Tutorial on the Component Interface

Open RObot COntrol Software

2.9.0

Tutorial on the Component Interface : *Open RObot COntrol Software* : 2.9.0

Copyright © 2002,2003,2004,2005,2006 Peter Soetens, FMTC

Abstract

This document gives an introduction on the different aspects of interfacing an Orocos component. This is an excerpt from the Component Builder's Manual and walks through the 'Hello World' program.

Orocos Version 2.9.0.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation, with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of this license can be found at http://www.fsf.org/copyleft/fdl.html.

Table of Contents

1. 1
    1. Introduction 1
    2. Hello World ! 3
        2.1. Using the Deployer 3
        2.2. Starting your First Application 4
        2.3. Displaying a TaskContext 6
        2.4. Listing the Interface 8
        2.5. Calling an Operation 8
        2.6. Sending a Operation 8
        2.7. Changing Values 9
        2.8. Reading and Writing Ports 9
        2.9. Last Words 9

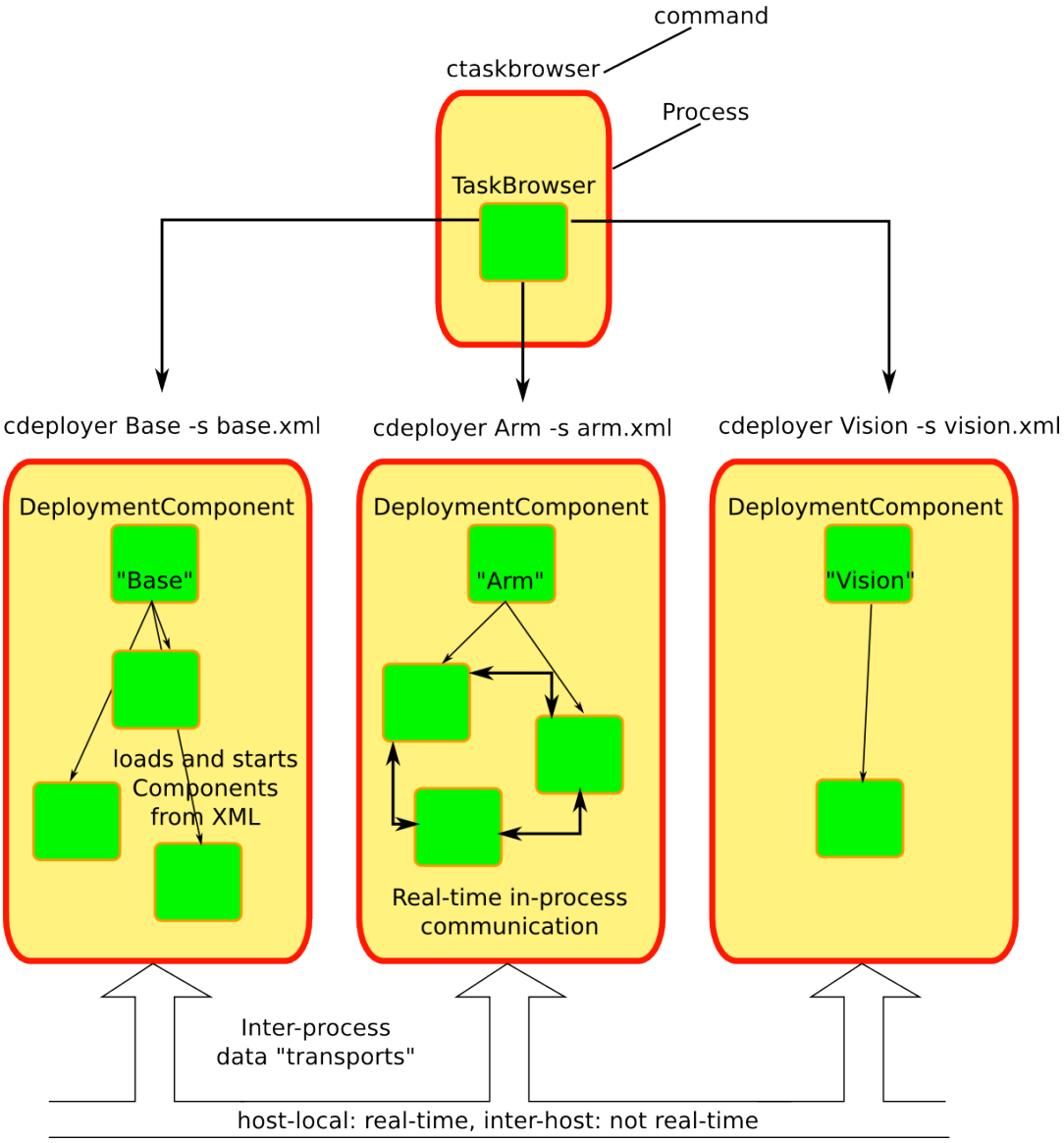
List of Figures

1.1. Typical application example for distributed control 2
1.2. Dynamic vs static loading of components 5
1.3. Schematic Overview of the Hello Component. 7

Chapter 1.

1. Introduction

This manual documents how multi-threaded components can be defined in Orocos such that they form a thread-safe robotics/machine control application. Each control component is defined as a "TaskContext", which defines the environment or "context" in which an application specific task is executed. The context is described by the three Orocos primitives: Operation, Property, and Data Port. This document defines how a user can write his own task context and how it can be used in an application.



Components are loaded into the process by a deployer, which gets its configuration through an XML file. Communication between processes is transparant to the component, but your data must be known to Orocos (cfr 'typekits' and 'transports'). Most new users start with a single process however, using the 'deployer' application.

Figure 1.1. Typical application example for distributed control

A component is a basic unit of functionality which executes one or more (real-time) programs in a single thread. The program can vary from a mere C/C++ function over a real-time program script to a real-time hierarchical state machine. The focus is completely on thread-safe time determinism. Meaning, that the system is free of priority-inversions, and all operations are lock-free. Real-time components can communicate with non real-time components (and vice verse) transparently.

Note

In this manual, the words task and component are used as equal words, meaning a software component built using the C++ TaskContext class.

The Orocos Component Model enables :

- Lock free, thread-safe, inter-component communication in a single process.
- Thread-safe, inter-process communication between (distributed) processes.
- Communication between hard Real-Time and non Real-Time components.
- Deterministic execution time during communication for the higher priority thread.
- Synchronous and asynchronous communication between components.
- Interfaces for run-time component introspection.
- C++ class implementations and scripting interface for all the above.

The Scripting chapter gives more details about script syntax for state machines and programs.

2. Hello World !

Important

Before you proceed, make sure you printed the Orocos Cheat Sheet [http://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/orocos_cheat_sheet.pdf] and RTT Cheat Sheet [http://www.orocos.org/stable/documentation/rtt/v2.x/doc-xml/rtt_cheat_sheet.pdf] ! They will definately guide you through this lengthy text.

This section introduces tasks through the "hello world" application, for which you will create a component package using the **orocreate-pkg** command on the command line:

```
$ rosrun ocl orocreate-pkg HelloWorld    # ... for ROS users
```

```
$ orocreate-pkg HelloWorld    # ... for non-ROS users
```

In a properly configured installation, you'll be able to enter this directory and build your package right away:

```
$ cd HelloWorld
$ make
```

In case you are *not* using ROS to manage your packages, you also need to install your package:

```
$ make install
```

2.1. Using the Deployer

The way we interact with TaskContexts during development of an Orocos application is through the *deployer* . This application consists of the DeploymentComponent which is responsible for creating applications out of component libraries and the DeploymentComponent which is a powerful console tool which helps you to explore, execute and debug componentss in running programs.

The TaskBrowser uses the GNU readline library to easily enter commands to the tasks in your system. This means you can press TAB to complete your commands or press the up arrow to scroll through previous commands.

You can start the deployer in any directory like this:

```
$ deployer-gnulinux
```

or in a ROS environment:

```
$ rosrun ocl deployer-gnulinux
```

This is going to be your primary tool to explore the Orocos component model so get your seatbelts fastened!

2.2. Starting your First Application

Now let's start the HelloWorld application we just created with **orocreate-pkg**.

Create an 'helloworld.ops' Orocos Program Script (ops) file with these contents:

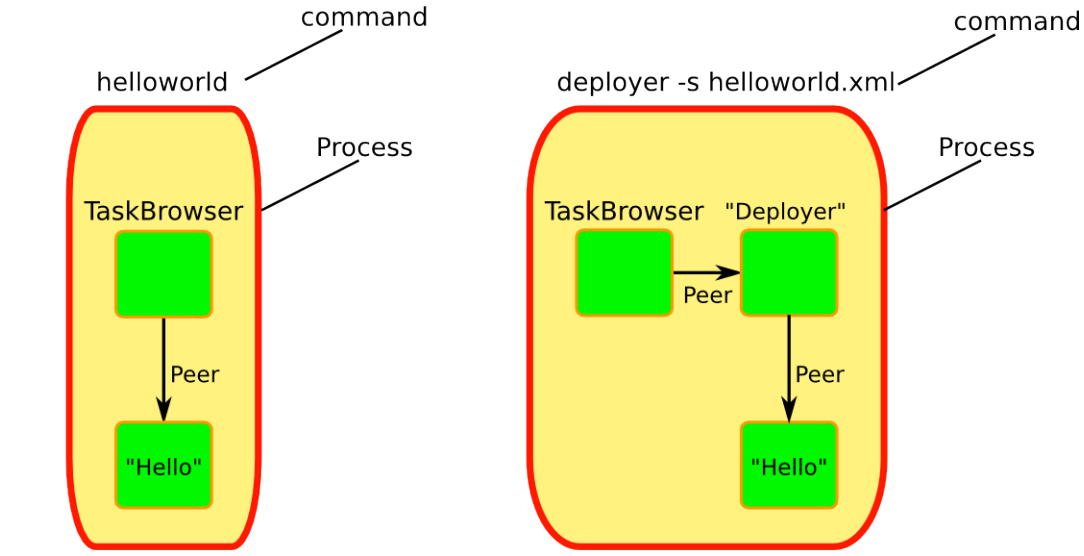
```
require("print")        // necessary for 'print.ln'
import("HelloWorld")    // 'HelloWorld' is a directory name to import

print.ln("Script imported HelloWorld package:")
displayComponentTypes() // Function of the DeploymentComponent

loadComponent("Hello", "HelloWorld") // Creates a new component of type 'HelloWorld'
print.ln("Script created Hello Component with period: " + Hello.getPeriod() )
```

and load it into the deployer using this command: **$ deployer-gnulinux -s helloworld.ops -linfo** This command imports the HelloWorld package and any component library in there. Then it creates a component with name "Hello". We call this a dynamic deployment, since the decision to create components is done at run-time.

You could also create your component in a C++ program. We call this static deployment, since the components are fixed at compilation time. The figure below illustrates this difference:



Static deployment:
The component is created in
the ORO_main() function

Dynamic deployment:
The component is loaded
by the deployer

The 'helloworld' executable is a static deployment of one component in a process, which means it is hard-coded in the helloworld.cpp file. In contrast, using the deployer application allows you to load a component library dynamically.

Figure 1.2. Dynamic vs static loading of components

The output of the deployer should be similar to what we show below. Finally, type **cd Hello** to start with the exercise.

```
0.000 [ Info   ][Logger] Real-time memory: 14096 bytes free of 20480 allocated.
0.000 [ Info   ][Logger] No RTT_COMPONENT_PATH set. Using default: .../rtt/install/lib/orocos
0.000 [ Info   ][Logger] plugin 'rtt' not loaded before.

...

0.046 [ Info   ][Logger] Loading Service or Plugin scripting in TaskContext Deployer
0.047 [ Info   ][Logger] Found complete interface of requested service 'scripting'
0.047 [ Info   ][Logger] Running Script helloworld.ops ...
0.050 [ Info   ][DeploymentComponent::import] Importing directory .../HelloWorld/lib/orocos/
gnulinux ...
0.050 [ Info   ][DeploymentComponent::import] Loaded component type 'HelloWorld'
Script imported HelloWorld package:
I can create the following component types:
  HelloWorld
  OCL::ConsoleReporting
  OCL::FileReporting
  OCL::HMIConsoleOutput
  OCL::HelloWorld
  OCL::TcpReporting
  OCL::TimerComponent
  OCL::logging::Appender
  OCL::logging::FileAppender
  OCL::logging::LoggingService
  OCL::logging::OstreamAppender
  TaskContext
```

```
0.052 [ Info   ][Thread] Creating Thread for scheduler: 0
0.052 [ Info   ][Hello] Thread created with scheduler type '0', priority 0, cpu affinity 15 and period 0.
HelloWorld constructed !
0.052 [ Info   ][DeploymentComponent::loadComponent] Adding Hello as new peer:  OK.
Script created Hello Component with period: 0
0.053 [ Info   ][Thread] Creating Thread for scheduler: 0
0.053 [ Info   ][TaskBrowser] Thread created with scheduler type '0', priority 0, cpu affinity 15 and
 period 0.
   Switched to : Deployer
0.053 [ Info   ][Logger] Entering Task Deployer

  This console reader allows you to browse and manipulate TaskContexts.
  You can type in an operation, expression, create or change variables.
  (type 'help' for instructions and 'ls' for context info)
    TAB completion and HISTORY is available ('bash' like)

Deployer [S]> cd Hello
   Switched to : Hello
Hello [S]>
```

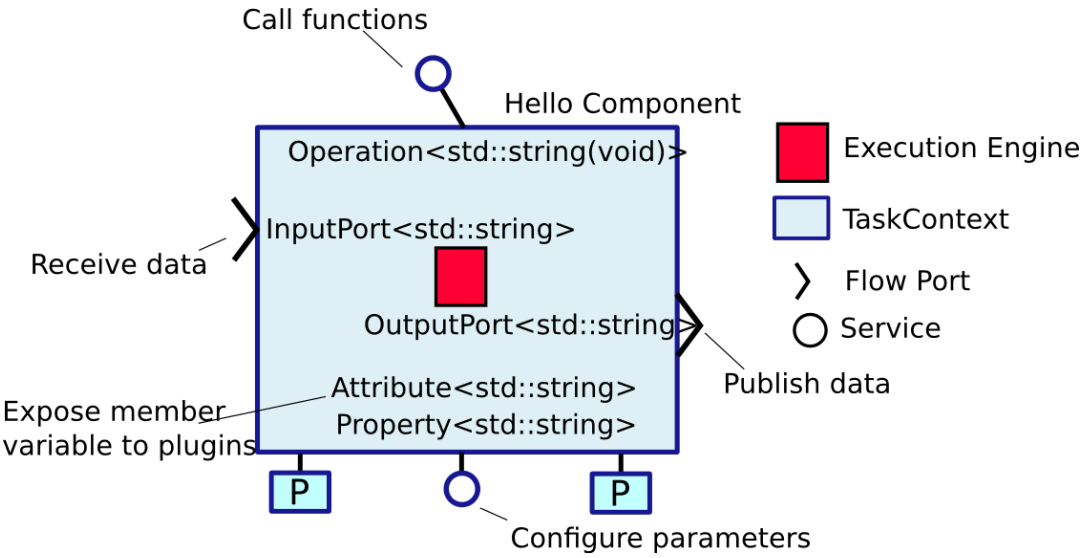
The first **[Info]** lines are printed by the Orocos Logger, which has been configured to display informative messages to console with the **-linfo** program option. Normally, only warnings or worse are displayed by Orocos. You can always watch the log file 'orocos.log' in the same directory to see all messages. After the **[Log Level]**, the **[Origin]** of the message is printed, and finally the message itself. These messages leave a trace of what was going on in the main() function before the prompt appeared.

Depending on what you type, the TaskBrowser will act differently. The built-in commands **cd**, **help**, **quit**, **ls** etc, are seen as commands to the TaskBrowser itself, if you typed something else, it tries to execute your command according to the Orocos scripting language syntax.

```
Hello[R] > 1+1
  = 2
```

2.3. Displaying a TaskContext

A component's interface consists of: Attributes and Properties, Operations, and Data Flow ports which are all public. The class TaskContext groups all these interfaces and serves as the basic building block of applications. A component developer 'builds' these interfaces using the instructions found in this manual.



Our hello world component.

Figure 1.3. Schematic Overview of the Hello Component.

To display the contents of the current component, type **ls**, and switch to one of the listed peers with **cd**, while **cd ..** takes you one peer back in history. We have two peers here: the Deployer and your component, Hello.

```
Hello [S]> ls

 Listing TaskContext Hello[S] :

 Configuration Properties: (none)

 Provided Interface:
  Attributes   : (none)
  Operations   : activate cleanup configure error getCpuAffinity getPeriod inFatalError
inRunTimeError isActive isConfigured isRunning setCpuAffinity setPeriod start stop trigger
update

 Data Flow Ports: (none)

 Services:
(none)

 Requires Operations :  (none)
 Requests Services   :  (none)

 Peers       : (none)
Hello [S]>
```

Note

To get a quick overview of the commands, type **help**.

The first line shows the status between square brackets. The [S] here means that the component is in the stopped state. Other states can be 'R' - Running, 'U' - Unconfigured, 'E' - runtime Error, 'F' - Fatal error, 'X' - C++ eXception in user code.

First you get a list of the Properties and Attributes (alphabetical) of the current component. Properties are meant for configuration and can be written to disk. Attributes export a C++ class value to the interface, to be usable by scripts or for debugging and are not persistent.

Next, the operations of this component are listed: each component has some universal functions like activate, start, getPeriod etc.

You can see that the component is pretty empty: no data flow ports, services or peers. We will add some of these right away.

2.4. Listing the Interface

To get an overview of the Task's interface, you can use the help command, for example *help this* or *help this.activate* or just short: *help activate*

```
Hello [R]> help this

Printing Interface of 'Hello' :

 activate( ) : bool
   Activate the Execution Engine of this TaskContext (= events and commands).
 cleanup( ) : bool
   Reset this TaskContext to the PreOperational state (write properties etc).
...
   Stop the Execution Engine of this TaskContext.

Hello [R]> help getPeriod
 getPeriod( ) : double
Get the configured execution period. -1.0: no thread associated, 0.0: non periodic, > 0.0: the period.

Hello [R]>
```

Now we get more details about the operations registered in the public interface. We see now that the *getPeriod* operations takes no arguments You can invoke each operation right away.

2.5. Calling an Operation

```
Hello [R]> getPeriod()
 = 0
```

Operations are called directly and the TaskBrowser prints the result. The return value of getPeriod() was a double, which is 0. This works just like calling a 'C' function. You can express calling explicitly by writing: **getPeriod.call()**.

2.6. Sending a Operation

When an operation is *sent* to the Hello component, another thread will execute it on behalf of the sender. Each sent method returns a SendHandle object.

```
Hello [R]> getPeriod.send()
 = (unknown_t)
```

The returned SendHandle must be stored in a SendHandle attribute to be useful:

```
Hello [R]> var SendHandle sh
Hello [R]> sh = getPeriod.send()
 = true
Hello [R]> sh.collectIfDone( ret )
 = SendSuccess
Hello [R]> ret
 = 0
```

SendHandles are further explained down the document. They are not required understanding for a first discovery of the Orocos world.

2.7. Changing Values

Besides calling or sending component methods, you can alter the attributes of any task, program or state machine. The TaskBrowser will confirm validity of the assignment with the contents of the variable. Since Hello doesn't have any attributes, we create one dynamically:

```
Hello [R]> var string the_attribute = "HelloWorld"
Hello [R]> the_attribute
 = Hello World
Hello [R]> the_attribute = "Veni Vidi Vici !"
 = "Veni Vidi Vici !"
Hello [R]> the_attribute
 = Veni Vidi Vici !
```

2.8. Reading and Writing Ports

The Data Ports allow seamless communication of calculation or measurement results between components. Adding and using ports is described in ???.

2.9. Last Words

Last but not least, hitting TAB twice, will show you a list of possible completions, such as peers, services or methods.

TAB completion works even across peers, such that you can type a TAB completed command to another peer than the current peer.

In order to quit the TaskBrowser, enter **quit**:

```
 Hello [R]> quit

1575.720 [ Info    ][ExecutionEngine::setActivity] Hello is disconnected from its activity.
1575.741 [ Info    ][Logger] Orocos Logging Deactivated.
```

The TaskBrowser Component is application independent, so that your end user-application might need a more suitable interface. However, for testing and inspecting what is happening

inside your real-time programs, it is a very useful tool. The next sections show how you can add properties, methods etc to a TaskContext.

Note

If you want a more in-depth tutorial, see the rtt-exercises package which covers each aspect also shown in this manual.