Distributing Orocos Components with CORBA

The CORBA Transport Library

Revision History

Revision 0.01	4 May 2006	ps
Initial version		
Revision 0.02	24 August 2006	ps
Update to new Orocos interfaces		
Revision 0.03	9 November 2006	ps
1.0.0 release updates		
Revision 0.04	3 August 2007	ps
1.2.2 release updates, added corbaloc options.		
Revision 0.05	9 April 2009	ps
Put proxy server as mandatory.		
Revision 0.06	2 November 2009	ps
Renamed to orocos-transports-corba.xml and minor additions.		
Revision 0.07	1 April 2010	ps
Document new CORBA Transport API (RTT 2.0)		
Revision 1.0	24 June 2011	ps
Provide basic instructions for setting up corba deployments.		

Abstract

This document explains the principles of the *Corba Transport* of Orocos, the *Open RObot COntrol Software* project. It enables transparant deployment accross networked nodes of plain Orocos C++ components.

Table of Contents

1. The CORBA Transport

This transport allows Orocos components to live in separate processes, distributed over a network and still communicate with each other. The underlying middleware is CORBA, but no CORBA knowledge is required to distribute Orocos components.

The Corba transport provides:

- Connection and communication of Orocos components over a network or between two processes on the same computer.
- Clients (like visualisation) making a connection to any running Orocos component using the IDL interface.
- Transparant use: no recompilation of existing components required. The library acts as a run-time plugin.

2. Setup CORBA Naming (Required!)

Important

Follow these instructions carefully or your setup will not work !

In order to distribute Orocos components over a network, your computers must be setup correctly for using Corba. Start a Corba Naming Service once with multicasting on. Using the TAO Naming Service, this would be:

```
$ Naming_Service -m 1 &
```

And your application as:

```
$ deployer-corba-gnulinux
```

OR: if that fails, start the Naming Service with the following options set:

```
$ Naming_Service -m 0 -ORBListenEndpoints iiop://<the-ns-ip-address>:2809 -ORBDaemon
```

The <*the-ns-ip-address*> must be replaced with the ip address of a network interface of the computer where you start the Naming Service. And each computer where your start the application:

```
$ export NameServiceIOR=corbaloc:iiop:<the-ns-ip-address>:2809/NameService
$ deployer-corba-gnlinux
```

With <*the-ns-ip-address*> the same as above.

For more detailed information or if your deployer does not find the Naming Service, take a look at this page: Using CORBA [http://www.orocos.org/wiki/rtt/frequently-asked-questions-faq/using-corba]

3. Connecting CORBA components

Normally, the Orocos deployer will create connections for you between CORBA components. Be sure to read the OCL DeploymentComponent Manual [http://www.orocos.org/sta-

ble/documentation/ocl/v2.x/doc-xml/orocos-deployment.html] for detailed instructions on how you can setup components such that the can be used from another process.

This is an example deployment script 'server-script.ops' for creating your first process and making one component available in the network:

```
import("ocl")                          // make sure ocl is loaded

loadComponent("MyComponent","TaskContext")  // Create a new default TaskContext
server("MyComponent",true)                  // make MyComponent a CORBA server, and
                                            // register it with the Naming Service ('true')
```

You can start this application with:

```
$ deployer-corba-gnulinux -s server-script.ops
```

In another console, start a client program 'client-script.ops' that wishes to use this component:

```
import("ocl")                          // make sure ocl is loaded

loadComponent("MyComponent","CORBA")        // make 'MyComponent' available in this
program
MyComponent.start()                         // Use the component as usual...connect ports etc.
```

You can start this application with:

```
$ deployer-corba-gnulinux -s client-script.ops
```

More CORBA deployment options are described in the OCL Deployment-Component Manual [http://www.orocos.org/stable/documentation/ocl/v2.x/doc-xml/orocos-deployment.html].

4. In-depth information

You don't need this information unless you want to talk to the CORBA layer directly, for example, from a non-Orocos GUI application.

4.1. Status

The Corba transport aims to make the whole Orocos Component interface available over the network. Consult the *Component Builder's Manual* for an overview of a Component's interface.

These Component interfaces are available:

- TaskContext interface: fully (TaskContext.idl)
- Properties/Attributes interface: fully (ConfigurationInterface.idl)
- OperationCaller/Operation interface: fully (OperationInterface.idl)
- Service interface: fully (Service.idl, ServiceRequester.idl)

- Data Flow interface: fully (DataFlow.idl)

4.2. Limitations

The following limitations apply:

- You need the **typegen** command from the 'orogen' package in order to communicate custom structs/data types between components.
- Interacting with a remote component using the CORBA transport will never be real-time. The only exception to this rule is when using the data flow transport: reading and writing data ports is always real-time, the transport of the data itself is not a real-time process.

5. Code Examples

Note

You only need this example code if you don't use the deployer application!

This example assumes that you have taken a look at the 'Component Builder's Manual'. It creates a simple 'Hello World' component and makes it available to the network. Another program connects to that component and starts the component interface browser in order to control the 'Hello World' component. Both programs may be run on the same or on different computers, given that a network connection exists.

In order to setup your component to be available to other components *transparantly*, proceed as:

```
// server.cpp
#include <rtt/transports/corba/TaskContextServer.hpp>

#include <rtt/Activity.hpp>
#include <rtt/TaskContext.hpp>
#include <rtt/os/main.h>

using namespace RTT;
using namespace RTT::corba;

int ORO_main(int argc, char** argv)
{
  // Setup a component
  TaskContext mycomponent("HelloWorld");
  // Execute a component
  mycomponent.setActivity( new Activity(1, 0.01 );
  mycomponent.start();

  // Setup Corba and Export:
  corba::TaskContextServer::InitOrb(argc, argv);
  TaskContextServer::Create( &mycomponent );

  // Wait for requests:
  TaskContextServer::RunOrb();

  // Cleanup Corba:
  TaskContextServer::DestroyOrb();
  return 0;
```

```
}
```

Next, in order to connect to your component, you need to create a 'proxy' in another file:

```
// client.cpp
#include <rtt/transports/corba/TaskContextServer.hpp>
#include <rtt/transports/corba/TaskContextProxy.hpp>

#include <ocl/TaskBrowser.hpp>
#include <rtt/os/main.h>

using namespace RTT::corba;
using namespace RTT;

int ORO_main(int argc, char** argv)
{
  // Setup Corba:
  corba::TaskContextServer::InitOrb(argc, argv);

  // Setup a thread to handle call-backs to our components.
  corba::TaskContextServer::ThreadOrb();

  // Get a pointer to the component above
  TaskContext* component = TaskContextProxy::Create( "HelloWorld" );

  // Interface it:
  TaskBrowser browse( component );
  browse.loop();

  // Stop ORB thread:
  corba::TaskContextServer::ShutdownOrb();
  // Cleanup Corba:
  TaskContextServer::DestroyOrb();
  return 0;
}
```

Both examples can be found in the corba-example package on Orocos.org. You may use 'connectPeers' and the related methods to form component networks. Any Orocos component can be 'transformed' in this way.

6. Timing and time-outs

By default, a remote method invocation waits until the remote end completes and returns the call, or an exception is thrown. In case the caller only wishes to spend a limited amount of time for waiting, the TAO Messaging service can be used. OmniORB to date does not support this service. TAO allows timeouts to be specified on ORB level, object (POA) level and method level. Orocos currently only supports ORB level, but if necessary, you can apply the configuration yourself to methods or objects by accessing the 'server()' method and casting to the correct CORBA object type.

In order to provide the ORB-wide timeout value in seconds, use:

```
// Wait no more than 0.1 seconds for a response.
ApplicationSetup::InitORB(argc, argv, 0.1);
```

TaskContextProxy and TaskContextServer inherit from ApplicationSetup, so you might as well use these classes to scope InitORB.

7. Orocos Corba Interfaces

Orocos does not require IDL or CORBA knowledge of the user when two Orocos components communicate. However, if you want to access an Orocos component from a non-Orocos program (like a MSWindows GUI), you need to use the IDL files of Orocos.

The relevant files are:

- TaskContext.idl: The main Component Interface file, providing CORBA access to a TaskContext.
- Service.idl: The interface of services by a component
- ServiceRequester.idl: The interface of required services by a component
- OperationInterface.idl: The interface for calling or sending operations.
- ConfigurationInterface.idl: The interface for attributes and properties.
- DataFlow.idl: The interface for communicating buffered or unbufferd data.

All data is communicated with CORBA::Any types. The way of using these interfaces is very similar to using Orocos in C++, but using CORBA syntax.

8. The Naming Service

Orocos uses the CORBA Naming Service such that components can find each other on the same or different networked stations. See also Using CORBA [http://www.orocos.org/wiki/rtt/frequently-asked-questions-faq/using-corba] for a detailed overview on using this program in various network environments or for troubleshooting.

The components are registered under the naming context path "TaskContexts/*ComponentName*" (*id* fields). The *kind* fields are left empty. Only the components which were explicitly exported in your code, using corba::TaskContextServer, are added to the Naming Service. Others write their address as an IOR to a file "*ComponentName*.ior", but you can 'browse' to other components using the exported name and then using 'getPeer()' to access its peer components.

8.1. Example

Since the multicast service of the CORBA Naming_Server behaves very unpredictable (see this link [http://www.theaceorb.com/faq/index.html#115]), you shouldn't use it. Instead, it is better started via some extra lines in /etc/rc.local:

```
#############################################################################
 #  Start CORBA Naming Service
 echo Starting CORBA Naming Service
 pidof Naming_Service || Naming_Service -m 0 -ORBListenEndpoints iiop://192.168.246.151:2809
-ORBDaemon

#############################################################################
```

Where 192.168.246.151 should of course be replaced by your ip adres (using a hostname may yield trouble due to the new 127.0.1.1 entries in /etc/hosts, we think).

All clients (i.e. both your application and the ktaskbrowser) wishing to connect to the Naming_Service should use the environment variable NameServiceIOR

```
[user@host ~]$ echo $NameServiceIOR
corbaloc:iiop:192.168.246.151:2809/NameService
```

You can set it f.i. in your .bashrc file or on the command line via

```
export NameServiceIOR=corbaloc:iiop:192.168.246.151:2809/NameService
```

See the orocos website for more information on compiling/running the ktaskbrowser.