    int asInt32();            // Get integer value.
    double asFloat64();       // Get floating point value.
    std::string asString();   // Get string value.
    char* asBlob();           // Get binary data value.

    …
}
```

yarp::os::Property

Dictionary type of data

```
Property prop;
prop.clear();
```

Works in pair <key, data>, where

- Key is a string

- Data is a `yarp::os::Value`

```
prop.put(“myInt”, 5);
prop.put(“myString”, “Hello World”);
prop.put(“myPi”, 3.14);
```

Entry can be grouped together, with a key

```
Property &myGroup = prop.addGroup(“group1”);
group1.put(“g1”, 2.5);
group1.put(“g2”, “We have cookies”);
```

Entry and group can be searched by the key

```
prop.check(“myInt”);
Value myInt = prop.find(“myInt”);
double myPi = prop.find(“myPi”).asFloat64();
Bottle &group = prop.findGroup(“myGroup”)
```

yarp::os::Bottle

Most flexible type of data.

```
Bottle bot;
void clear();
```

Can hold variable number of Value.

```
bot.addInt32(5);
bot.addString("hello");
```

Bottle can be appended or nested one into another.

```
Bottle& b1 = addList();
b1.addFloat64(10.2);
```

A Property can be an element of a Bottle

```
Property &prop = bot.addDict();
prop.put(“pib”, “Help me”);
```

Bottle can be accessed using indexes.
Size is the number of element you can get()

```
Value &v0 = bot.get(0);
Value &v1 = bot.get(1);
```

yarp::sig::ImageOf<PixelType>

Container for image type

Template working with many different pixel types

Full documentation here: http://www.yarp.it/classyarp_1_1sig_1_1ImageOf.html

```
ImageOf<PixelRgb> yarpImage;
yarpImage.resize(300,200);
PixelRgb  rgb;
rgb = yarpImage.pixel(10, 20);
```

Working with Ports – Client/Server

Ports are identified by their name.

Constraints:

- Names must be unique
- Names must start with '/' character
- No '@' character allowed

Ideal for client/server pattern

```
yarp::os::Port myPort;
myPort.open(“/port”);

Bottle b;
port.read(b);
int n = b.get(0).asInt32();
n++;
b.clear();
b.addInt32(n);
myPort.write(b);

myPort.close();
```

Working with Ports -- Streaming

In case of continuously broadcasted data (e.g. video streaming), a yarp::os::BufferedPort<T> can be used.

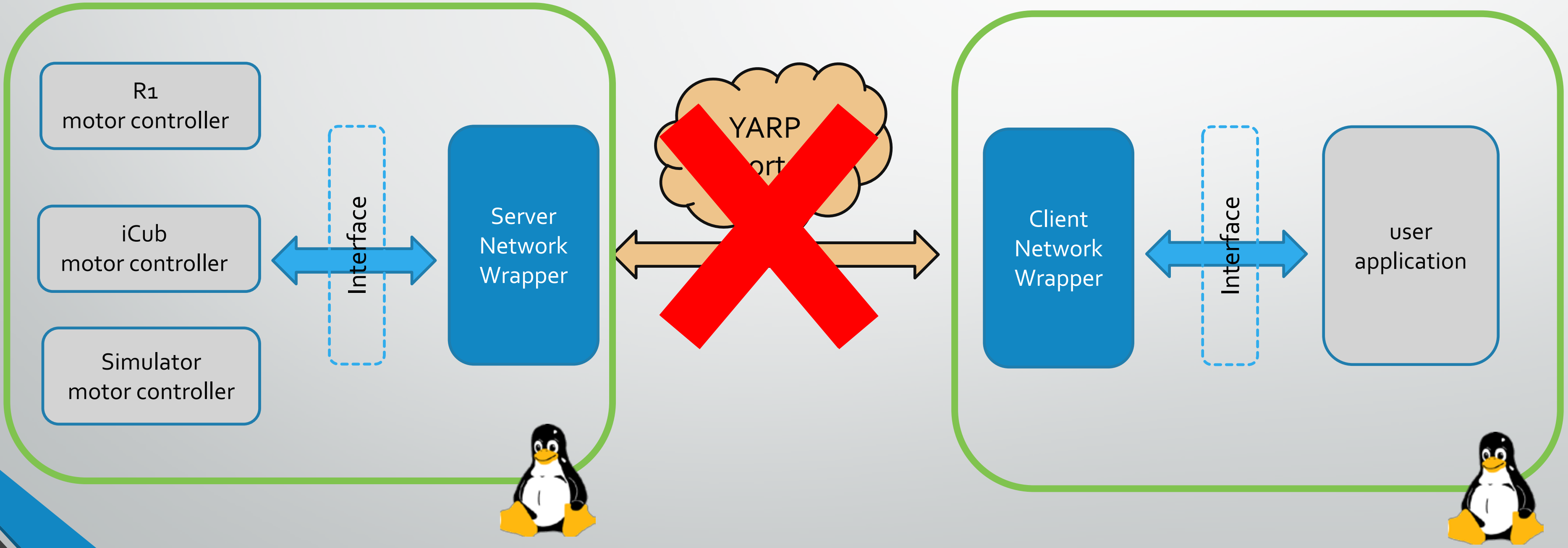
Main differences:

- Data type is fixed for port lifetime
- Memory creation/destruction is handled by the port
- Buffering policy can be set (default latest message is kept)
- A dedicated thread handles the read/write operations optimizing user thread cycle

```
BufferedPort<Bottle> port;

port.open("/out");

// Get memory to write into.
Bottle& b = port.prepare();

b.clear();

b.addString(“Hello world”);

port.write();
```

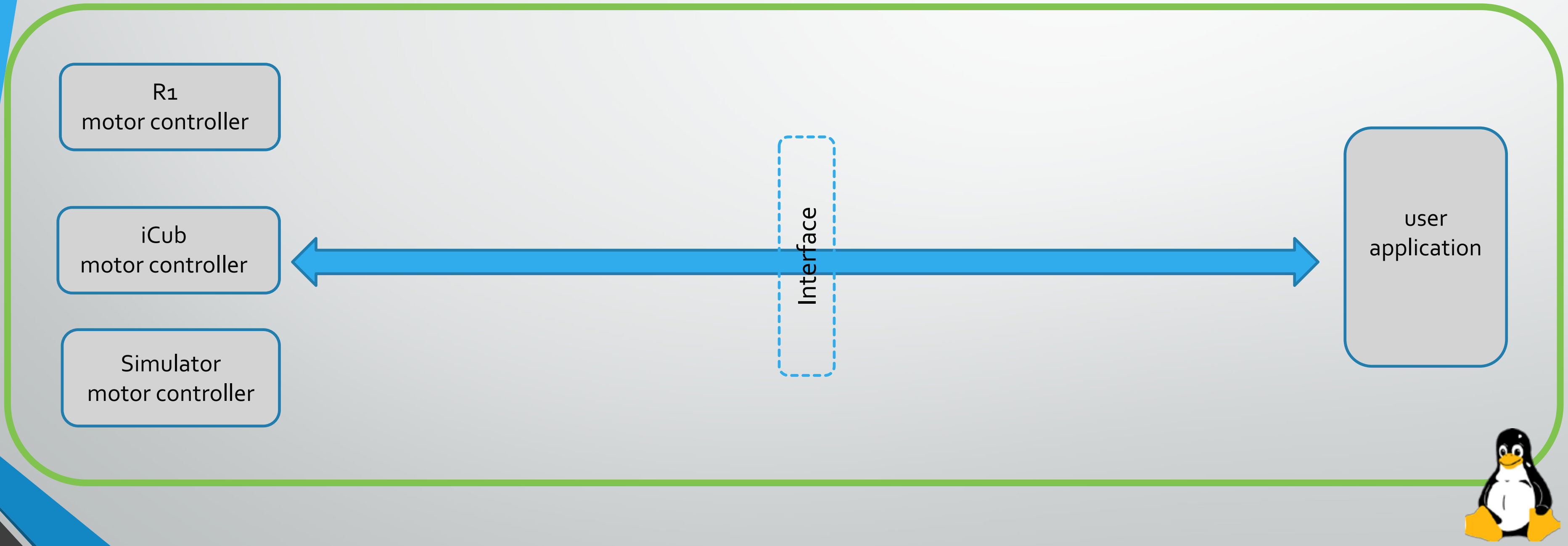
Hardware abstraction

Client & Server on the same machine



Hardware abstraction

Client & Server on the same machine



Interfaces

A class with pure virtual methods.

Servers provide functionalities by implementing required methods.

Clients use the functionalities by calling provided methods.

```
IPositionControl::getAxes() = 0;
IPositionControl::positionMove(…) = 0;
IPositionControl::relativeMove(…) = 0;
IPositionControl::checkMotionDone(…) = 0;
IPositionControl::setRefSpeed(…) = 0;
IPositionControl::setRefAcceleration(…) = 0;
IPositionControl::getRefSpeed(…) = 0;
IPositionControl::getRefAcceleration(…) = 0;
IPositionControl::getTargetPosition(…) = 0;
IPositionControl::stop(…) = 0;
```

Opening a device

Devices are opened by mean of a special class called “PolyDriver”.

PolyDriver is a polymorphic class which can turn into any device.

Keyword “device” tell YARP which device we really want to open.

All other parameters will be propagated to the specified device.

```
PolyDriver mystica;

Property config;

config.put(“device”, “device_type”);
config.put(“deviceParam1”, paramValue1);
config.put(“deviceParam2”, paramValue2);
...

mystica.open(config);
```

Remote Control Board

Device devoted to provide remote access to the robot motor control is the "remote_controlboard"

Required parameter to configure it are:

- Remote port prefix: remote

- Local port name: local

```
PolyDriver poly;

Property config;

config.put(“device”, “remote_controlboard”);
config.put(“remote”, “/icub/head”);
config.put(“local”, “/<myApplication>”);
...

poly.open(config);
```

CONTINUE →

Remote Control Board

Once opened, we need to specify which interface we want to work with.

To get a specific view of the device:

- create a pointer to the interface we want to use
- fill it by calling the .view(…) function

In case the device does not implement that interface, the pointer will be NULL!

A device can implement more than one interface.

```
IPositionControl2 *posControl = NULL;

poly.view(posControl);

if(!posControl)   // handle error
   ...

posControl->getAxes(…);
posControl->positionMove(…);

IVelocityControl2 *velControl = NULL;
poly.view(velControl);

velControl->velocityMove(…);
```

IPositionControl

Give access to main position control commands.

Used to send high level targets, with a velocity & acceleration profile.

For getters, memory must be allocated by user.

Units in YARP are SI compliant, except angles for controlboard, which are in degrees, degrees/s

IPositionControl

```
int joints;
posControl->getAxes(&joints);        // Get number of joints

posControl->setRefSpeed(0, 5);       // set a speed of 5 degrees/s for joint 0
posControl->positionMove(0, 30);     // move the joint 0 to +30 degrees

bool done = false;
do
{
    checkMotionDone(&done);          // this function checks the movement completion
}
while(!done);

posControl->positionMove(0, 0);      // reset joint position to 0
```

Other YARP features

- Resource finder
- Threads, Modules
- Plugin loader
- Carriers: mjpeg, h264, portmonitor, shared memory, depth Image, ROS
- yarpView, yarpScope, Logger, MotorGui
- YarpManager, YarpViz
- Thrift
- robotInterface

Other middleware

Cool!

“But I think ROS is better”

YARP

Ports can be typed or not

Multi-platform (also mobile)

Run-time reconfiguration of connections

Different carriers, user custom

QoS, channel prioritization

Smaller community

Rich set of libraries ad tools

Packages for all supported distributions

ROS

Both topic and service are strongly typed

Ubuntu only (ROS2 Win & Mac)

Connections from a topic use the same protocol

No concept of carrier (DDS on ROS2)

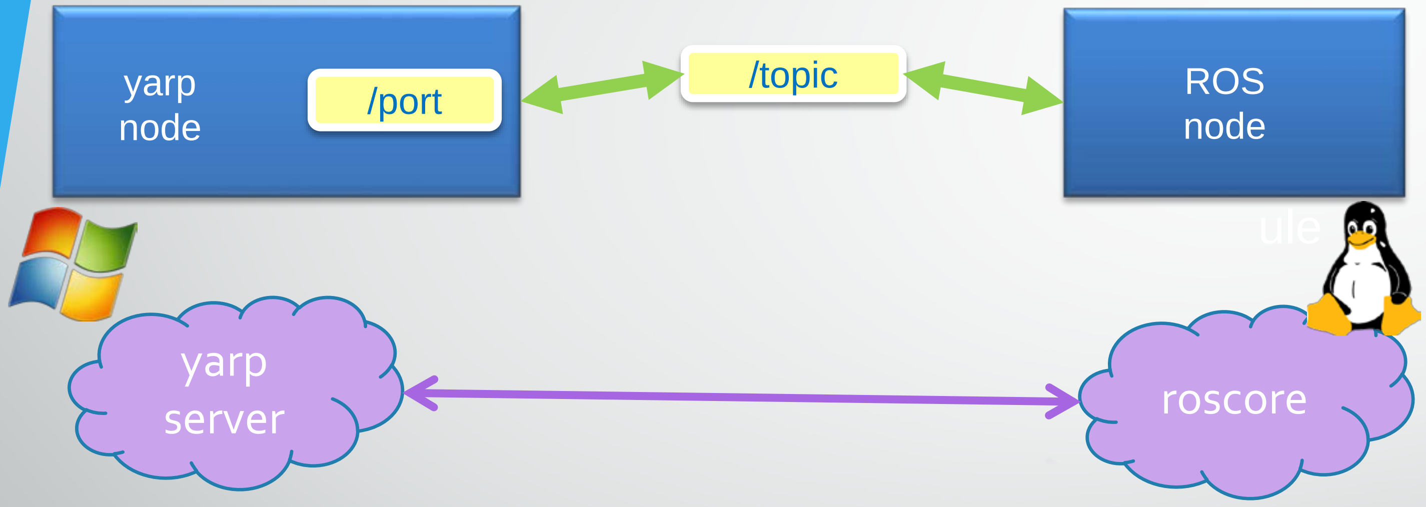
QoS on ROS2

Huge and very active community

Much more rich set of libraries and tools

Packet management

YARP - ROS carrier



YARP ask roscore to establish a new connection

YARP loads a specific carrier to convert data into ROS-like type on the fly

No need to have ROS installed

```
/topic@/yarpNode
```

THANKS FOR THE ATTENTION!