# Orocos - Yarp Binding User Guide

Charles Lesire
ONERA, Toulouse, France
`charles.lesire@onera.fr`

version 1.1

## 1 Introduction

The Orocos-Yarp binding package has been released on the RoboTIS project website of Onera at `http://robotis.onera.fr/orocos`

It has been designed to automatically export Orocos components data ports to the Yarp framework.

An extension has also been made to the DeploymentComponent to accept incoming requests from Yarp nodes. These requests are binded to Orocos State Machine execution, with the possibility to define the FSM parameters.

The package is under a LGPL licence.

## 2 Installation

The OroYarp library needs Orocos and Yarp to be installed:

- Orocos RTT version 1.10.x,
- Orocos OCL version 1.8.x,
- Orocos KDL to build KDL bindings,
- Boost program_options and date_time packages (tested with version 1.40),
- Yarp version 2.2.3 or later.

Installation is based on CMake (2.6 or later). The library will be installed to:

- `INSTALL_DIR/bin` for deployer-yarp binary,
- `INSTALL_DIR/include/robotis/oroyarp` for header files,
- `INSTALL_DIR/lib` for the binding toolkit.

`INSTALL_DIR` is by default set to `/usr/local`.

# 3 The OroYarp Toolkit

## 3.1 The YarpNode component

The Orocos-Yarp binding framework is carried by a ToolkitPlugin object. To use it, you must call `RTT::Toolkit::Import(OroYarp::YarpToolkit)` from your C++ code. Then you will be able to create a YarpNode from an existing component. A YarpNode is a TaskContext that takes data ports from its original component and binds them to Yarp. Listing 1 shows how to create a YarpNode component.

Listing 1: YarpNode creation code

```
TaskContext* yarpnode = OroYarp::YarpNode::createYarpNode(&myComp);
connectPorts(yarpnode, &myComp);
yarpnode->start();
```

The yarpnode component will now publish data to Yarp port as soon as your component updates its data port (using DataOnPortEvent call-backs), and vice versa.

## 3.2 Orocos supported types

Supported types are basic RTT types. Building the package with KDL support provides KDL types conversion. Table 1 describes Orocos-Yarp type conversion procedures.

Table 1: Orocos-Yarp type correspondence.

| Orocos type | Yarp bottle operation |
|---|---|
| i: **int** | addInt(i) |
| d: **double** | addDouble(d) |
| b: **bool** | addInt(b) |
| s: std::string | addString(s) |
| v: std::vector<**double**> | $\forall d \in v$, addDouble(d) |
| v: KDL::Vector | $\forall i = 0 \ldots 2$, addDouble(v[i]) |
| r: KDL::Rotation | $\forall i = 0 \ldots 2, j = 0 \ldots 2$, addDouble(r(i, j)) |
| f: KDL::Frame | adds rotation matrix f.M and then frame origin f.p |
| t: KDL::Twist | $\forall i = 0 \ldots 5$, addDouble(t[i]) |
| w: KDL::Wrench | $\forall i = 0 \ldots 5$, addDouble(w[i]) |

An example can be found on `tests/yarpnode_test.cpp` file.

### 3.3 The ’bottle’ type

The toolkit also defines a ’bottle’ type usable by Orocos components and scripts. A ’bottle’ is the Orocos type name for a Yarp Bottle. Orocos components can then directly export Yarp Bottles understandable by any Yarp node. An example can be found on `tests/yarpbottle_test.cpp` file.

## 4 The deployer-yarp

The OroYarp library provides a Yarp version of the Deployer, namely the `deployer-yarp` executable. It works as a classical OCL deployer, and provides specific Yarp functionalities.

### 4.1 The `yarpnode` method

A `yarpnode` method allows to build a YarpNode corresponding to a loaded component. It proceeds exactly the code of listing 1 with some extra test cases.

The deployer can export a component to Yarp at startup. This is requested by setting the component server flag to true in the deployer configuration file. Listing 2 shows an extract of the corresponding xml code.

Listing 2: Deployer configuration file

```
<struct name=”MyComponent” type=”MyComponentClass”>
  <simple name=”Server” type=”boolean”><value>1</value></simple>
  <!-- -->
</struct>
```

### 4.2 Scripting and Requests

The deployer opens two Yarp ports: one for scripting from Yarp, the other for some kind of ”request management”.

#### 4.2.1 Scripting from Yarp

The deployer scripting port is named ”/DeployerName/Scripting”. The incoming string data are interpreted as script expression and are then directly executed using the deployer `scripting()->execute` function.

#### 4.2.2 Executing FSM from Yarp

The deployer request port is named ”/DeployerName/Request”. When something is received on this port, it is parsed and proceeded as a request to launch a State Machine. The input format must be a string carried by a Yarp bottle. This string must match the following regular expression:

$$[A - Za - z0 - 9\_]^{+}\ [\,[\backslash s]\ [\hat{}\backslash s]^{+}\,]^{*}$$

The first word is interpreted as the FSM name. The following words (separated by spaces) are interpreted as FSM parameters. Each parameter is updated using its TypeInfo `read` function, defined in the TemplateTypeInfo structure imported in the Orocos toolkit.

FSM must follow a specific design pattern to be runnable from Yarp nodes:

- The initial state of the FSM must be empty, as this state is used to set parameters values (the FSM is first activated, then parameters are modified, and the FSM is finally started);
- An instance of the FSM must be created with default parameter values.

Only not running FSM can be launched from Yarp. Stopping a running FSM is not performed by the deployer processing. An example can be found on `tests/yarpscripting*` files.

# 5 Extending the Toolkit

The OroYarp toolkit can be extended to support your own Orocos-Yarp type conversion. To map a type from Orocos to Yarp, you must add conversion functions extending the YarpTypeConversion structure. Listing 3 gives an example of a YarpTypeConversion code.

Listing 3: YarpTypeConversion structure for OrocosT

```
template <>
struct StdYarpTypeConversion<OrocosT> :
  public YarpTypeConversion<OrocosT, YarpT> {
    static bool copyYarpToOrocos(const YarpT& y, OrocosT& o) {
      // The conversion code, e.g. o = y.asDouble();
      return true;
    };
    static bool copyOrocosToYarp(const OrocosT& o, YarpT& y) {
      // The conversion code, e.g. y.addDouble(o);
      return true;
    };
};
```

Then you must define your own toolkit that will register this type conversion to OroYarp. It could be done by the code of listing 4.

See Orocos' Extending Real-Time Toolkit web page for more detail on creating your own toolkit.

Listing 4: OroYarp Toolkit registering OrocosT conversion

```
class MyYarpToolkitPlugin : public OroYarp::YarpToolkitPlugin {
  public:
    virtual std::string getName() { return "MyYarpToolkit"; };
    virtual bool loadTypes() {
      OroYarp::YarpPortCreator::Instance()
        ->registerType<OrocosT>("myOrocosType");
      return YarpToolkitPlugin::loadTypes();
    };
};
```

## 6 Controlling the Yarp Fakebot from on Orocos Controller

The OroYarp package provides an example of interaction between Orocos and Yarp components. It uses the Yarp Fakebot tutorial, available in the Yarp directory in `example/tutorial`. The sources are also included (and compiled) in the OroYarp package. Figure 1 shows the complete example setup: the Fakebot is embedded in a Yarp node, the Controller is executed by the Orocos deployer, and the OroYarp package provides a Yarp node interface to connect the controller and the bot.

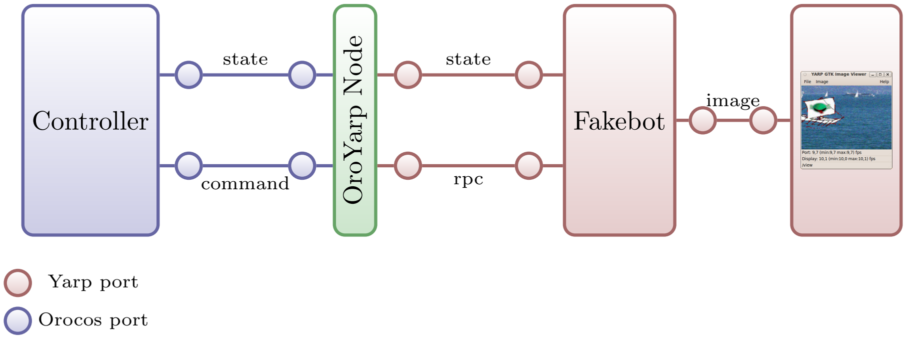


Figure 1: The Fakebot example setup.

The Controller reads the fakebot state (a 2D position vector) and sends a goal position for its current `control_axis`. Its behavior is described in algorithm 1.

To launch the example, you must execute the following command lines in separate terminals:

1. `yarp server` : starts the Yarp naming service;
2. `yarpview` : starts the Yarp image viewer;

**Algorithm 1** Controller behavior

```
loop
  if $||state - current\_goal|| < delta$ then
    $new\_goal \leftarrow rand()$
    send command "set pos $control\_axis$ $new\_goal$"
  end if
end loop
```

3. `robotis-yarp-fakebot --file fakebot.ini` : starts the Yarp fakebot; must be launched in directory `example` (where file `fakebot.ini` is);
4. `robotis-yarp-controller --auto` : starts the Orocos controller.

The `robotis-yarp-controller` command has the following options:

`--help` displays the allowed options;

`--log-level Warning` set the RTT log level (default is 'Warning', 'Info' displays controller information);

`--view /view` set the YarpView port name (used if auto-connect is on – default is '/yarpview/img:i');

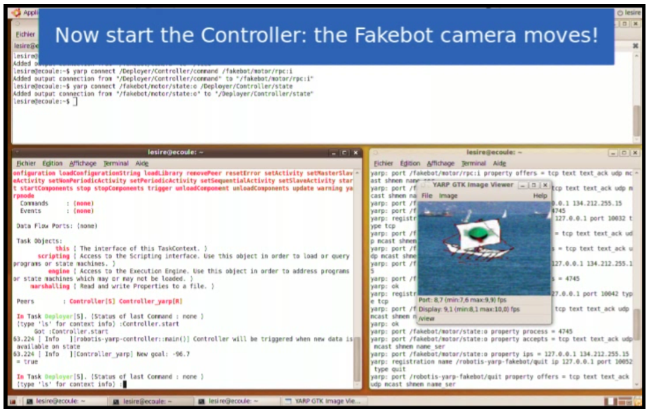
`--auto` set auto-connect on: creates the controller Yarp node and connects Yarp ports.

If the controller is launched without the `--auto` option, the Controller Yarp node must be created (`yarpnode("Controller")` in the deployer) and the following port connections must be set by hand:

- `yarp connect /fakebot/camera /yarpview/img:i`
- `yarp connect /fakebot/motor/state:o /Deployer/Controller/state`
- `yarp connect /Deployer/Controller/command /fakebot/motor/rpc:i`

Then the controller component must be started (`Controller.start` in the deployer).

The view displays a background image on which a boat is moving. The controller component gives goal position to the Fakebot, making its camera move on one axis. When the goal has been reached, the controller draws randomly a new goal and sends it again to the Fakebot. The example is described in a movie (figure 2) available on the RoboTIS website.


Now start the Controller: the Fakebot camera moves!

```
lesire@ecoule:~$ yarp connect /Deployer/Controller/command /fakebot/motor/rpc:i
Added output connection from "/Deployer/Controller/command" to "/fakebot/motor/rpc:i"
lesire@ecoule:~$ yarp connect /fakebot/motor/state:o /Deployer/Controller/state
Added output connection from "/fakebot/motor/state:o" to "/Deployer/Controller/state"
lesire@ecoule:~$ 
```

```
onfiguration loadConfigurationString loadLibrary removePeer resetError setActivity setMasterSlav
eActivity setNonPeriodicActivity setPeriodicActivity setSequentialActivity setSlaveActivity star
t startComponents stop stopComponents trigger unloadComponent unloadComponents update warning ya
rpnode
  Commands     : (none)
  Events       : (none)

 Data Flow Ports: (none)

 Task Objects:
           this ( The interface of this TaskContext. )
      scripting ( Access to the Scripting interface. Use this object in order to load or query
programs or state machines. )
         engine ( Access to the Execution Engine. Use this object in order to address programs
or state machines which may or may not be loaded. )
    marshalling ( Read and write Properties to a file. )

 Peers        : Controller[S] Controller_yarp[R]

 In Task Deployer[S]. (Status of last Command : none )
 (type 'ls' for context info) :Controller.start
      Got :Controller.start
63.224 [ Info   ][robotis-yarp-controller::main()] Controller will be triggered when new data is
 available on state
63.224 [ Info   ][Controller_yarp] New goal: -96.7
 = true

 In Task Deployer[S]. (Status of last Command : none )
 (type 'ls' for context info) :
```

```
yarp: port /fakebot/motor/rpc:i property offers = tcp text text_ack udp mc
ast shmem name_ser
yarp: port /f                              tcp text text_ack udp m
cast shmem na
yarp: port /f                              0.0.1 134.212.255.15
yarp: port /f                              4745
yarp: registr                              127.0.0.1 port 10032 t
ype tcp
yarp: port /f                              = tcp text text_ack ud
p mcast shmem
yarp: port /f                              s = tcp text text_ack u
dp mcast shme
yarp: port /f                              127.0.0.1 134.212.255.1
5
yarp: port /f                              s = 4745
yarp: ok
yarp: registr                              27.0.0.1 port 10042 typ
e tcp
yarp: port /f                              tcp text text_ack udp
mcast shmem n
yarp: ok
yarp: port /fakebot/motor/state:o property process = 4745
yarp: port /fakebot/motor/state:o property accepts = tcp text text_ack udp
 mcast shmem name_ser
yarp: port /fakebot/motor/state:o property ips = 127.0.0.1 134.212.255.15
yarp: registration name /robotis-yarp-fakebot/quit ip 127.0.0.1 port 10052
 type quit
yarp: port /robotis-yarp-fakebot/quit property offers = tcp text text_ack
udp mcast shmem name_ser
```

```
Port: 8,7 (min:7,6 max:9,9) fps
Display: 9,1 (min:8,1 max:10,0) fps
/view
```

Figure 2: Example movie screenshot.