RTI DDS Overview

How to use RTI DDS

RTI DDS is used in this project to establish a communication network between all programs / machines that are a part of it. The framework can be used to send and receive messages of user-defined types. These messages can only be sent and received within so-called domains. It also offers a variety of quality of service settings, such as reliability or fault tolerance. For more information on the RTI DDS framework, visit https://www.rti.com/products/connext-dds-professional.

IDL Files

IDL files are used to define message data types. The resulting message classes can be used to create, send and receive message objects.

```
/*
Example of a basic IDL type called OtherType, stored as OtherType.idl.
The struct can contain fields of different types, such as boolean or long.
Include guards are (sadly) necessary.
*/

#ifndef OTHER_TYPE
#define OTHER_TYPE
struct OtherType {
    boolean val1;
    long val2;
}
#endif
```

```
/*
TestType.idl includes some more complex examples.
Other IDL types can be used for fields as well, but must be included.
*/
#include OtherType.idl

#ifndef TEST_TYPE
#define TEST_TYPE

/*Enums can be used to create custom types as well*/
enum Some_enum {
    some_val
    other_val
}

/*
Keys can be used as unique identifiers, similar to databases. Multiple fields can be combined to one key as well,
if they are all marked as //@key.
Sequences are arrays of variable size.
*/
struct TestType {
    string name; //@key
    Some_enum val1;
    sequence<double> values_double;
    OtherType val2;
}
#endif
```

IDL files can be used in different programming languages, but might need to be prepared first. In Matlab, as can be seen in *matlab_basics.md*, IDL files can simply be imported. For C++, rtiddsgen must be used to create header and .cpp files from IDL files, which can then be imported and used in a C++ project (see https://community.rti.com/static/documentation/connext-dds/5.3.0/doc/api/connext*dds/api*cpp2/group__DDSNddsgenModule.html for modern C++).

C++ Examples

DDS Domain Participants

Messages are sent and received within domains. To participate in a domain, for writing and reading data, a domain participant must be created first. With a domain participant, publisher/writer/subscriber/reader objects can be created.

In C++, a default (regarding its QoS settings) domain participant is created like this:

```
#include <dds/domain/DomainParticipant.hpp>
...
dds::domain::DomainParticipant some_participant(domain_id);
```

Each domain has its own unique domain id. This domain id should be used in the constructor. Domains must not be created by the user.

Important note: Only create one domain participant for each domain in which a participation is required, unless different QoS settings on a participant level need to be applied. If necessary, share this participant within your application. More participants for the same domain could lead to unnecessary overhead. For this reason, if you use the *cpm_lib*, always use the participant singleton for domain 0 if a communication within this domain should be performed.

Topics

Topics give a context for a data exchange within a domain. A domain can contain multiple topics. They are used to define the name and the type of a 'conversation' between publisher and subscribers.

There are a different constructors available to create a topic. If no special type names, QoS settings or listeners are required, topics can be set using the following constructor:

```
#include <dds/topic/Topic.hpp>
...
#include "TestType.hpp"
...
dds::topic::Topic<TestType> some_topic(some_participant, topic_name);
```

In this case, the IDL type TestType can be used to send message objects. The topic has the unique identifying name *topic_name* (string) and participates in the domain that the domain participant *some_participant* is connected to.

Topics cannot be created twice within an application. There are several strategies to use a topic in different classes: Make it accessible from the outside, pass it as an argument, create a singleton... If you know that your topic has been created before, you can also use

```
dds::topic::Topic<TestType> same_topic = dds::topic::find<dds::topic::Topic<TestType>>(some_participant,
topic_name);
```

to find an already created topic.

Filters

Filtered topic can be created to only handle data that matches the filter expression (for subscriber). Received data can be filtered as well using a QueryCondition.

```
#include <dds/topic/ddstopic.hpp>
#include <dds/sub/ddssub.hpp>
#include "TestType.hpp"
...
dds::topic::ContentFilteredTopic<TestType> some_filtered_topic(some_topic, filter_topic_name, dds::topic::Filter
("name = test AND ..."));
```

The ContentFilteredTopic is based on the topic created before, is called filter*topic*name and filters by name and other criteria.

This content filtered topic can be used for a subscriber/reader instead of a 'normal' topic. The *cpm_lib* offers a wrapper for creating filtered topics for vehicle ids.

Subscriber / Data Reader

Subscriber can be used in combination with listeners to receive data or to create data reader. QoS settings can be set for subscribers as well. To create a subscriber with default QoS settings and without any listener, type:

```
#include <dds/sub/ddssub.hpp>
...
dds::sub::Subscriber some_subscriber(some_participant);
```

Usually, subscribers are used in combination with data reader. Listeners require more careful resource management and will not be explained further. For more information on listeners, please refer to https://community.rti.com/static/documentation/connext-dds/5.2.0/doc/api/connext*dds/api*cpp2/classListener.html.

Data reader can also be configured regarding their QoS or other settings. To create a basic reader, type:

```
#include <dds/sub/ddssub.hpp>
...
//Use a subscriber that was created before
dss::sub::DataReader<TestType> some_reader(some_subscriber, some_topic);
//Do not use a dedicated subscriber
dds::sub::DataReader<TestType> other_reader(dds::sub::Subscriber(some_participant), some_topic);
```

A data reader offers a read and a take operation. After calling `read()`, the retrieved data remains in the data reader and can be retrieved again, whereas `take()` removes the data. Use the latter unless you need to retrieve the same data from the data reader again another time.

Retrieve data synchronously

To retrieve data synchronously if it is available, type:

```
dds::sub::LoanedSamples<TestType> samples = some_reader.take();

for (auto sample : samples) {
    if (sample.info().valid()) {
        sample.data() ... //Get the message (here of type TestType) and process it
    }
}
```

You can also wait for data until a timeout occurs:

```
#include <dds/core/cond/WaitSet.hpp>
... //Other includes mentioned before
// Create a WaitSet
dds::core::cond::WaitSet waitset;
// Create a ReadCondition for a reader with a specific DataState
dds::sub::cond::ReadCondition read_cond(some_reader, dds::sub::status::DataState::any());
// Attach conditions
waitset += read_cond;
...
//Wait either for 2 seconds or for a new signal, then take the received data (if it exists)
waitset.wait(dds::core::Duration(2, 0));
for (auto sample : reader.take()) {
    if (sample.info().valid()) {
        ...
    }
}
```

Retrieve data asynchronously

```
#include <functional>
#include <dds/sub/ddssub.hpp>
#include <dds/core/ddscore.hpp>
#include <rti/core/cond/AsyncWaitSet.hpp>
...
dds::core::cond::StatusCondition read_condition(some_reader);
rti::core::cond::AsyncWaitSet waitset;

read_condition.enabled_statuses(dds::core::status::StatusMask::data_available());
read_condition->handler(std::bind(&Subscriber::handler, this, func));
waitset.attach_condition(read_condition);
waitset.start();
```

An AsyncWaitSet can be used to asynchronously handle incoming data via a callback function. The callback function *func* is registered as part of the read condition and is triggered whenever data is available. You can either use a std::function object or a lambda function for *func*.

The read condition is attached to the waitset. Other conditions than read conditions can be used as well. After calling `waitset.start()`, the callback function can be triggered. (You can also `stop()` the asynchronous wait)

In the callback function, the waitset must be released again in case another thread wants to access the samples as well. This should (probably) also be performed to allow for future callbacks.

```
...
//Within the callback function or a wrapper function
// Take all samples This will reset the StatusCondition
dds::sub::LoanedSamples<MessageType> samples = reader.take();

// Release status condition in case other threads can process outstanding
// samples
waitset.unlock_condition(dds::core::cond::StatusCondition(reader));

// Process sample
...
```

Samples are processed as shown before.

Publisher / Data Writer

```
#include <dds/pub/ddspub.hpp>
...
dds::pub::Publisher some_publisher(some_participant);
dds::pub::DataWriter<TestType> some_writer(some_publisher, some_topic);
```

The creation of a publisher / data writer is thus similar to the creation of a subscriber / data reader. Publisher can be created with listeners and non-default QoS settings as well. To send data, data writer are used.

Sending Data

```
TestType some_message(...);
some_writer.write(some_message);
```

Data is sent using the `write(...)` function of a data writer. All participants within the same domain are able to receive that message (depending on the QoS settings).

Best Practice

See https://community.rti.com/best-practices/create-few-publishers-and-subscribers-possible and https://community.rti.com/best-practices/create-few-domainparticipants-possible. Also take a look at the other best practices.

QoS

QoS, or quality of service, are properties of the communication used or expected by a participant, publisher, subscriber etc. By changing the QoS settings, these properties can be configured to set up the desired communication structure between all participants, e.g. regarding the reliability or durability of message sending and retrieval.

A few QoS settings will be presented in the following sections. Not all settings are covered, so please refer to https://community.rti.com/glossary/qos for more information.

Note: Some QoS settings must be set for both e.g. data reader and data writer. Otherwise, messages might not be received due to incompatible QoS settings.

Code Examples

History Example

```
dds::sub::DataReader<TestType> reader(some_subscriber, some_topic, (dds::sub::qos::DataReaderQos() << dds::core::
policy::History::KeepAll()));
```

The data reader will now keep all samples it received until max*sample*sper_instance samples have been stored (set by the *resource limits* QoS policy). Samples are still deleted when the `take()` function is used.

Reliability Example

```
dds::sub::DataReader<TestType> reader(some_subscriber, some_topic, (dds::sub::qos::DataReaderQos() << dds::core::
policy::Reliability::Reliable()))
...
dds::pub::DataWriter<TestType> writer(some_publisher, some_topic, (dds::pub::qos::DataWriterQos() << dds::core::
policy::Reliability::Reliable()))
```

Reliability can e.g. be set to *reliable* or *best effort*.

XML Files

QoS settings can be set either programmatically or in XML files. For Matlab, it was decided to use XML files, partially because of the documentation and partially because some setting in the file depends on the IP address and must thus be changed depending on the system. More information on settings QoS (using XML) can be found here: https://community.rti.com/static/documentation/connext-dds/5.2.3/doc/manuals/connext*dds/html*files/RTI*ConnextDDS* CoreLibraries_UsersManual/Content/UsersManual/XMLConfiguration.htm.

The XML file used for local communication looks like this (some comments are taken from RTI's own XML files):

```
<?xml version="1.0"?>
<!--
Description
...
-->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="C:/Program Files/rti_connext_dds-5.3.1/bin/../resource/schema
/rti_dds_qos_profiles.xsd"
     version="5.3.1">
    <!-- ...
        A QoS library is a named set of QoS profiles.
    -->
    <qos_library name="MatlabLibrary">

        <!-- ...
             A QoS profile groups a set of related QoS.
        -->
        <qos_profile name="LocalCommunicationProfile" base_name="BuiltinQosLibExp::Generic.StrictReliable"
is_default_qos="false">
            <participant_qos>
                <!-- Limit discovery to localhost -->
                <discovery>
                    <accept_unknown_peers>false</accept_unknown_peers>
                    <initial_peers>
                        <element>localhost</element>
                        <element>TEMPLATE_IP</element> <!-- This needs to be the IP Adress of the current machine
-->
                    </initial_peers>
                </discovery>
            </participant_qos>
        </qos_profile>

    </qos_library>
</dds>
```

In this example, a library called *MatlabLibrary* is created, which contains a profile called *LocalCommunicationProfile*. Profiles can be applied to DDS entities to set QoS settings. In this case, any DDS participant that uses these settings can only send and receive messages on the same system. *TEMPLATE_IP* must be set to the current IP address of the machine to limit the communication to that address - setting localhost alone would not be sufficient (if the IP is left out, no communication between participants on the same machine can take place at all).

MatlabLibrary::LocalCommunicationProfile can, if the XML file has been imported, be used to configure a domain participant as follows:

```
#include <dds/domain/DomainParticipant.hpp>
#include <dds/core/QosProvider.hpp>
...
dds::core::QosProvider local_comm_qos_provider("QOS_LOCAL_COMMUNICATION.xml");
dds::domain::DomainParticipant hlcParticipant(hlcDomainNumber, local_comm_qos_provider.participant_qos());
```