NetworkCentric
Core Avionics

RODOS tutorial Middleware

Version: 1.0
Date: 10.10.2008
Author: Sergio Montenegro

RODOS Tutorial Middleware

How to begin.

Read and understand first the RODOS-Tutorial and then the RODOS-Middleware tutorial.

use

```
     % linux-executable <list of sources>
to compile and
     %tst
```

to execute – e.g.

```
     % linux-executable sender.cc
     % tst
```

For each example program please first read and understand the code, then compile and execute and see if it acts as expected. Then modify and continue trying.

Try to combine different programs together.

Useful to know:

For output in masks we use the support_programms/maskgenerator. Please first take a look to mask.txt, you will find the same macro names [NAME] in all programs.

The middleware communication is based on a publisher/subscriber protocol. This is a multicast protocol. There is no connection from a sender to a receiver (connectionless).

Publishers make messages public under a given topic.

Subscribers to a given topic get all messages which are published under the given topic.

To establish a transfer path, both the publisher and subscriber have to share the same topic.

A Topic is a pair - data-type and an integer representing a topic identifier.

To match a communication pair, both data-type and topic identifier must match.

For a topic there may be zero, one or several subscribers. A message which is published under a topic will be distributed to 0 or more subscribers.

Each subscriber has a reference to the associated topic and a “putter” to store messages.

All subscribers are registered in a list. Each time a message is published the list of all subscribers will be searched and for each subscriber where the topic matches the associated putter will be called to store a copy of the published message.

Using a network interface and the corresponding gateways, the middleware may make the node borders transparent. Applications running on different computers may communicate as if they were on the same computer.

Stabilising common topics

demo_topics.h declares all topics and data structures used for communication for the tutorial. These topics and data structures shall be known by all communicating applications.

Therefore it has to be declared globally and all applications must include this file.

demo_topics.cc allocates all topics. For each topic declaration the corresponding (exactly one) object has to be created. There may not be replications of topic objects.

Try to replicate topic objects or topic identifiers. You shall get an error message.

First try to compile the topics:

```
% linux-executable demo_topics.cc
% tst
```

This program does nothing, but you can get error messages if you replicate topics.

Publisher / Sender

There are some examples of senders:

sender01.cc is a simple publisher with a thread to send simple messages.

sender02.cc is a publisher which sends simple messages from an event handler. No thread is needed.

senderPosition.cc sends more complex messages using an extra data strcuture.

You may compile one or more of them together. Each compilation has to have the demo_topics.cc

```
%linux-executable demo_topics.cc sender01.cc
%tst
%linux-executable demo_topics.cc sender02.cc
```

```
%tst
%linux-executable demo_topics.cc sender02.cc senders01.cc
%tst
%linux-executable demo_topics.cc sender02.cc senderPosition.cc
%tst
%linux-executable demo_topics.cc sender*.cc
%tst
```

Compile and execute different combinations. You will see the reports of each sender each time it publishes in a topic. There is no subscriber, therefore no message will be received.

Subscribers/Receivers

There are several examples of subscribers. You may compile one or more together, but as for the publisher, each compilation has to have the demo_topics.cc.

receiverthread.cc is the most simple receiver, a thread waits on a topic until something arrives. But his method can loose messages if the thread is not waiting when the message arrives.

receiverputter.cc is subscriber which creates an own putter to handle messages to topics conunter1 and counter2. You man try to add or remove topics. No Thread is required to access the messages.

receivercommbuff.cc has a CommBuffer as a putter, to store the last published message. It provides a Thread which look periodic to the buffer.

receiverasync.cc has a asynchronous FIFO to store the messages. It provides a Thread which look periodic to the fifo.

receiversync.cc has a synchronous FIFO to store the messages. It provides a Thread which is activated each time some one (a publisher) has written data to the fifo (as a putter).

Try to compile one or more of them together. There is no publisher, therefore no subscriber will get data.

```
%linux-executable demo_topics.cc receiverputter.cc
%tst
%linux-executable demo_topics.cc receiver*.cc
%tst
```

Publishing and receiving

Now we may combine publishers and receivers.

See which publisher publishes which topic and see which subscriber receives which topic.

Try different combinations or all senders and all receivers.

Eg:

```
    %linux-executable demo_topics.cc receiverputter.cc sender01.cc
    %tst
```

or

```
    %linux-executable *.cc
    %tst
```

A Shortcut

Because it is common to make mistakes when declaring objects as external in a .h file and creating the corresponding objects in a .cc file, we added a shortcut using defines.

Using this shortcut you just declare the external objects in the .h file and the .cc file will be adapted to it automatically.

Just move

demo_topics.h_extra -> demo_topics.h

and

demo_topics.cc_extra -> demo_topics.cc

First try to understand how it works using the cpp pre-processor for g++.

see the example in

```
    % preprocessor_demo
```

for beginners it is better to not use this shortcut, but for big projects where many people work, we do recommend to use it.