

micROS-drt User's Guide

micROS-drt, an implementation of ROS 1.x on DDS



(for micROS-drt v0.30, last edited on 30.09.2015)

Part I Introduction

1.1 What is micROS-drt?

micROS-drt(micROS distributed real-time) is a modified ROS1 C++ kernel (i.e., the roscpp package) which adopts DDS (Data Distribution Systems for Realtime Systems) as its underlying message transfer protocol. DDS is an Object Management Group's standard for pub/sub middleware (<http://portals.omg.org/dds/>). It supports high-performance, scalable and QoS-assuring message delivery. It has been applied into many industry-level systems.

1.2 Why is micROS-drt?

By replacing the original ROS message protocols (TCPROS & UDPROS) with DDS, we can support a set of advanced features in ROS message delivery process, such as UDP-based multicast and setting the QoS properties (transport priority, latency budget, etc.) of a specific topic. In concrete, we can achieve the benefits as follows.

General benefits

(1) Multicast in message delivery. The DDS middleware will automatically select the multicast protocol when necessary. Therefore, when there are n listeners in a topic ($n \geq 2$), you can achieve significant performance benefit.

(2) Robustness in unexpected settings. Since DDS is a mature and industry-level message delivery system, it behaves better in various complex communications settings. For example, it has better reconnection behavior when dropping out of wireless.

QoS-related benefits

(1) Transport priority and latency budget of a topic. With micROS-drt, you can specify the transport priority and latency budget of a topic. It is useful in many real-time settings, such as multi-robot collaboration with limited wireless network bandwidth.

(2) Expected message arriving deadline. With micROS-drt, you can set the expected message arriving period as well as the behavior when a message not arriving in this period. It is also useful in real-time settings.

(3) Time-based message filter. You can filter unnecessary messages based on time properties. For example, if you need only update the state of a sensor per minute, you can ignore other redundant messages by simply setting a property while you subscribing the topic.

(4) Other QoS-related features. With micROS-drt, you can specify other QoS-related properties provided by the underlying DDS middleware in a simplified way when advertising/subscribing a topic, such as reliability of message delivery, data-centric update, message order method, and message valid period.

Compatibility

(1) Existing ROS packages need no modification. Existing ROS packages can easily benefit from DDS by simply replacing the libroscpp.so in the ROS installation directory. Those packages needs no modification and re-compilation.

(2) Existing ROS programming paradigm is kept. Existing ROS programming paradigm is kept,

except for several new APIs to set topic QoS are added.

(3) Interoperable with official ROS kernel. The micROS-drt can smartly choose appropriate protocol when it communicate with an official C++ or Python ROS kernel. The preferred protocol of a topic can also be specified when advertising it.

In fact, the ROS official has also the plan to integrate ROS 2.0, the next generation of ROS, with DDS (http://design.ros2.org/articles/ros_on_dds.html). ROS2.0 will be a great milestone of the development of ROS. We hope micROS-drt will be an alternative to it under certain circumstances. The major difference between micROS-drt and future ROS 2.0 includes: (1) We keeps compatibility with existing ROS 1.x package and programming paradigm, since protecting the existing investment is important to the ROS community. (2) We lays strong emphasis on exploiting the QoS-features of the underlying DDS middleware, because it is important to multi-node and multi-robot applications.

Part II Installation Guide

There are two approaches to install micROS-drt. The first one is replacing the official roscpp library (libroscpp.so) with the DDS-enabled library we provided. It is convenient but strict with the running environment, which must be identical with our one, i.e., x86 or x64-based Ubuntu 14.04 and ROS Indigo. Alternatively, you can chose installing micROS-drt from its source code.

2.1 Prerequisite

Before installing micROS-drt, you should install a DDS-compliant middleware. Currently, micROS-drt supports OpenSplice DDS 6.4 community edition (<http://www.prismtech.com/dds-community>) and OpenDDS 3.6 (<http://www.opendds.org/>).

For those who chose OpenSplice DDS, please ensure that the DDS environment variables have been set correctly. Usually, you can achieve this goal by simply running

```
source %DDSInstallationPath%/release.com
```

in which the %DDSInstallationPath% is the installation path of your OpenSplice DDS. Besides, if you chose installing micROS-drt from its source code, this path should be the HDE directory of your Opensplice DDS. More details can be found in the manual of OpenSplice DDS.

For those who chose OpenDDS, please ensure that the DDS environment variables have been set correctly. Usually, you can achieve this goal by simply running

```
source %DDSInstallationPath%/setenv.sh
```

in which the %DDSInstallationPath% is the installation path of your OpenDDS.

2.2 Installation from binary library

- (1) Installing ROS Indigo.
- (2) Downloading libroscpp.so from [here](#). Please choose your architecture (x86 or x64) and your DDS production (OpenSplice DDS or OpenDDS).
- (3) Replacing the libroscpp.so in the ROS directory with the file you downloaded.
- (4) If you want to use micROS-drt newly added APIs, please download the micROS-drt header files from [here](#) and copy them into the include/ros directory.

2.3 Installation from source code (Recommended)

- (1) Installing ROS Indigo from source code (<http://wiki.ros.org/indigo/Installation/Source>)

- (2) Downloading the modified roscpp source code from [here](#).
- (3) Replacing src/ros_comm/roscpp directory with the directory you downloaded.
- (4) Finding the “#set DDS VENDOR” line in the CMakeList.txt and then modifying this part to set the DDS middleware vendor. If you select OpenSplice DDS, it looks like

```
#set DDS VENDOR
option(USE_OPENSPLICE_DDS "Using OpenSplice DDS" ON)
option(USE_OPENDDS "Using OpenDDS" OFF)
```

and if your select OpenDDS, it looks like

```
#set DDS VENDOR
option(USE_OPENSPLICE_DDS "Using OpenSplice DDS" OFF)
option(USE_OPENDDS "Using OpenDDS" ON)
```

- (5) Recompiling the roscpp package. You can add “--pkg roscpp” to the catkin_make_isolated command to achieve this goal.

Part III Developer's Guide

As we have mentioned, existing ROS applications can easily benefit from DDS without modification. After replacing the libroscpp.so or recompile the roscpp package (and the applications when necessary), all ROS messages will be automatically switched to the DDS middleware. However, if you want to set the real-time QoS parameters of a topic, you should modify your program and use the newly added APIs in micROS-drt.

Note that the QoS parameters you specified only take effects when you using micROS-drt on both publisher and subscriber side. And the QoS parameters are only available to the topic-based pub/sub communication.

3.1 Setting QoS parameters

In micROS-drt, the QoS parameters are set on the node-topic level, which means you can specify different message delivery QoS parameters (e.g., transport priority and latency budget) for each topic on each node. Even two nodes both publish messages on one topic, they can set different transport priority. Besides, the publisher and the subscriber can specify different QoS parameters for their actions.

Table I QoS Parameters that micROS-drt supports

Message publisher QoS Parameters (Using advertiseWithQoS() method)	
Transport priority	Message transport priority
Latency budget	Expected latency from publisher to the subscriber
Best effort delivery	Using the best effort message delivery protocol or not
Data centric update	Not supported yet
Message valid period	The valid period (lifespan) of a message
Message subscriber QoS Parameters (Using advertiseWithQoS() method)	
Deadline	Expected message arrival rate (Not supported yet)
Behavior when deadline is not fulfilled	Not supported yet

Message order method	Message is ordered by the sending time stamp or the arriving sequence
Time filter duration	Performing time-based message filtering
Best effort delivery	Using the best effort message delivery protocol or not
data_centric_update	Not supported yet

In the original ROS, we use the `NodeHandle::advertise()` method to create a Publisher which is used to publish on a topic, and the `NodeHandle::subscribe()` to subscribe the messages on a topic. In `micROS-drt`, we add two new method, `NodeHandle::advertiseWithQoS()` and `NodeHandle::subscribeWithQoS()`. They can be regarded as the QoS available version of `advertise()` and `subscribe()`. If you want to specify the QoS parameters, you should use them instead of the original ones.

3.1.1 advertiseWithQoS()

The `advertiseWithQoS()` has the following variations. The meaning of the parameter *topic*, *queue_size*, *latch* and *ops* are identical to the original `advertise()` method.

1. *Publisher advertiseWithQoS(const std::string& topic, uint32_t queue_size, TransportPriority priority, bool latch = false)*
2. *Publisher advertiseWithQoS(const std::string& topic, uint32_t queue_size, Duration latency_budget, bool latch = false)*
3. *Publisher advertiseWithQoS(const std::string& topic, uint32_t queue_size, AdvertiseQoSOptions& qos_ops, bool latch = false)*
4. *Publisher advertiseWithQoS(AdvertiseOptions& ops, AdvertiseQoSOptions& qos_ops);*

The first one just added a *priority* parameter to the original `advertise()` method. Its type, `TransportPriority`, is an enum type which is defined as follows:

```
enum TransportPriority
{ ExtremelyLow, VeryLow, Low, Normal, High, VeryHigh, ExtremelyHigh };
```

In other words, you can set the transport priority of the messages of this topic by selecting one from those above seven levels. For example, you can set the transport priority of the topic “chatter” to the low level by the following statement.

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000, ros::Low);
```

The second one just added a *latency_budget* parameter to the original `advertise()` method. It can set the delivery latency budget (from the publisher to the listener) of the message on this topic. It is a hint to the underlying DDS middleware, which will automatically adapt its behavior to meet the requirements of the shortest delay if possible. It is of the `ros::Duration` type.

The third one and the fourth one involves a new structure named `qos_ops`. This structure are defined as follows:

```
struct AdvertiseQoSOptions
{
    TransportPriority transport_priority; //message transport priority of this topic
    Duration latency_budget; //message latency budget of this topic
    bool using_best_effort_protocol; //using best effort transport protocol or not
    bool data_centric_update; //using data centric update or not
    Duration msg_valid_period; // message valid period
};
```

This structure controls the QoS behavior when this node publishes a message on this topic. The meaning of its members are as follows:

transport_priority. We have explained it earlier. The default value of this member is `ros::Normal`.

latency_budget. We have explained it earlier. The default value of this member is `ros::DURATION_MIN`, which tells the underlying DDS middleware to deliver the message as soon as possible.

using_best_effort_protocol. It tells the underlying DDS middleware to use the best effort transport protocol or the reliable one. When using the unreliable transport protocol, DDS will only attempt to deliver the data, and no arrival-checks are being performed and any lost data is not re-transmitted (non-reliable). When using the reliable transport protocol, extra arrival-checks are performed and data may get re-transmitted in case of lost data. However, it may cause some extra performance cost. The default value of this member is *false*, which means using the reliable protocol.

data_centric_update. Not formally supported in this version.

msg_valid_period. It tells the underlying DDS middleware the valid period (lifespan) of a message. When this time period has expired and the subscriber doesn't receive it yet, the message will be automatically discarded. It is useful to reduce network traffic under certain circumstances. Its default value is `ros::DURATION_MAX`, indicating that the message does not expire.

Note that if you set *using_best_effort_protocol* to true, you have to set the QoS parameter *using_best_effort_protocol* of the subscriber to true as well. Or else your subscriber cannot receive the messages published by this node, because they adopt different protocols.

The following code is an example to use the third variation of `advertiseWithQoS()`. Note that when declaring a *AdvertiseQoSOptions*, its members will have default values and you can only modify the members you want to change.

```
ros::AdvertiseQoSOptions adv_qos_ops;  
adv_qos_ops.using_best_effort_protocol=true;  
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000, adv_qos_ops);
```

3.1.2 *subscribeWithQoS()*

The `subscribeWithQoS()` has the following variations. The meaning of the parameter *topic*, *queue_size*, *<callback>* and *ops* are identical to the original `advertise()` method. We only add a new parameter *qos_ops* whose type is *SubscribeQoSOptions*.

1. *Subscriber subscribeWithQoS(const std::string& topic, uint32_t queue_size, SubscribeQoSOptions& qos_ops, <callback>)*
2. *Subscriber subscribeWithQoS(SubscribeOptions& ops, SubscribeQoSOptions& qos_ops);*

SubscribeQoSOptions are defined as follows:

```
struct SubscribeQoSOptions  
{  
    Duration deadline;  
    DeadlineMissedCallback deadline_cb;  
    bool ordered_by_sending_timestamp;  
    Duration time_filter_duration;  
    bool using_best_effort_protocol;
```

```

    bool data_centric_update;
};

```

This structure controls the QoS behavior related to the action of this node subscribing this topic. The meaning of its members are as follows:

deadline. It sets the period within which this node expects a new message on this topic. Its default value is `ros::DURATION_MAX`. And it is not formally supported in this version.

deadline_cb. It is a callback function which will be invoked when no new message is arrived before the deadline. It is not formally supported in this version as well.

ordered_by_sending_timestamp. It controls the message is ordered according to the sending time stamp (publisher side) or the arriving time stamp (local side). Its default value is *true*.

time_filter_duration. It specifies the time period to filter the messages. In a time period, only one message will be received and other messages will be discarded. For example, if a sensor node publishes its state at a 100hz frequency and another node only wants to process the state 1 times per second, you can use this parameter to filter unnecessary messages. It is of `ros::Duration` type and its default value is `ros::DURATION_MIN`, which means no filter.

using_best_effort_protocol. It tells the underlying DDS middleware to use the best effort transport protocol or the reliable one. When using the unreliable transport protocol, DDS doesn't support message arrival-checks are being performed and any lost data is not re-transmitted (non-reliable). When using the reliable transport protocol, extra arrival-checks are performed and data may get re-transmitted in case of lost data. However, it may cause some extra performance cost. The default value of this member is *false*, which means using the reliable protocol.

data_centric_update. Not formally supported in this version.

Note that the `subscribeWithQoS()` method doesn't have the `transport_hints` parameter in contrast with the original `subscribe()` method, since when you specify a QoS requirement the kernel will choose DDS protocol automatically.

3.2 Selecting the preferred protocol of a topic

At runtime, the `micROS-drtkernel` has the ability to negotiate the message delivery protocol with other nodes. It will choose the DDS protocol in the first place. However, if the remote node is an official ROS kernel without DDS protocol support, it will smartly switch to ROSTCP or ROSUDP. This ability is realized based on an enhancement of the official ROS's protocol negotiate framework.

If you don't want to use DDS on a topic, you can specify the preferred protocol when you subscribe the topic using the `transport_hints` parameter of the `subscribe()` method. The following statement shows that the preferred protocol is DDS.

```

ros::Subscriber sub = nh.subscribe("my_topic", 1, callback, ros::TransportHints().dds());

```

Please refer to [here](#) for more information about the `transport_hints` parameter.

3.3 Examples

3.3.1 Writing a Publisher with QoS

The following program creates three topics with different transport priority.

```

#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Publisher high_pub, normal_pub, low_pub;

    high_pub = n.advertiseWithQoS<std_msgs::String>("high", 1000, ros::High);
    normal_pub = n.advertiseWithQoS<std_msgs::String>("normal", 1000, ros::Normal);
    low_pub = n.advertiseWithQoS<std_msgs::String>("low", 1000, ros::Low);

    ros::Rate loop_rate(20);

    while (ros::ok())
    {
        //Publishing messages on each topic here
        .....

        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}

```

3.3.1 Writing a Subscriber with QoS

The following program subscribe a topic with a time-based filter.

```

#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;

    ros::Duration d(2, 0);
    ros::SubscribeQoSOptions qos_ops;
    qos_ops.time_filter_duration=d;
    ros::Subscriber sub = n.subscribeWithQoS("chatter", 1000, qos_ops, chatterCallback);
    ros::spin();

    return 0;
}

```

More examples can be found [here](#).