**Alarms Data Modelling**

In this Alarms data modeling example, we will:

- describe the data model structure

- the use of keyed data and the benefits

**Special Note**

There are 3 examples of code provided to show an Alarms data model.

*Examples #1 and #2*

Within the *src* directory under c++ and c++11, the Alarms\_publisher.cxx provides source code that illustrates how to populate the Alarm Topic and then publish it.

Note that the Alarms\_subscriber.cxx is generated code (from rtiddsgen) that has remained unchanged.

To build the executable for either Windows or Linux OS’s, refer to the README.md file that resides under the c++ or c++11 directories. Cmake is used to create the additional files that are needed prior to compiling, and then is used once again to compile and link the final executable.

*Example #3*

The Python example is complete and requires no further; Alarms\_publisher.py provides the example code required in publishing the Alarms Topic.

**Alarms Data Structure**

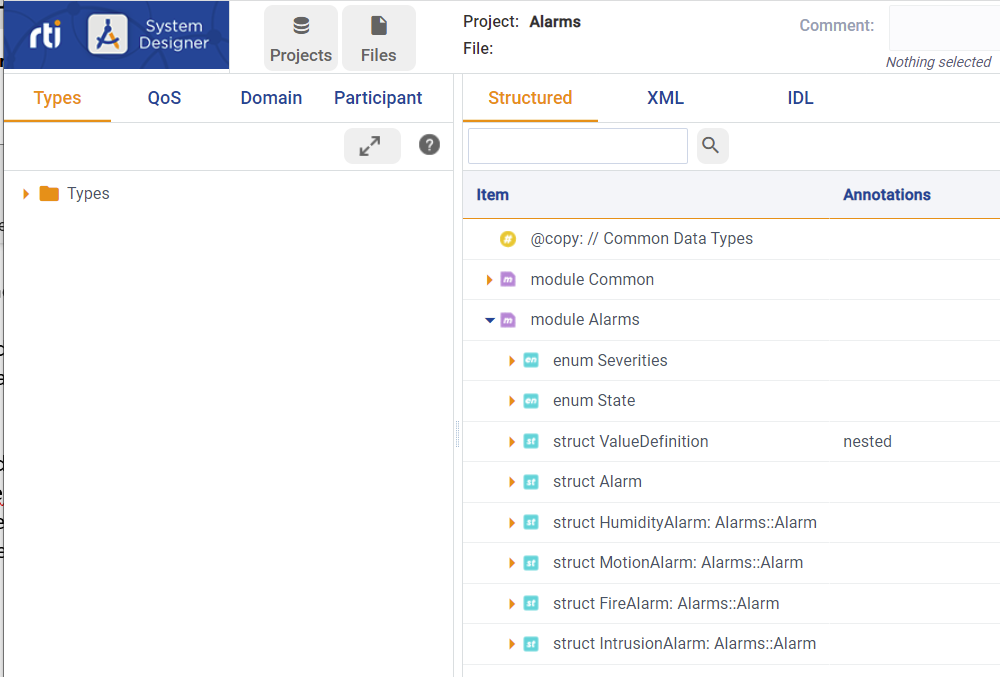
A useful tool provided by RTI is System Designer; it is GUI based and can assist you in creating your data model as well as create QoS settings. It runs in a web browser environment, creates, and continuously saves an XML representation of both data model and QoS while you are designing your system.

This Alarm example makes use of this tool; a representation of the *Alarms data model* is shown below.

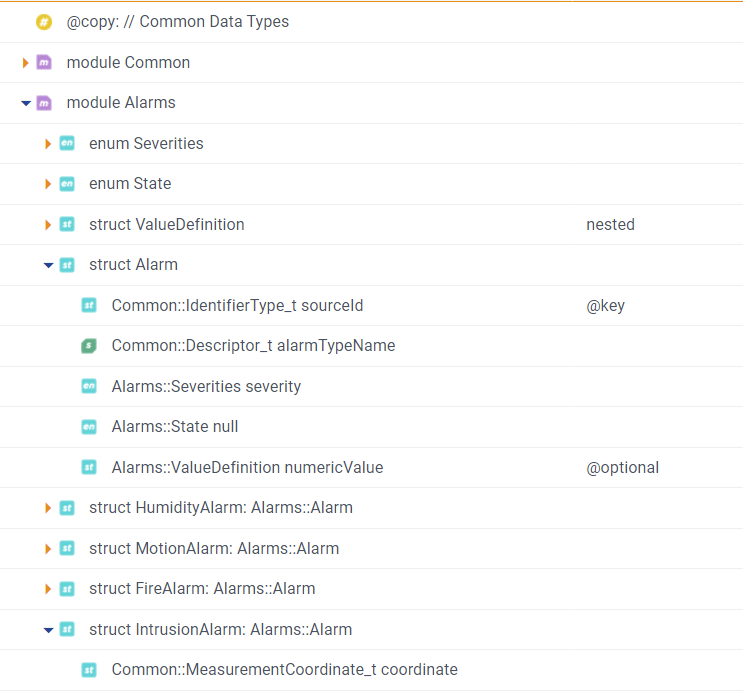
*module Common* contains the many basic data type definitions used in this model and these definitions are then referenced in module Alarms. *module Alarms* is the section that defines the final data structures that become the Topics used by Connext DDS; in this example there are 4 structures supported (listed below) and each structure can become a separate Topic in your application. All you must do is instantiate it and use it.

* HumidityAlarm
* MotionAlarm
* FireAlarm
* IntrusionAlarm

In the code examples presented later, we will be using the IntrusionAlarm as the instantiated Topic.



Referring to the screen shot below outlining the Alarm data model, the *intrusionAlarm* is defined as a specific Topic and will hold *coordinate* data; it also inherits *Alarm* data and this holds *sourceID* (an ID associated with the alarm), TypeName (a description of the alarm), severity (self-explanatory), null (the alarm state), and a *numericValue* (used to further describe the alarm).



**Keyed Data - Instances**

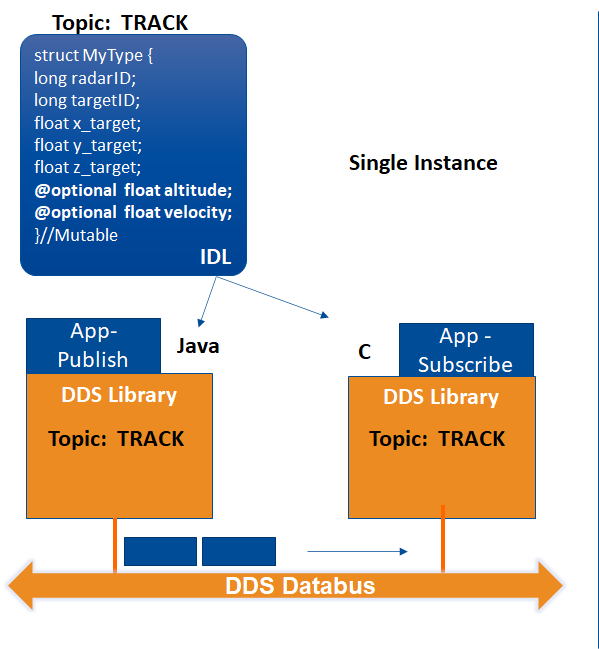
Of note are the tags associated with data:

@key

sourceID is a data field tagged with a *key* value and this is a powerful tool. Any topic that contains the key identifier can be separated into unique *instances*.

*Instances* are flavors of Topics that are treated separately – they are managed in separate queues on the publisher and subscriber sides and can be configured uniquely. Typically, instances are used to convey commonly defined data with each instance being associated with a specific object.

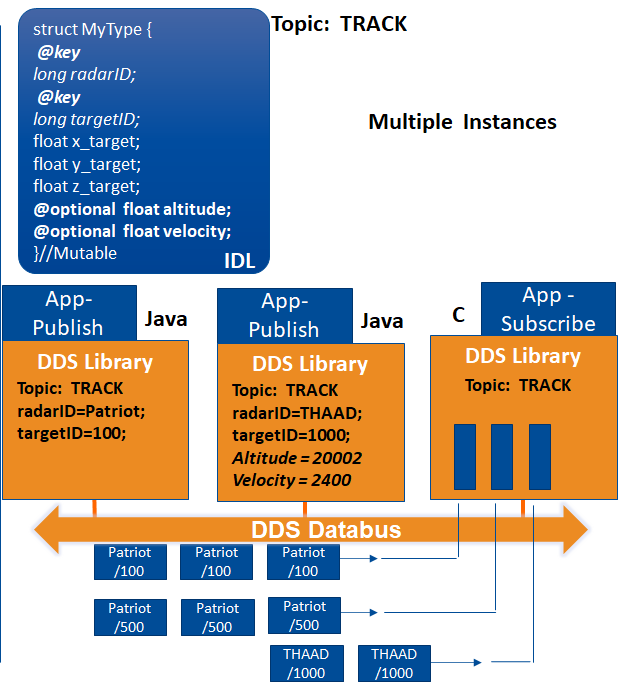
Let’s look at an example as shown below. In a system where there is one Topic and no instances (or keys) defined, that Topic is sent from publisher to subscriber and is managed in a single queue on both sides.



In the second case below, we have defined multiple instances. As a real-world example, we are looking at a defensive missile system with two types of radar – low altitude Patriot missile defense and THAAD, a high-altitude ICBM defensive system. Both radar types can identify a target and can send x, y, and z coordinates of the target to a central control system using a Topic called TRACK. Without Instances, sorting out the incoming Topics and associating target locations with specific targets can be a significant problem and would tax the application software sitting atop Connext DDS.

Instances can be used to segregate targets easily.

The Topic name TRACK is used but *radarID* and *targetID* are used as keyed fields; for each unique combination of keys, one instance is created. In this case the radar type and target ID are unique and will send an instance of TRACK that identifies an incoming missile with x, y and z coordinates. Each instance is managed in a separate queue and presented to the application as a grouping of data that is associated with one target. Quality of Service settings governing each Topic can also be set on a per instance basis.



In terms of coding, and returning to the original data model above, we can create two instances of IntrusionAlarm in the following way.

Using the RTI Code Generator (for C++ generated code) and using the Alarms.xml file generated by System Designer, a code template will be created yielding source that still requires modification to be useful. Within the code you will see the following snippet, creating the data structure called *data* (that is linked to the Topic name which is defined earlier in the code).

Alarms\_IntrusionAlarm \*data = Alarms\_IntrusionAlarmTypeSupport::create\_data();

if (data == NULL) {

return shutdown\_participant(

participant,

"Alarms\_IntrusionAlarmTypeSupport::create\_data error",

EXIT\_FAILURE);

}

This creates only one instance of the Topic. To create a second instance, you must create a separate data structure (called *data2* in this example):

Alarms\_IntrusionAlarm\* data2 = Alarms\_IntrusionAlarmTypeSupport::create\_data();

if (data == NULL) {

return shutdown\_participant(

participant,

"Alarms\_IntrusionAlarmTypeSupport::create\_data error",

EXIT\_FAILURE);

}

You application can then use instances in the following way:

Instance 1 (*data* structure) is populated with data as shown…. Note that the keyed data (sourceID.id and *sourceID*.*resourceID*) are defined and would be associated with an alarm device (its serial number, model number etc.)

data->sourceId.id = 1;

data->sourceId.resourceId = 10;

data->alarmTypeName = "alarm name";

data->severity = Critical;

data->null = Open;

Instance 2 (*data2* structure) is also populated with data (from a different alarm sensor) and its keyed values are different than those of instance 1:

data2->sourceId.id = 2;

data2->sourceId.resourceId = 20;

data2->alarmTypeName = "alarm name";

data2->severity = Critical;

data2->null = Open;

Each of these instance will be managed separately and a unique QoS can be assigned to each instance if required.

**Old School Instance Creation**

Using C++98 code as a reference, the “old school” implementation of Instances creates (calculates) a handle prior to every transmission of the sample. Samples from c++ source code are shown.

1. **The data sample is created:**

//Create memory for Instance #1: this will allocate space for the Alarms data structure

Alarms\_IntrusionAlarm \*data = Alarms\_IntrusionAlarmTypeSupport::create\_data();

if (data == NULL) {

return shutdown\_participant(

participant,

"Alarms\_IntrusionAlarmTypeSupport::create\_data error",

EXIT\_FAILURE);

}

1. **The data sample is populated:**

//Instance #1 - populating the content of this sample

char alarm\_string[] = "alarm name";

data->sourceId.id = 1; //note the keyed value is set to something unique

data->sourceId.resourceId = 10; //note the keyed value is set to something unique

data->alarmTypeName = alarm\_string;

data->severity = Critical;

data->null = Open;

data->numericValue.number = (float) samples\_written;

data->numericValue.Units = Celcius;

1. **The data sample is sent and a handle is calculated each time:**

//Write Instance #1

retcode = typed\_writer->write(\*data, DDS\_HANDLE\_NIL);

if (retcode != DDS\_RETCODE\_OK) {

std::cerr << "write error " << retcode << std::endl;

}

**New School Instance Creation (Greater efficiency)**

For a more efficient method of sending Instances, the handle can be calculated once and used repeatedly. The c++ source code makes use of this method.

1. **The data sample is created, just like before:**

// Create data for writing, allocating all members

Alarms\_IntrusionAlarm \*data = Alarms\_IntrusionAlarmTypeSupport::create\_data();

if (data == NULL) {

return shutdown\_participant(

participant,

"Alarms\_IntrusionAlarmTypeSupport::create\_data error",

EXIT\_FAILURE);

}

1. **But now the Instance handle is created (calculated once) and stored for re-use:**

DDS\_InstanceHandle\_t **fl265Handle** = typed\_writer->register\_instance(\*data);

1. **The data sample is populated, just like before:**

//Instance #1 - populating the content of this sample

char alarm\_string[] = "alarm name";

data->sourceId.id = 1; //note the keyed value is set to something unique

data->sourceId.resourceId = 10; //note the keyed value is set to something unique

data->alarmTypeName = alarm\_string;

data->severity = Critical;

data->null = Open;

data->numericValue.number = (float) samples\_written;

data->numericValue.Units = Celcius;

1. **The data sample is sent but now references the handle that has been created:**

if (typed\_writer->write(\*data, **fl265Handle**) != DDS\_RETCODE\_OK) {

std::cerr << "write error " << retcode << std::endl;

}

Note that the handle created must always be specified along with the correct Instance… in this case it is **\*data.**  (The example code for \*data2 is not shown. )

In cases where the registered Instance will no longer be required, the Instance can be deregistered as follows:

if (typed\_writer->unregister\_instance(\*data, fl265Handle) != DDS\_RETCODE\_OK) {

std::cerr << "Alarms\_IntrusionAlarmTypeSupport::Instance 1 unregistration error " << retcode

<< std::endl;

}

**Python Example**

The Python example has been generated using Connext 7.0.0 “rtiddsgen” and modified. Use the code as-is to run both Publisher (Alarms\_publisher.py) and Subscriber (Alarms\_subscriber.py).