Distributing Communication with Real-Time Message Queues

Revision History

Revision 0.1	November 2, 2009	ps
	Initial version	

Abstract

This document explains the principles of the *MQueue Library* of Orocos, the *Open RObot COntrol Software* project. It enables real-time communication between processes on the same node.

Table of Contents

1. Overview

This transport allows to do inter-process communication between Orocos processes on the same node. It uses the POSIX messages queues where available. This includes GNU/Linux systems and Xenomai.

The MQueue transport provides:

- Connection and Communication of Orocos data flow streams between processes
- The ability to set these up using C++ syntax.
- The ability to set these up using the Corba transport by creating the MQueue as an 'Out-Of-Band' transport.

1.1. Status

As of this writing, MQueues only transport data flow as streams.

1.2. Requirements and Setup

You must enable the ENABLE_MQUEUE flag in CMake. This will, depending on your target, try to detect your mqueue.h header file and library. MQueue also requires the boost::serialization library.

Only Gnu/Linux and Xenomai installations which provide this header can be used.

The transport must get to know your data type. There are two options. If your data type is only Plain Old Data (POD), meaning, it does not contain any pointers or dynamically sized objects, the transport can byte-copy your data. If your data type is more complex, it must use the boost::serialization library to transport your type and your type must be known to this framework.

See below on how to do this.

2. Transporting user types.

Be sure to read the 'Writing Plugins' manual such that your data type is already known to the RTT framework. This section extends that work to make the known data type transportable over MQueues.

2.1. Transporting 'simple' data types

Simple data types without pointers or dynamically sized objects, can be transported quite easily. They are added as such:

```
// myapp.cpp
#include <rtt/types/TemplateTypeInfo.hpp>
#include <rtt/transports/mqueue/MQTemplateProtocol.hpp>

using namespace RTT;
using namespace RTT::mqueue;
using namespace RTT::types;

struct MyData {
  double x,y,x;
  int stamp;
};

int ORO_main(int argc, char** argv)
{

   // Add your type to the Orocos type system (see: Writing plugins)
   Types()->addType( new types::TemplateTypeInfo<MyData, false>("MyData") );

   // New: Install the template protocol for your data type.
   Types()->getType("MyData")->addTransport(ORO_MQUEUE_PROTOCOL_ID, new
mqueue::MQTemplateProtocol<MyData>() );

   // rest of your program can now transport MyData between processes.

}
```

As the code shows, only one line of code is necessary to register simple types to this transport.

In practice, you'll want to write a plugin which contains this code such that your data type is loaded in every Orocos application that you start.

2.2. Transporting 'complex' data types

Data types like std::vector or similar can't just be byte-copied. They need special treatment for reading and writing their contents. Orocos uses the boost::serialization library for this. This library already understands the standard containers (vector,list,...) and is easily extendable to learn your types. Adding complex data goes as such:

```
// myapp.cpp
#include <rtt/types/TemplateTypeInfo.hpp>
#include <rtt/transports/mqueue/MQSerializationProtocol.hpp>

using namespace RTT;
using namespace RTT::mqueue;
using namespace RTT::types;

struct MyComplexData {
  double x,y,x;
  std::vector<int> stamps;
  MyComplexData() { stamps.resize(10, -1); }
};

// New: define the marshalling using boost::serialization syntax:
namespace boost {
namespace serialization {

template<class Archive>
void serialize(Archive & ar, MyComplexData & d, const unsigned int version)
{
  ar & d.x;
  ar & d.y;
  ar & d.z;
  ar & d.samps; // boost knows std::vector !
}
}
}

int ORO_main(int argc, char** argv)
{

   // Add your type to the Orocos type system (see: Writing plugins). Same as simple case.
   Types()->addType( new types::TemplateTypeInfo<MyComplexData,
false>("MyComplexData") );

   // New: Install the Serialization template protocol for your data type.
   Types()->getType("MyComplexData")->addTransport(ORO_MQUEUE_PROTOCOL_ID, new
mqueue::MQSerializationProtocol<MyComplexData>() );

   // rest of your program can now transport MyComplexData between processes.

}
```

When comparing this to the previous section, only two things changed: We defined a serialize() function, and used the MQSerializationProtocol instead of the MQTemplateProtocol to register our data transport. You can find a tutorial on writing your own serialization func-

tion on: The Boost Serialization Website [http://www.boost.org/doc/libs/1_40_0/libs/serialization/doc/index.html].

3. Connecting ports using the MQueue transport

Orocos will not try to use this transport by default when connecting data flow ports. You must tell it explicitly to do so. This is done using the ConnPolicy object, which describes how connections should be made.

In addition to filling in this object, you need to setup an outgoing data stream on the output port, and an incomming data stream at the input port which you wish to connect. This can be done in C++ with or without the help from the CORBA transport.

3.1. Bare C++ connection

If you don't want to use CORBA for setting up a connection, you need to use the createStream function to setup a data flow stream in each process. This requires you to choose a name of the connection and use this name in both processes:

```
// process1.cpp:

 // Your port is probably created in a component:
 OutputPort<MyData> p_out("name");

 // Create a ConnPolicy object:
 ConnPolicy policy = buffer(10); // buffered connection with 10 elements.
 policy.transport = ORO_MQUEUE_PROTOCOL_ID; // the MQueue protocol id
 policy.name_id   = "mydata_conn";          // the connection id

 p_out.createStream( policy );
 // done in proces1.cpp

// process2.cpp:

 // Your port is probably created in a component:
 InputPort<MyData> p_in("indata");

 // Create a ConnPolicy object:
 ConnPolicy policy = ConnPolicy::buffer(10); // buffered connection with 10 elements.
 policy.transport = ORO_MQUEUE_PROTOCOL_ID; // the MQueue protocol id
 policy.name_id   = "mydata_conn";          // the connection id

 p_in.createStream( policy );
 // done in proces2.cpp . We can now transmit data from process1 to
 // process2 .
```

Both ends must specify the same connection policy. Also, the RTT assumes that the createStream is first done on the output side, and then on the input side. This is because it is an error to connect an input side without an output side producing data. When an output side opens a connection, it will send in a test data sample, which will notify the input side that someone is sending, and that the connection is probably correctly set up.

If either output or input would disapear after the connection has been setup (because their process crashed or did not clean up), the other side will not notice this. You can re-start your component, and the ports will find each other again.

If you want proper connection management, you need to use the CORBA approach below, which keeps track of appearing and disappearing connections.

3.2. CORBA managed connections

The CORBA transport supports 'Out-Of-Band' (OOB) connections for data flow. This means that CORBA itself is used to setup the connection between both ports, but the actual data transfer is done using OOB protocol. In our case, CORBA will be used to setup or destroy MQueue streams.

This has several advantages:

- Dead streams are cleaned up. CORBA can detect connection loss.
- You don't need to figure out a common connection name, the transport will find one for you and CORBA will sync both sides.
- Creating out-of-band connections using the CORBA transport has the same syntax as creating normal connections.
- The CORBA transport will make sure that first your output stream is created and then your input stream, and will cleanup the output stream if the input stream could not be created.

So it's more robust, but it requires the CORBA transport.

An Out-Of-Band connection is always setup like this:

```
TaskContext *task_a, *task_b;
// init task_a, task_b...

ConnPolicy policy = ConnPolicy::buffer(10);

// override default transport policy to trigger out-of-band:
policy.transport = ORO_MQUEUE_PROTOCOL_ID;

// this is the standard way for connecting ports:
task_a->ports()->getPort("name")->connectTo( task_b->ports()->getPort("outdata"), policy );
```

The important part here is that a policy.transport is set, while using the connectTo function of base::PortInterface. Normally, setting the transport is not necessary, because the RTT will figure out itself what the best means of transport is. For example, if both ports are in the same process, a direct connection is made, if one or both components are proxies, the transport will use the transport of the proxies, in our case CORBA. However, the transport flag overrides this, and the connection logic will pick this up and use the specified transport.

Overriding the transport parameter even works when you want to test over-CORBA or over-MQueue transport with using two process-local ports. The only thing to do is to set the transport parameter to the protocol ID.

Finally, if you want to use the CORBA IDL interface to connect two ports over the mqueue transport, the workflow is fairly identical. The code below is for C++, but the equivalent can be done in any CORBA enabled language:

```
#include <rtt/transports/corba/CorbaConnPolicy.hpp>
// ...
using namespace RTT::corba;

CControlTask_var task_a, task_b;
// init task_a, task_b...

CConnPolicy cpolicy = toCORBA( RTT::ConnPolicy::buffer(10) );

// override default transport policy to trigger out-of-band:
cpolicy.transport = ORO_MQUEUE_PROTOCOL_ID;

// this is the standard way for connecting ports in CORBA:
CDataFlowInterface_var dataflow_a = task_a->ports();
CDataFlowInterface_var dataflow_b = task_b->ports();

dataflow_a->createConnection("name", dataflow_b, "outdata", cpolicy );
```

Similar as connectTo above, the createConnection function creates a fully managed connection between two data flow ports. We used the toCORBA function from CorbaConnPolicy.hpp to convert RTT policy objects to CORBA policy objects. Both RTT::ConnPolicy and RTT::corba::CConnPolicy structs are exactly the same, but RTT functions require the former and CORBA functions the latter.

Alternatively, you can use the create streams functions directly from the CORBA interface, in order to create unmanaged streams. In that case, the code becomes:

```
#include <rtt/transports/corba/CorbaConnPolicy.hpp>
// ...
using namespace RTT::corba;

CControlTask_var task_a, task_b;
// init task_a, task_b...

CConnPolicy cpolicy = toCORBA( RTT::ConnPolicy::buffer(10) );

// override default transport policy and provide a name:
cpolicy.transport = ORO_MQUEUE_PROTOCOL_ID;
cpolicy.name_id = "stream_name";

// this is the standard way for connecting ports in CORBA:
CDataFlowInterface_var dataflow_a = task_a->ports();
CDataFlowInterface_var dataflow_b = task_b->ports();

dataflow_b->createStream("outdata", cpolicy );
dataflow_a->createStream("name", cpolicy );
```

Note that creating message queues like this leaves out all management code and will not detect broken connections. It has the same constraints as if the streams were setup in C++, as shown in the previous section.