RTI Use Case for Vehicle Tracking Systems

What This Example Does

This example shows three applications. You can run them on the same machine or separate machines in the same network. This example shows radar tracks, but the concepts and the quality of service (QoS) tuning is also applicable for other vehicle-tracking use cases. There are minor differences in the data model, described in the section Data Model Considerations.

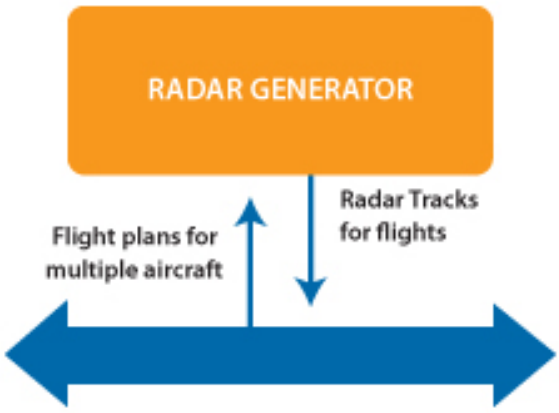
The three applications are:

Flight-plan generator (FlightPlanGenerator):

- Provides flight plans for aircraft

Radar generator (RadarGenerator):

- Provides fast radar track data
- Receives a flight plan from the flight-plan generator
- Associates a flight ID from the flight plan with a radar track if the flight plan is available



Air traffic control GUI (TrackGui):

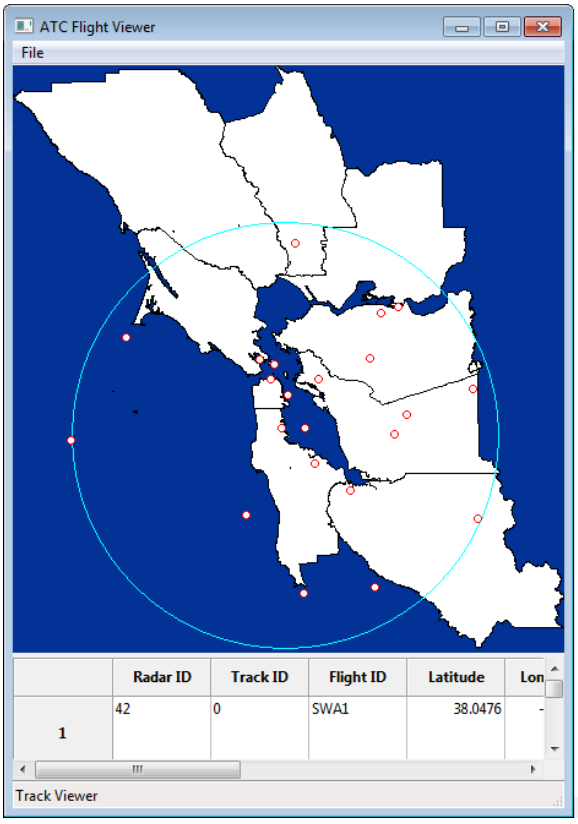
- Receives radar tracks and flight plan
- Displays the radar track
- If the radar track has an associated flight ID, looks up the flight plan and displays the plan with the track

Let's Run the Example

Overview

In this document, we will refer to the location where this example was extracted as EXAMPLE_HOME.

Navigate to the EXAMPLE_HOME\ExampleCode\scripts directory.



In this directory, there are three separate batch files to start the applications. These applications are called:

- FlightPlanGenerator.bat/FlightPlanGenerator.sh
- RadarGenerator.bat/RadarGenerator.sh
- TrackGui.bat/TrackGui.sh

You can run these batch files on the same machine, or you can copy this example and run on multiple machines. If you run them on the same machine, they will communicate over the shared memory transport. If you run them on multiple machines, they will communicate over UDP.

When you run the Flight Plan Generator, it sends 200 flight plans by default. It publishes these immediately, and because of their QoS settings, they remain available to the other applications as they the other applications come online.

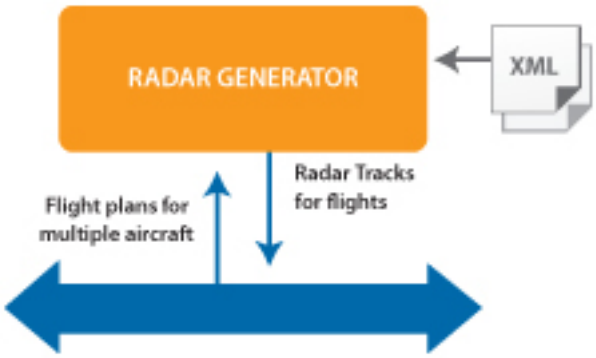
When you run the Radar Generator, it sends 20 radar tracks at startup by default. Every 120 seconds, it adds another radar track. It is configured using QoS to send the track data with the lowest latency. You can modify these values using the command-line parameters. You can also run more than one radar generator application, but you should use the command-line parameters to give each one a different radar ID.

When you run the Air Traffic Control GUI, you can see track data and flight-plan information view. If you are running all the applications on a single machine, this data is being sent over shared memory.

If you have access to multiple machines on the same network, start running these applications on separate machines. Note: If you do not have multicast on your network, see the section Run the Example with No Multicast for details on how to change the configuration to run without multicast.

Configure Throughput and Latency for Your Requirements

RTI Connext DDS allows you to specify QoS configuration in an XML format. This allows you to separate your application logic from your network capabilities and rapidly reconfigure your application for new deployment scenarios.



This example shows you how to configure your application to maximize throughput at the expense of latency, or minimize latency at the expense of throughput.

Individual vehicle or radar tracking applications within a distributed system will have different requirements for data latency or throughput. A collision-avoidance system or a self-defense system may have strict requirements for low latency. A recording or logging system may need to record a high throughput of vehicle position or radar track data, but may not need to receive it with the low latency of your other applications.

Run the Example with Increased Throughput and Increased Latency

By default, the radar generator application runs with the lowest possible latency. To run it with increased throughput at the expense of latency, use the following parameter:

```
scripts\RadarGenerator.bat --high-throughput
```

You can also increase the number of tracks it sends at startup, how often it should create new tracks, the maximum number of tracks it can send at once, and how fast it should update:

```
--start-tracks [number]  Number of tracks the generator should generate
                         at startup
--max-tracks [number]    Maximum tracks the generator sends at once
--run-rate [number]      Run in real time, faster, or slower.  At
                         default rate, all tracks are updated every
                         100ms.  If you set this to 2 the generator
                         will run twice as fast, updating all tracks
                         every 50ms.
```

```
--creation-rate [number] How fast to create new tracks.
```

Run Multiple Radar Generators

The RadarGenerator application is acting as a unique sensor that updates radar positions. As we will discuss later in the Data Model Considerations section, each RadarGenerator application needs a unique ID. So, if you run more than one RadarGenerator application, you should run with the option:

```
scripts\RadarGenerator.bat --radar-id [number]
```

Run the Example with No Multicast

If your network doesn't support multicast, there are two steps you must take:

- Run all applications with the parameter --no-multicast. This causes the applications to load the .xml files that do not depend on multicast in the network.
- Edit the base_profile_no_multicast.xml file to add the address of the machines that you want to contact. These addresses can be valid UDPv4 or UDPv6 addresses.

```
<discovery>
  <initial_peers>
      <!-- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! -->
      <!-- Insert addresses here of machines you want      -->
      <!-- to contact                                       -->
      <!-- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! -->
      <element>127.0.0.1</element>
      <!-- <element>192.168.1.2</element>-->
  </initial_peers>
</discovery>
```

Let's Build the Example

Directory Overview

Inside of EXAMPLE_HOME, the source code is divided into:

1. Documentation in Docs/
2. Source in ExampleCode/
    3. Linux makefiles in make/
    4. Windows solution files in win32/
    5. Source in src/
        - RTI Connext DDS interface data-type descriptions in Idl/. This describes the data types sent over the network.

- RTI Connext DDS QoS configurations in Config/.
- RTI Connext DDS infrastructure code that is used by all applications in CommonInfrastructure/. This is the code that all applications call to start using RTI Connext DDS to send data.
- RTI Connext DDS generated type code in Generated/
- Application-specific RTI Connext DDS publishing and subscribing code in FooInterface.h and FooInterface.cxx.

Building the Example

On all platforms, the first thing you must do is set an environment variable called NDDSHOME. This environment variable must point to the ndds.5.x.x directory inside your RTI Connext DDS installation. For more information on how to set an environment variable, please see the RTI Core Libraries and Utilities Getting Started Guide.

Windows Systems

On a Windows system, start by opening up the win32\AirTrafficExample-<compilerver>.sln file.

This code is made up of a combination of libraries, source, and IDL files that represent the interface to the application. The Visual Studio solution files are set up to automatically generate the necessary code and link against the required libraries.

You can also watch a video online showing how tot build: How to Build the Air Traffic Control Example on Windows.

Linux Systems

To build the applications on a Linux system, change directories to the ExampleCode directory and use the command:

gmake –f make/Makefile.<platform>

The platform you choose will be the combination of your processor, OS, and compiler version. This version of the example only supports i86Linux2.6gcc4.5.5

You can also watch a video online showing how to build: How to Build the Air Traffic Control Example on Linux

Under the Hood

Data Model Considerations

When modeling data in DDS, one of the biggest considerations is “what represents a single element or real-world object within my data streams?” In the case of vehicle tracking, you can generally assume that each vehicle should be represented as a unique object.

In DDS, these unique real-world objects are modeled as instances. Instances are described by a set of unique identifiers called keys, which are denoted by the //@key symbol.

So, in our example, we use a trackId as a key field:

```
        long trackId; //@key
```

There is one wrinkle to our assumption that each vehicle should be represented as a unique object: What if the same vehicle is being tracked by multiple sensors?

Often you want to maintain updates from each sensor separately. In DDS, this means that instead of each vehicle being the unique real-world object, *the combination of the vehicle and the sensor* becomes the unique real-world object. To support this, the sensor ID also becomes a key field:

```
        long radarId; //@key
        long trackId; //@key
```

If you can assume that there is one sensor per vehicle, such as a GPS that is updating the position of an emergency vehicle, you do not need to worry about a sensor unique ID being part of the data. In that case, the instance is the vehicle.

Configuration Details: XML Configuration for Radar Throughput

Since a major component of building a radar- or vehicle-tracking system is performance tuning, we'll talk about that first, before going into too much detail about the code itself.

The source code loads a series of XML files that it uses to configure the delivery and resource characteristics for the data. The .xml files are in the Config directory; they specify the communication characteristics for the data, such as whether the data is reliable. XML QoS profiles can inherit from each other.

Multicast

Multicast is enabled in the multicast_base_profile.xml file.

In the example, the use of multicast is encapsulated in a QoS profile called "OneToManyMulticast." This is disabled by default, because high-throughput multicast data may slow down your network if you are on a wireless LAN. However, you can enable it for testing by editing the multicast_base_profile.xml file. Multicast is used by default for discovery, so if your network does not support multicast, see the section Run the Example with No Multicast for details on how to change the configuration to run without multicast.

```
<qos_profile name="OneToManyMulticast">
    <datareader_qos>
        <!-- Uncomment this to enable user data over multicast.  This is
            commented out for systems that do not have multicast, or
            with switches that block some multicast traffic -->
        <!--<multicast>
            <value>
                <element>
                    <!- - Must be a valid multicast address- ->
                    <receive_address>239.255.5.1</receive_address>
                </element>
```

```
            </value>
          </multicast>-->
      </datareader_qos>
  </qos_profile>
```

Batching

Batching small data enables throughput at the expense of latency. The Radar's default profile "LowLatencyRadar" does not use batching. Batching is enabled in the "HighThroughputRadar" configuration.

```
<batch>
   <enable>true</enable>

   <!-- If the batch hits 1024 bytes, flush to the network -->
   <max_data_bytes>1024</max_data_bytes>

   <!-- You can decide on the maximum amount of additional latency
       you are willing to sacrifice for better throughput. -->
   <max_flush_delay>
     <sec>0</sec>
     <nanosec>200000000</nanosec>
   </max_flush_delay>
 </batch>
```

Common DDS Infrastructure for all Applications

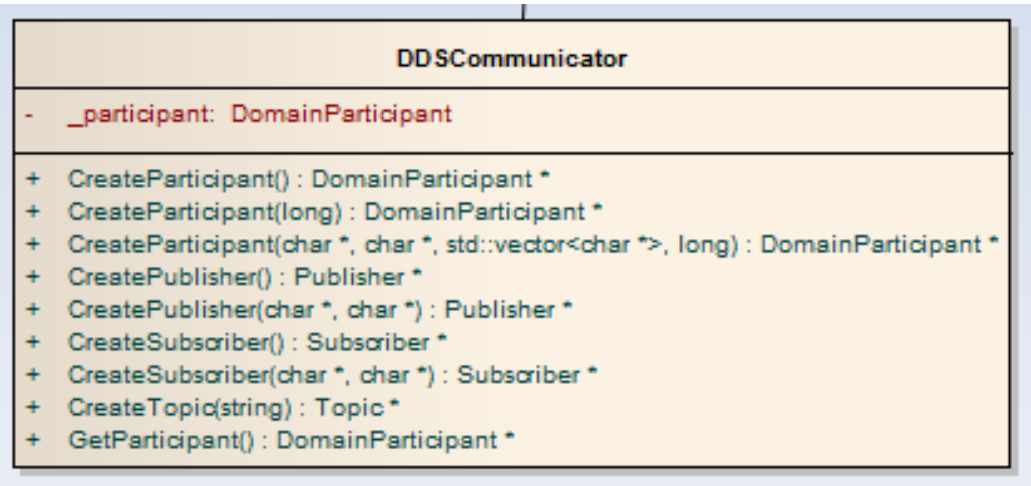


Figure 1: DDS Communicator class. Contains code for creating all the basic RTI Connext DDS objects that are used for network communications and automatic discovery.

Now let's look at the code that you will write once and use in every DDS application. The code in CommonInfrastructure/DDSCommunicator.h/.cxx creates the basic objects that start DDS communications. The DDSCommunicator class encapsulates the creation and initialization of the DDS DomainParticipant object.

All applications need at least one DomainParticipant to discover other RTI Connext DDS applications and to create other DDS Entities. More information on what a DomainParticipant does is described in this glossary entry on RTI's Community Portal. Typically, an application has only one DomainParticipant.

In the source code, you can see this in the class DDSCommunicator, in CommonInfrastructure/DDSCommunicator.cxx:

```
DomainParticipant* CreateParticipant(
        long domain, char *participantQosLibrary,
        char *participantQosProfile)
{
        _participant =
                ThePariticipantFactory->create_participant_with_profile(
                        domain, participantQosLibrary,
                        participantQosProfile, NULL,
                         STATUS_MASK_NONE);
   ...
}
```

The DomainParticipant's QoS is loaded from one or more XML files. The profile to load is specified by the participantQosLibrary and participantQosProfile. The full list of DomainParticipant QoS is described on RTI's Community Portal in the HTML API Documentation.

Applications

RadarGenerator (C++):

This application sends and receives data over the network. The code to create the application's DDS interface is in the class RadarInterface. This class is composed of three objects:

- DDSCommunicator
- RadarWriter
- FlightPlanReader

The DDSCommunicator object creates the necessary DDS Entities that are used to create the RadarWriter and FlightPlanReader Entities.

The RadarWriter class is a wrapper around a DDS DataWriter that sends radar data. The FlightPlanReader class is a wrapper around a DDS DataReader that receives flight plan data.

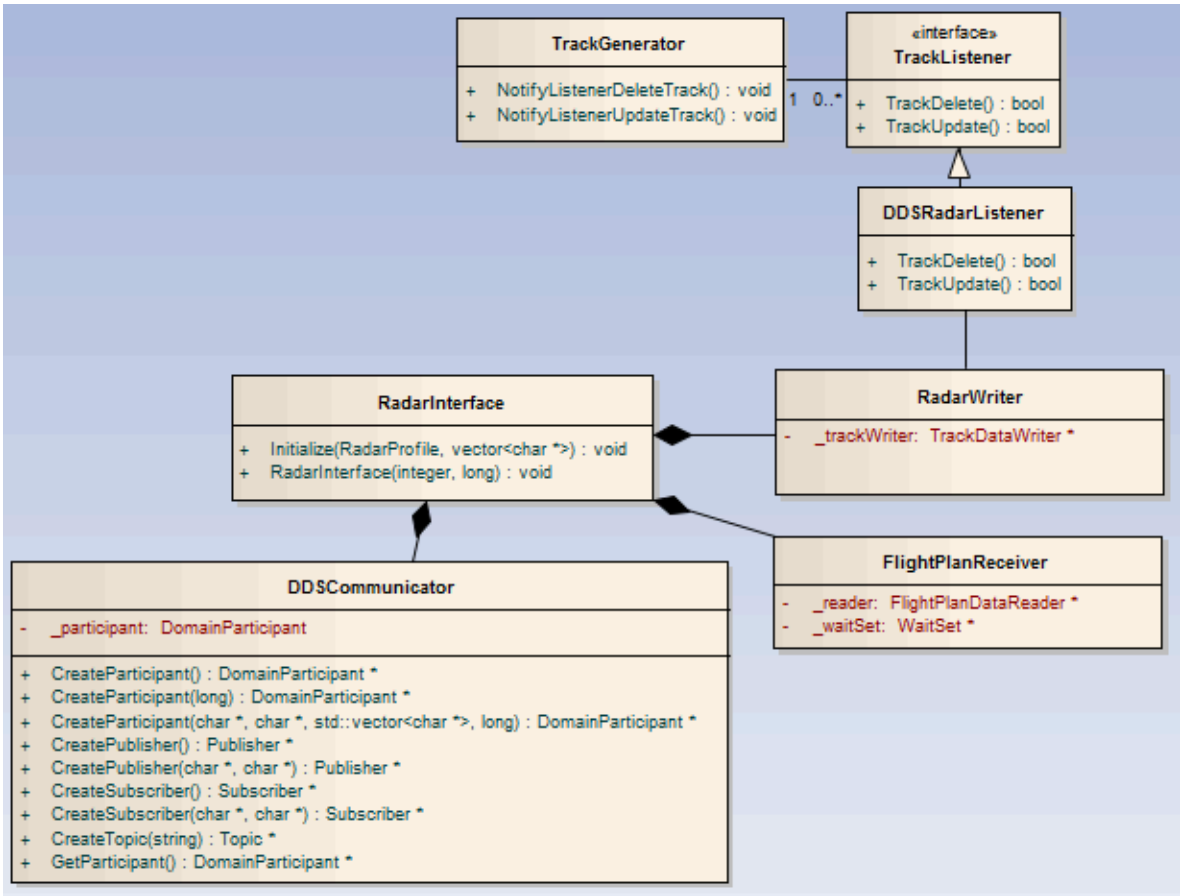


Figure 2: RadarGenerator Application Major Classes

Generating Data

The TrackGenerator.cxx/TrackGenerator.h files are generating the radar track data for this example. Most of this code is not necessary for understanding how to use RTI Connext DDS, but exists to create interesting-looking data for the application. This code is designed to represent a third-party library with no dependency on RTI Connext DDS.

This code is responsible for:

- Creating a specified number of tracks at startup
- Publishing tracks at a specified rate. Setting the rate to 1, it will publish all tracks and then sleep for 100ms. If you increase the rate to 2, it will sleep for 50ms between sending updates of all tracks, etc.
- Creating new tracks after a specified number of seconds in "real time." If you increase the run rate, the number of seconds between creations of new tracks will decrease.
- Not creating tracks beyond the specified maximum number of tracks it can provide.
- Maintaing a list of flight plan data, and adding the flight ID to the track data if it is available.

The parts of this code that are interesting for RTI Connext DDS usage are the `AddFlightPlan`, `CorrelateFlightPlanWithTrack`, and `NotifyListenersUpdateTrack/NotifyListenersDeleteTrack` calls. The first two calls are important because they show another thread (the main thread) receiving data from RTI Connext DDS, and adding this flight plan data to the Radar Generator using `AddFlightPlan`. A real application would be correlating this flight plan data and the radar track data, but this application

simply associates the first track in its queue with the first flight plan available in the `CorrelateFlightPlanWithTrack` call. This application uses an observer pattern to notify one or more listeners that a track has been updated or deleted in the `NotifyListenersUpdateTrack` and `NotifyListenersDeleteTrack` calls.

Sending Data

The RadarInterface.cxx file contains the code for sending the generated track data over the network.

The application publishes the track data in the PublishTrack() call. The RTI Connext DDS call that actually sends data over the network is _trackWriter->write(track, handle). This call accepts the data that will be sent over the network, and a handle to the data. In this example, we pass in a NIL handle. However, you can get better performance in some cases by pre-registering your data and using the handle.

The application publishes a track drop message in the DeleteTrack() call. It does this by calling _trackWriter->dispose(track, handle).

Radar Data Model - Data Type

Radar is modeled in the AirTrafficControl.idl file. RTI Connext DDS uses IDL, the Interface Definition Language defined by the OMG to define language-independent data types. More information about IDL can be found in the RTI Connext Users' Manual. The IDL type definition can be found in the file src/Idl/AirTrafficControl.idl. This can also be found in the SharedDataTypes Visual Studio project on Windows.

```
struct Track
{
    long radarId; //@key
    long trackId; //@key

    ...
    FlightId flightId;

    double latitude;
    double longitude;
    double altitude;

    ...
};
```

This is modeled as very simple data, with only a few fields. The most important thing to notice is that the track ID and a radar ID are marked with the tag //@key. This indicates that these IDs make up the unique identifier of an individual track. The flight ID may or may not be available when the track is created, so it is not part of the unique ID of a track. More information about uniquely identifying elements of your data can be found in this best practice on RTI's online Community Portal.

By telling the middleware that we are representing unique real-world objects within our Topic, we allow the middleware to do smart things with our data, such as keeping a separate cache for each of our unique objects. This will be covered more in the section Radar Data Delivery Characteristics (QoS).

The data also has a latitude, longitude, and altitude. We could optionally add more fields, such as bearing and speed.

Radar Data Model - Topic

The radar data can be represented as a single Topic. It is a best practice to define the Topic name inside your IDL because this prevents applications from hard coding the Topic name string, and because the Topic name is part of the interface.

```
const string AIR_TRACK_TOPIC = "AirTrack";
```

Radar Data Delivery Characteristics (QoS):

By default, radar data is sent rapidly, so it could potentially be sent without reliability enabled. However, it is important that the receiving application receives the last update of each track.

To support this, we have enabled **reliability** in this example, combined with a **history depth of 1**. As we discussed above, each track is modeled as a unique instance. Modeling data as instances in combination with reliability and a history depth of 1 means that:

- The application has a single space in its queue for each radar track, and
- It will reliably deliver whatever is currently in each space in its queue.

By doing this, we ensure that the last piece of data that is sent (the last update or the drop message) will be delivered. The Reliability and History QoS are enabled in XML. Note that these settings must be enabled on both the DataWriter and the DataReader to ensure reliable delivery.

```
<reliability>
   <kind>RELIABLE_RELIABILITY_QOS</kind>
</reliability>
<history>
   <kind>KEEP_LAST_HISTORY_QOS</kind>
    <depth>1</depth>
</history>
```

Beyond this, the radar data is tuned for low-latency, high-throughput data. As described above, in this example we can further increase throughput at the expense of latency by enabling batching.

Receiving Data

This application is receiving Flight-plan data as well as sending Radar Track data. The main function in RadarGenerator.cxx is calling `WaitForFlightPlans()` in a loop, waiting to be notified that flight-plans are available. When it is notified of flight-plans, it adds them to the RadarGenerator.

The code for receiving the Flight-Plan data from the middleware is in the `FlightPlanReader` class, in RadarInterface.cxx. This class creates a DDS WaitSet, which allows it to block an application thread until Flight-Plan data becomes available from the middleware.

The code for waiting for Flight-Plan data is in the method `WaitForFlightPlans()`. The code for removing Flight-Plan data from the middleware queue is in the `ProcessFlightPlans()` method.

Flight-Plan Generator (C++):

This application sends flight-plan data over the network, and is the simplest of the three applications in this example.

The code to create the application's DDS interface is in the class FlightPlanPublisherInterface. This class is directly responsible for writing data. This class also contains a DDSCommunicator that is used to create a DomainParticipant and the FlightPlanDataWriter that actually sends data over the network.

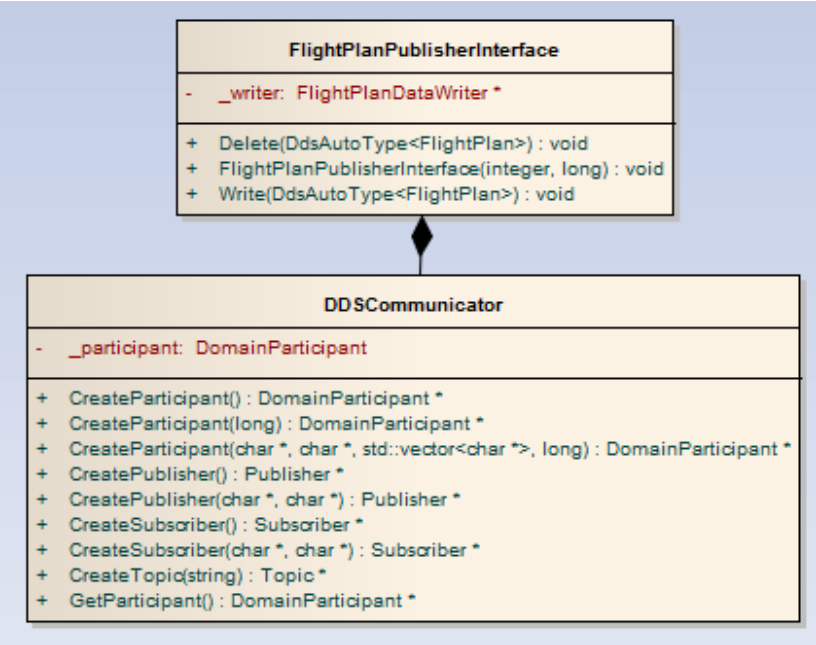


Figure 3: Flight Plan Generator Major Classes

Flight-Plan Data Model

The flight plan is modeled in DDS as Occasionally Changing State Data, which has the following characteristics:

- It is updated only when the state of some object changes-in this case, the flight plan, which may be published or updated according to conditions
- That object's state is not constantly changing
- Other applications want to know the current state of each object—even if it was published before they started up

Flight-Plan Data Model - Data Type

The data type is modeled in the AirTrafficControl.idl file. You can see this data type using the Analyzer tool. The FlightPlan data type is more complex than the Track data type, including enumerations, strings, and a sequence of alternate aerodromes.

```
struct FlightPlan
{
  // Up to seven characters that represent the unique flight ID
  FlightId flightId; //@key
```

```
  // flight rules (enumeration -> IFR, VFR, initially IFR followed by
changes, initially VFR followed by changes
  FlightRulesKind flightRules;

  // type of flight (enumeration -> scheduled air service, non-
scheduled air transport, general aviation, military, etc.)
  FlightTypeKind flightType;

  // equipment and capabilities (enumeration -> NO_COMMS,
STANDARD_COMMS, etc.)
  EquipmentKind equipmentType;

  ...
};
```

Flight-Plan Data Model - Topic

The flight plan data can be represented as a single Topic. It is a best practice to define the Topic name inside your IDL because the Topic name is part of the interface.

```
const string AIRCRAFT_FLIGHT_PLAN_TOPIC = "FlightPlan";
```

Representing Unique Flights

State data represents the state of some element or object in the real world - in this case, the flight plan for a particular flight. In DDS, real-world objects are modeled as instances. (See the RTI's online glossary for more detailed definitions of keys and instances).

Instances are described by a set of unique identifiers called keys. In this case, the key is the unique flight identifier - a string with a maximum of seven characters that includes the airline ID and the flight number.

```
struct FlightPlan
{
  // Up to seven characters that represent the unique flight ID
  FlightId flightId; //@key
  ...
};
```

Flight-Plan Data Delivery Characteristics (QoS)

This data must be sent reliably because it is not being sent all the time. This is configured in the XML. Note that reliability must be enabled in both the <datawriter_qos> and <datareader_qos> sections.

```
<reliability>
   <kind>RELIABLE_RELIABILITY_QOS</kind>
</reliability>
```

One of the benefits of using RTI Connext DDS for sending state data is the ability to send data as it changes, but to ensure that any interested late-joiner will receive the data as soon as it starts up. This means that the flight plan can be sent as soon as it is available or updated and if a

particular application has not been started yet, it will still receive the flight plan as soon as it starts. To enable this, data must be sent with a transient-local or higher level of durability.

```
<durability>
   <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
</durability>
```

In order to deliver only the most recent update to any flight plan, this XML configures the history cache on the DataWriter to maintain a history of size one for each flight plan instance.

```
<history>
   <kind>KEEP_LAST_HISTORY_QOS</kind>
    <depth>1</depth>
 </history>
```

This data does not need to be tuned for extreme throughput, but we do tune it for fast reliability. Details are described in the FlightPlanStateData XML profile.

Air Traffic Control GUI (C++)

The Air Traffic Control GUI uses a Model-View-Presenter architecture to receive updates from the network and then present them to the UI as they happen. The Model and Presenter pieces of this code show how to use RTI Connext DDS to receive multiple data streams over the network, and to correlate them into data used by the GUI classes.

Receiving Track Data

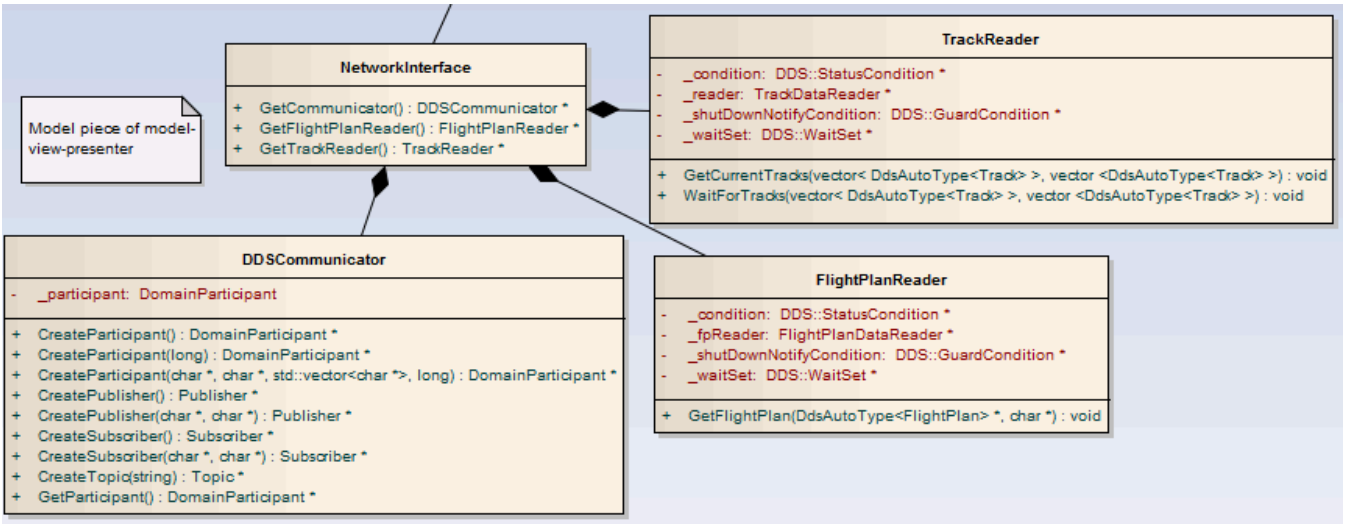


Figure 4: The "Model" piece of the GUI, including all data received from the network from the TrackReader and FlightPlanReader classes

The GUI application receives flight plans from the Flight-plan Generator application and receives tracks from the Track Generator application. It uses two DDS DataReaders to receive the data.

This application does not need the lowest-possible latency, so the presenter periodically polls for Track updates by calling `GetCurrentTracks()` on the TrackReader:

```
reader->GetCurrentTracks(&tracks);
```

The `GetCurrentTracks()` call will retrieve all the alive track data currently in the TrackDataReader's queue by calling the RTI Connext DDS API:

queue, and actually removes them as it processes them:

```
// This reads all ALIVE track data from the queue, and loans it to the
// application in the trackSeq sequence.  See below where you must return
// the loan.
DDS_ReturnCode_t retcode = _reader->read(
                  trackSeq, sampleInfos,
                  DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
                  DDS_ANY_VIEW_STATE, DDS_ALIVE_INSTANCE_STATE);
```

This call retrieves the ALIVE instances from the TrackDataReader's queue, but leaves the data in the queue instead of taking it out. When it accesses that data, it is getting only a single update for each track. This is guaranteed because the QoS for the TrackDataReader specifies that it has a history depth = 1. This means that there will be at most one update to the Track data in the queue for each unique Track instance. In other words, if you are tracking 500 aircraft, there will be a single update for each of those 500 aircraft in the queue.

After processing the ALIVE instances, the TrackReader queries for NOT_ALIVE instances in the queue, and actually removes them as it processes them:

```
// Now we access the queue to look for notifications that tracks have been
// deleted.  We do not leave this in the queue, but remove the deletion
// notifications.
retcode = _reader->take(
                  trackSeq, sampleInfos,
                  DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
                  DDS_ANY_VIEW_STATE,
                  DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE |
                  DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE);
```

Receiving Flight-Plan Data

The second piece of the network code that is interesting is in the `FlightPlanReader` class. The UI needs to display the FlightPlan data that is associated with a particular RadarTrack. To allow this, the `FlightPlanReader` has a method that queries for a single FlightPlan using the `flightId`. Since the `FlightPlan` contains a `flightId` field that should match the `flightId` field in the `RadarTrack`, the higher layers can retrieve the `RadarTrack` first, and then call `GetFlightPlan()` to retrieve the flight plan associated with the track.

Since the `flightId` is the key field of the data – in other words, it is the unique identifier of an individual `FlightPlan` instance – the code retrieves a single `FlightPlan` by using the RTI Connext DDS instance information.

The steps it uses to look up a particular flight plan are:

1. Use flight plan unique ID (key field), which is available as a part of the radar track data
2. Create a dummy flight plan object with just the unique ID set
3. Call `lookup_instance` on the flight plan DataReader to map from the unique ID to the instance handle.
4. If the instance handle is not Nil, call `read_instance()` to retrieve the updates in the queue for just that flight plan. Note that due to the QoS setting history depth = 1, there will only be a single update in the queue for each flight plan.

```
// Create a placeholder with only the key field filled in.  This will
// be used to retrieve the flight plan instance (if it exists).
DdsAutoType<FlightPlan> flightPlan;
strcpy(flightPlan.flightId, flightId);

// Look up the particular instance
const DDS_InstanceHandle_t handle =
        _fpReader->lookup_instance(flightPlan);

// ... Check if the instance is null

// Reading just the data for the flight plan we are interested in
_fpReader->read_instance(flightSeq, infoSeq,
                         DDS_LENGTH_UNLIMITED,
                         handle);

// ...
_fpReader->return_loan(flightSeq, infoSeq);
```

Notifying the UI of Data Arrival

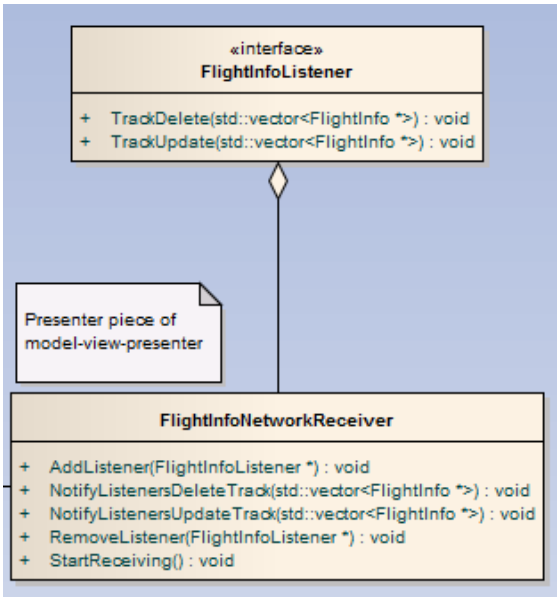


Figure 5: Classes representing the "Presenter" part of the Model-View-Presenter architecture. Responsible for querying for data updates from the network and notifying the UI via listeners that there are changes to Tracks.

The `FlightInfoNetworkReceiver` class is the Presenter of the Model-View-Presenter pattern. It can be thought of as the glue between the data arriving on the network and the UI that displays the track updates. The `FlightInfoNetworkReceiver` class periodically polls for updates to flights, and then notifies the UI that flights have been updated or deleted. This could be changed to receive notifications as data arrives, but this is not necessary for a UI application.

The FlightInfoNetworkReceiver's main purpose is:

1. Poll for updates to Track data using the TrackReader class
2. For each Track data instance, call GetFlightPlan() on the FlightPlanReader to retrieve the flight plan with a flightId that matches the flightId in the track.
3. Assemble the Track and the FlightPlan into a single object that is used by the UI classes to present the appropriate information

If you have an application that needs to receive data with extremely low latency, you can use a listener rather than polling. The code for accessing data is identical, but instead of using your own thread and waiting on a WaitSet, or polling in a loop, you can install a listener that will notify you of data availability. To see an example of this, you can generate a hello world application using the rtiddsgen code generation tool.

The `FlightInfoNetworkReceiver` class follows an Observer pattern, with one or more listeners that wait for updates to track data. In this example there are two Observers: the `TrackPanel` and the `TablePanel`. The `FlightInfoNetworkReceiver` notifies those two classes about the latest data, and they update the UI view based on these notifications.

Displaying the Data in the UI

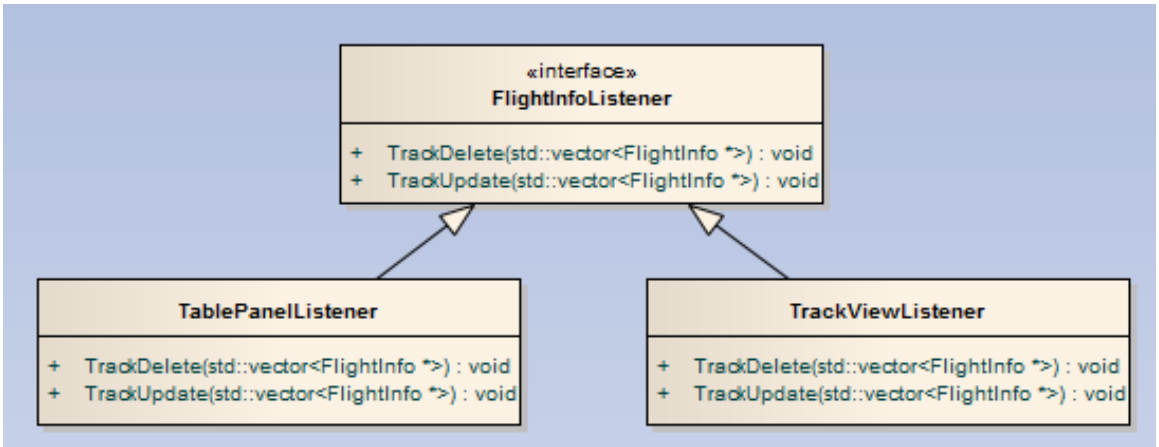


Figure 6: The GUI components are updated by the TablePanelListener and the TrackViewListener

The UI uses wxWidgets to display tracks as circles that approach SFO. The code in the TrackGUi.cxx file takes the incoming track data and displays it on a map. This code is entirely independent of RTI Connext DDS, and is not required for receiving data over the network. Most of this code is related to drawing the map, converting between coordinate types (lat/long, UTC, and screen coordinates), and refreshing the table view.

Next Steps

Extra Credit: Record and Replay

If you want to experiment further, you can use the Record/Replay tool to record live track data and replay it later. This is useful in a real system because it allows you to:

- Record what is happening in your distributed system to post-process the data and look for anomalies.
- Record and replay live data to test against real-world scenarios.
- Stress test your system by replaying data faster than it was originally recorded.

Extra Credit: View the Application using Tools

To get an idea of what these applications are creating, you can use the RTI Analyzer tool to see:

- What Topics they are sending and receiving on the network
- What the structure of the applications' data looks like

The RTI Analyzer tool must be configured to communicate in the same domain that your application is using. This video shows how to get started using RTI Analyzer and how to view your data types.

You can also watch a video online showing how to view the data using RTI Analyzer: Viewing the Air Traffic Control Example applications with the RTI Analyzer Tool

Extra Credit: Change the domain in code.

You can change the domain used by RadarGenerator by changing this code snippet in RadarInterface.cxx:

```
if (NULL == CreateParticipant(0, qosFileNames, libName, profileName))
```

Change the '0' to a different number between 0 and 232 and now this application is in a different domain. Run the application. Now it will no longer discover or communicate with the other applications.

Separation of domains is useful if you have subsystems in your distributed system that should not communicate. This is also useful if you have multiple developers working on the same project and you do not want their applications or tests interfering with each other.
Change the application back to domain 0 and rebuild.

You can also watch a video online showing changing the Domain and Viewing in Analyzer.

Join the Community

If you have questions or you would like to discuss variations of this use case, please post questions on the RTI Community Forum.

Love RTI? Too much free time? This use case example is also available on GitHub. You can contribute to this use case, or to our feature examples. Instructions on how to contribute to our projects are available on this page.

Download RTI Connext DDS

Check out more of the RTI Connext product family and learn how RTI Connext products can help you build your distributed systems. Download the free trial.