RTI Use Case for Streaming Video Data

What This Example Does

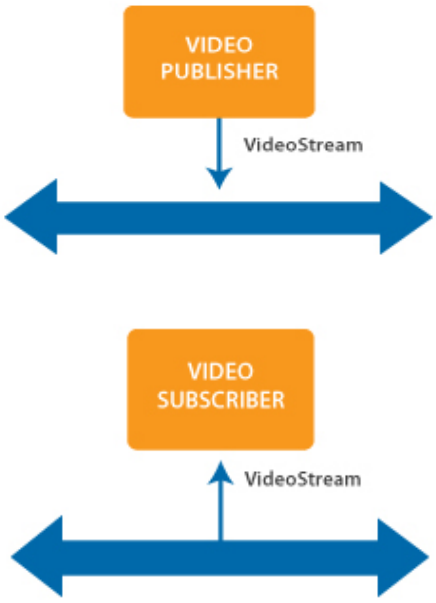
This example opens a video and sends that data stream from a single Video Publisher application to one or more Video Subscriber applications which display the video. By default, this demo sends data over unicast, but you can use command-line parameters to stream the video over multicast to multiple subscribers at the same time. This provides a benefit over TCP because RTI Connext DDS can provide reliability over UDP multicast, allowing for high-efficiency streaming of video data to many subscribers at once.

Video Publisher:

- Discovers Video Subscribers
- Checks metadata about the stream to see if the Video Subscriber supports compatible codec(s)
- Publishes streaming video data if one or more compatible subscribers are online

Video Subscriber

- Configures metadata with descriptions of supported codec(s)
- Subscribes to streaming video
- Displays streaming video



Let's Run the Example

Download RTI Connext DDS

If you do not already have RTI Connext DDS installed, download and install it now. You can use a 30-day trial license to try out the product. Your download will include the libraries that are required to run the example, and tools you can use to visualize and debug your distributed system.

Linux Only: Download the GStreamer Framework

On CentOS, you can install GStreamer and its development plugins:

yum install gstreamer gstreamer-devel gstreamer-plugins-base-devel

Run the Example

These examples can be run on the same machine or on multiple machines. The video file will be loaded by the Video Publisher application and sent to one or more Video Subscriber applications. If the applications are running on the same machine, they are sending data over shared memory. If they are running on multiple machines, they are sending data over the network.

To Start the Video Publisher Application:

Run the batch file: scripts\VideoPublisher.bat or the shell script: scripts/VideoPublisher.sh

To Start the Video Subscriber Application:

Run the batch file: scripts\VideoSubscriber.bat or the shell script: scripts/VideoSubscriber.sh

To Start the Video Subscriber With Multicast Enabled:

Optionally, you can enable the video data to be sent over multicast. To do this, run the batch file: scripts\VideoSubscriber.bat --multicast or the shell script: scripts/VideoSubscriber.sh --multicast

Example Output

When you start the Video Publisher example, you will see the following text in the console:

```
--- Starting publisher application. ---
This application will read a video file, and publish it over RTI Connext DDS
middleware
Waiting for a compatible video subscriber to come online
Waiting for a compatible video subscriber to come online
Waiting for a compatible video subscriber to come online
```

Once it discovers a compatible Video Subscriber application, it prints:

```
Initializing and starting video source
Video Source: VideoSource started
```

When you start the Video Subscriber example, you will see the following text in the console:

```
--- Starting subscriber application. ---
This application will subscribe to a video feed over RTI Connext DDS
middleware
```

Once it discovers a Video Publisher application, it will print the following text and you should see a video playing:

```
. . . . . .
. . . . . .
```

Let's Build the Example

Directory Overview

In the installation directory (EXAMPLE_HOME), the files are divided into:

- Further documentation in Docs/
- Source code in ExampleCode/
    - Linux makefiles in make/
    - Windows solution files in win32/
    - 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.
        - 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 the EXAMPLE_HOME\ExampleCode\win32\StreamingVideoExample-<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.

For more information, see this video showing how to build the 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. Currently, this example only supports one platform: i86Linux2.6gcc4.5.5.

For more information, see this video showing how to build the example on Linux.

Under the Hood

Data Model and QoS Considerations

Streaming Video Data Type

Video streams consist of a combination of video images and metadata such as the video format, video width, and video

height. These can be divided into multiple streams, or combined into a single stream. In this example, the image data and the metadata are encoded in the same stream, or Topic. Metadata updates and video image updates have different sizes, so this example uses an octet sequence to represent the updates.

RTI Connext DDS uses IDL to describe the format of data that will be sent over the network. The IDL that is used to encode the video stream looks like this:

```
struct VideoStream
{
    // This allows a subscriber to differentiate between different
    // video publishers that are publishing the same data on the same
    // Topic.
    long stream_id; //@key

    // Some video formats may require metadata to be sent with the
    // binary video data, such as flags or a frame sequence_number.
    unsigned long flags;
    unsigned long sequence_number;

    // 1024 * 1024 is the maximum frame size
    // Data will be allocated at this size, but
    // only the actual size of the frame will
    // be sent.
    sequence <octet, 1048576> frame;
};
```

The key piece is that a variable-length binary sequence is defined that contains the video frames:

```
sequence <octet, 1048576> frame;
```

Many video codecs have variable sizes of data – some packets contain a full keyframe, while other packets contain smaller updates to the image that is shown on-screen. DDS sequences support this variable-length data. RTI Connext DDS will allocate the full maximum size of one or more VideoStream structures up front (allocating a number up to the size of the queue, which is controlled by QoS described below). However, if a particular frame is smaller than the maximum, only the data in that frame is sent over the network. This combination of preallocating the data to a maximum size and sending only the actual data size leads to low-latency performance while conserving bandwidth.

Writing and Reading Streaming Video Data

These applications have a simple data interface. The Video Publisher application has a single DDS DataWriter object, which is an object that is associated with a single Topic and data type, and sends streaming video data over the network, or shared memory. The Video Subscriber has a single DDS DataReader object, which is an object that is associated with a single Topic and data type, and receives streaming video data over the network. A DataWriter and DataReader must have a matching Topic to communicate
Most applications both send and receive data using multiple DataWriters, DataReaders, and Topics, but these simple applications are only sending or receiving streaming video data on a single Topic.

This data is sent as a single stream, with no reliability enabled. This means that when data is sent from the Video Publisher to the Video Subscriber, it doesn't resend data if it is dropped over the network. If there is a network disruption, any data that was sent during that time is not stored or re-sent by the middleware.

Discovering Video Formats

Not all video feeds will necessarily be readable by all applications. This application sends meta-information about the video codecs that are used to encode the data, and uses that to decide whether to send data to a particular application. It does this by using a QoS policy that allows applications to send metadata during the RTI Connext DDS discovery process.

The Video Publisher application opens a file in WebM video format – a royalty-free, open video compression for use with HTML5 video. The WebM file format contains up to three streams: video, audio, and subtitles. After opening the file, the application demuxes the WebM format and removes just the video data for streaming. It sends just the video part of the file in a VP8 video compression format. This video is built using GStreamer, a framework for doing various processing of media data.

The Video Subscriber application is expecting a VP8 stream with a size of 640x360 pixels. This information is part of the metadata that the Video Subscriber sends over the network. If the Video Publisher is providing a video that is a different format or size, it will not send to the Video Subscriber.

The Video Subscriber configures this metadata when it creates the DataReader object. The DataReader in the Video Subscriber application is configured with a copy of this metadata in the USER_DATA QoS. The USER_DATA QoS allows the application developer to send application-specific data as a part of discovery.

```
DDS_DataReaderQos readerQos;
 . . .
 readerQos.user_data.value.from_array(
    reinterpret_cast<const DDS_Octet *>(videoMetadata.c_str()),
      videoMetadata.length());
```

This metadata about the supported codecs is a string that the application code queries from the GStreamer framework it is built on.

On the Video Publisher side, the application installs a listener on the DomainParticipant, so it can be notified when the DomainParticipant discovers DataReaders. In the listener, the application checks whether the discovered DataReader has put data into the USER_DATA QoS and whether that data describes a codec that is compatible with the Video Publisher's codec.

```
void VideoPublisherDiscoveryListener::on_data_available(
        DDS::DataReader *reader)
```

```
{
    . . .
     // Access the discovery information about remote DataReaders

    retcode = builtinReader->take(...);

}
```

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. It allows you to install a listener for discovery data. For simplicity, this example is constrained to listen to Participant discovery, DataWriter discovery, or DataReader 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 * DDSCommunicator::CreateParticipant (
        long domain,
        const std::string &participantQosLibrary,
        const std::string &participantQosProfile,
        DDS::DataReaderListener *discoveryListener,
        DiscoveryListenerKind listenerKind)

{
        _participant =
                TheParticipantFactory->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.

This example allows you to install a listener for discovery. The code for adding a listener to discovery is located in two methods: `PrepareFactoryForDiscoveryListener()` and `InstallDiscoveryListener()`. The first method changes the default behavior of enabling the DomainParticipant at the time of creation. This allows us to install a listener for discovery data on the disabled DomainParticipant. By doing this, the application will not miss any discovery messages.

The second method checks the type of listener that is being installed. There are three types of discovery that an application can listen for:

- DomainParticipant: learning about other DomainParticipants
- Publication: learning about DataWriters belonging to other DomainParticipants
- Subscription: learning about DataReaders belonging to other DomainParticipants

This example only allows the application to install a single listener to one type of discovery for the sake of simplicity, but it is possible to install listeners for each type of discovery, or a single listener that listens to all three types of discovery.

Adding a listener for discovery:

```
// Lookup the builtin DataReader to listen for either Participant
// discovery, DataWriter discovery, or DataReader discovery.
DDS::DataReader *builtinReader =
        (DDSSubscriptionBuiltinTopicDataDataReader *)
                _participant->get_builtin_subscriber()->
                                lookup_datareader(discoveryTopic);

...

// Listen for discovery events
builtinReader->set_listener(discoveryListener,
                DDS_DATA_AVAILABLE_STATUS);

// Enable the DomainParticipant
_participant->enable();

}
```

Applications

Video Publisher (C++):

This application sends data over the network. The code to create the application's DDS interface is in the class VideoPublisherInterface. This class is composed of two objects:

- DDSCommunicator
- VideoStreamDataWriter

The DDSCommunicator object creates the necessary DDS Entities that are used to create the VideoStreamDataWriter.

The VideoStreamDataWriter class is a DDS DataWriter that uses the Topic `VIDEO_TOPIC`, and the VideoStream data type from the VideoData.idl file.

Listening to Discovery

The Video Publisher application listens to the RTI Connext DDS discovery data. It installs a discovery listener that tells it:

- If a DataReader has been discovered with the same Topic.
- If the DataReader has configured metadata about the codecs it supports as a part of its USER_DATA QoS.

The discovery listener is created in VideoPublisherInterface.h/.cxx. It uses normal DDS mechanisms to take the data out of the discovery queue by calling `take()`. Then, it checks to see if the discovered DataReader specified any metadata in the USER_DATA QoS when it was created. If it did, this calls back a class that will check if that USER_DATA is compatible with the video format this application is providing.

VideoPublisher.h:

```
class VideoPublisherDiscoveryListener : public DDS::DataReaderListener
{
        ...
        virtual void on_data_available(DDS::DataReader *reader);

};
```

VideoPublisher.cxx:

```
void VideoPublisherDiscoveryListener::on_data_available(
        DDSDataReader *reader)
{
        // Cast the DataReader passed into this callback to a
        // "SubscriptionBuiltinTopicDataDataReader" that gets notifications
        // about remote DataReaders.
        DDS::SubscriptionBuiltinTopicDataDataReader *builtin_reader =
                (DDS::SubscriptionBuiltinTopicDataDataReader*) reader;
        DDS_SubscriptionBuiltinTopicDataSeq data_seq;
        DDS_SampleInfoSeq info_seq;
        DDS_ReturnCode_t retcode;

        // Used to retrieve the metadata that was stored in the user_data
        // Qos to send as part of discovery.
        char *readerData;

        // We only process newly seen participants
        retcode = builtin_reader->take(data_seq, info_seq,
                DDS_LENGTH_UNLIMITED,
                DDS_ANY_SAMPLE_STATE, DDS_NEW_VIEW_STATE,
                DDS_ANY_INSTANCE_STATE);
```

```
        ...
                // see if there is user_data associated with the DataReader
                if (data_seq[i].user_data.value.length() != 0)
                {
                        // This sequence is guaranteed to be contiguous
                        readerData =
                                (char*)&data_seq[i].user_data.value[0];

                        // Call back the handler (which will call the video
                        // infrastructure) to find out if the remote codec
                        // is compatible with the codecs this application
                        // is publishing.
                        isCompatible =
                                _handler->CodecsCompatible(readerData);
                        ...
                }
        ...
}
```

The handler that is checking if the codecs are compatible is in the VideoPublisher.cxx file:

```
class CodecCompatibilityCheck : public CodecCompatibleHandler
{
public:
...
        // Calls the gstreamer framework to see if the subscribing app's
        // codec is compatible with what we are sending.  If not, we return
        // false.
        virtual bool CodecsCompatible(std::string codecString)
        {
                if (_source->IsMetadataCompatible(codecString))
                {
                        _discoveredCompatibleReader = true;
                }
                return _discoveredCompatibleReader;
        }
}
```

Sending Data

The VideoPublisher receives video data from the GStreamer framework. In the VideoPublisher.cxx file, it creates a listener that is notified when the GStreamer framework provides a video frame. When it receives a frame, it writes it to the network.

```
class FrameHandler : public EMDSFrameHandler
{
        virtual void FrameReady(void *obj, EMDSBuffer *buffer)
        {
                ...

                // Sending the data over the network (or shared memory)
                pubInterface->Write(*streamData);
```

```
        }
}
```

QoS Considerations

The Video Publisher DataWriter is configured with reliability enabled:

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

However, the Video Subscriber is not configured with reliability enabled:

```
<reliability>
        <kind>BEST_EFFORT_RELIABILITY_QOS</kind>
</reliability>
```

In this configuration, the data is sent in BEST_EFFORT mode, without reliability. This is a valid configuration, and is part of the Request-Offered QoS paradigm of DDS, which allows a DataWriter to offer a higher level of service than a DataReader needs. Since the DataReader is requesting BEST_EFFORT, the DataWriter sends the data BEST_EFFORT.

This is configured this way to make it easy to change the configuration file in a single place to enable reliability.

Video Framework

The application receives notifications from the GStreamer framework to say that video data is ready to be processed and sent over the network. The VideoSource.cxx file contains a wrapper around the GStreamer framework, including specifying the type of codecs the application can handle. This code is not directly related to RTI Connext DDS, so it is not covered in depth.

Video Subscriber (C++)

Receiving Video Data

This application receives video data over the network. The code to create the application's DDS interface is in the class VideoSubscriberInterface. This class is composed of two objects:

- DDSCommunicator
- VideoStreamReader

The VideoStreamReader class is a wrapper around the RTI Connext DDS VideoStreamDataReader class. It provides an API for registering one or more listeners for streaming video data. The VideoStreamReader class uses a DDS listener to receive updates about streaming video data:

```
In VideoSubscriberInterface.h:
class VideoStreamListener : public DDS::DataReaderListener
{
```

```
public:
 ...

        // --- Callback for data ---
        // Notified FROM THE MIDDLEWARE THREAD that data is available.
        // notifies VideoStreamReader of data, which calls its handlers
        // HANDLERS REGISTERED WITH THE VideoStreamReader SHOULD NOT BLOCK
        virtual void on_data_available(DDS::DataReader *reader);
};

In VideoSubscriberInterface.cxx:
void VideoStreamListener::on_data_available(DataReader *reader)
{
        VideoStreamDataReader *videoReader =
                VideoStreamDataReader::narrow(reader);
        VideoStreamSeq dataSeq;
        SampleInfoSeq infoSeq;
        DDS_ReturnCode_t retCode = DDS_RETCODE_OK;

        if (videoReader == NULL)
        {
                return;
        }
        while (retCode != DDS_RETCODE_NO_DATA)
        {

                retCode = videoReader->take(dataSeq, infoSeq);

                ...
                for (int i = 0; i < dataSeq.length(); i++)
                {
                        if (infoSeq[i].valid_data == DDS_BOOLEAN_TRUE)
                        {

                                ...
                                double timestamp =
                                        infoSeq[i].source_timestamp.sec +
                                (infoSeq[i].source_timestamp.nanosec
                                        / NANOSEC_TO_SEC);
                                _reader->NotifyHandlers(&dataSeq[i],
                                                timestamp);
                        }
                }

                // Returning the loaned video data to the middleware.  Note
                // that the data was loaned to the application because the
                // application passed an empty sequence to the take() call.
```

```
                // This is more efficient than copying the data into a
                // local buffer.
                videoReader->return_loan(dataSeq, infoSeq);
        }
};
```

The VideoStreamReader translates the DDS video data into a format the framework recognizes in the `NotifyHandlers` method:

```
void VideoStreamReader::NotifyHandlers(VideoStream *frame,
                                double timestamp)
{
        EMDSBuffer *buffer = NULL;

        for (std::vector<VideoEventHandler *>::iterator it =
                                                _eventHandlers.begin();
                it != _eventHandlers.end(); it++)
        {
                DDS_Octet *frameBinaryData =
                        frame->frame.get_contiguous_buffer();

                // Allocate a new buffer
                EMDSBuffer *buffer = new EMDSBuffer(frame->frame.length());

                // Copy the data and metadata into the new buffer
                buffer->SetData(frameBinaryData, frame->frame.length());
                buffer->SetSeqn(frame->sequence_number);
                buffer->SetTimestamp(timestamp);

                // If this is not the video end, notify the handler that a
                // new frame update should be processed
                (*it)->OnFrameUpdate(buffer);
        }
...

};
```

Notifying the Video Framework of Data Arrival

The Video Subscriber creates a class called `VideoDisplayHandler` that is responsible for notifying the framework that data has arrived. This uses an `EMDSVideoOutput` object to notify the framework of video frames.

```
In VideoSubscriber.cxx
class VideoDisplayHandler : public VideoEventHandler
```

```
{
public:
        ...
        virtual void OnFrameUpdate(EMDSBuffer *buffer)
        {
                _outputHandler->
                        GetFrameHandler()->FrameReady(_outputHandler,
                                                      buffer);
        }
...
};
```

Video Framework

The application sends notifications to the GStreamer framework to say that video data has been received from the network and is ready to be processed. The VideoOutput.cxx file contains a wrapper around the GStreamer framework, including specifying the type of codecs the application can handle. This code is not directly related to RTI Connext DDS, so it is not covered in depth.

Next Steps

Extra Credit

Make the Video Subscriber Codec Incompatible

You can easily change the Video Subscriber application to request a codec that is incompatible with the Video Publisher application. The easiest way to make this change is by changing the properties that the Video Subscriber is expecting. You do this by editing VideoOutput.cxx:

```
_displayPipeline =
    (GstPipeline *)gst_parse_launch(
    "appsrc name=\"src\" is-live=\"true\" do-timestamp=\"true\" "
    "caps=\"video/x-vp8, width=(int)640, height=(int)360, "
    "pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)1000/1\" ! "
    "queue2 ! vp8dec ! queue2 ! "
    "videorate ! video/x-raw-yuv,framerate=25/1 ! "
    "ffmpegcolorspace ! "
    "directdrawsink name=\"sink\"",
     NULL);
```

Change video/x-vp8 to video/MP2T. When you make this change and rebuild, the Video Publisher application will recognize that the Video Subscriber is expecting an incompatible codec and will not send to it. You will see the following output:

```
Waiting for a compatible video subscriber to come online
Waiting for a compatible video subscriber to come online
Discovered a DataReader with an incompatible codec. Ignoring it (not sending it
any data)
Waiting for a compatible video subscriber to come online
```

The Video Publisher application calls the DDS `ignore_subscription()` API to ignore the Video Subscriber's DataReader.

```
void VideoPublisherDiscoveryListener::on_data_available(
                                            DDSDataReader *reader)
{
    . . .
     retcode = participant->ignore_subscription(
                               info_seq[i].instance_handle);
}
```

In your real application, you can take this farther by providing video with different resolutions or quality, and sending that information as a part of the discovery data. Based on this discovery information, you can publish different data.

Make the Video Reliable

If you want to make the video data reliable, you can edit the video_stream_multicast.xml or video_stream_no_multicast.xml file (depending on whether you are running with no multicast available.) Edit the file to change BEST_EFFORT_RELIABILITY_QOS to RELIABLE_RELIABILITY_QOS.

```
<datareader_qos>
    <!-- Streaming video data can be reliable or best-effort depending
         on network characteristics -->
    <reliability>
        <kind>BEST_EFFORT_RELIABILITY_QOS</kind>
    </reliability>
</datareader_qos>
```

The DataWriter is already configured to send data reliably if the DataReader is reliable. In the example, we adjusted the reliability protocol settings to repair dropped samples in a timely manner. Please refer to the XML in the example for detailed settings.
The DataWriter is configured with "non-strict reliability." This means that a cache of data is kept on the DataWriter. The data in the DataWriter's cache is delivered reliably. However, if there is a network disruption, the DataWriter is allowed to overwrite data that the DataReader has not yet received. This offers some reliability for a network that may be busy, but allows the DataWriter to continue sending data even if a DataReader has not received it. This is good for a system like a UAV, where it may be more important to send the current video rather than ensuring that the entire video is received.

```
<!-- Video data can be best-effort or reliable, depending -->
<!--  on network characteristics                          -->
<reliability>
    <kind>RELIABLE_RELIABILITY_QOS</kind>
</reliability>

<!-- Reliably deliver 50 video frames, but do  not block -->
<!-- if the reader does not receive data                 -->
<history>

   <!-- If you need strict reliability, this should be  -->
   <!-- changed to keep all history.                    -->
   <kind>KEEP_LAST_HISTORY_QOS</kind>
   <depth>50</depth>
```

```
</history>
```

You can make the data strictly reliable by changing the QoS from KEEP_LAST_HISTORY_QOS to KEEP_ALL_HISTORY_QOS.

View a More Complete Demo

This source code was written in collaboration with the University of Granada. To read their papers and get more information, you can visit their website at: tl.ugr.es/emds.
The University of Granada demo goes beyond the code in this example by providing the following features:

- Supporting both video and audio data streams
- Encoding metadata about the video or audio data stream using JSON (JavaScript Object Notation)
- Benchmarking the data send rates

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.

Explore More with 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. If you haven't already, download the free trial.

The video shipped with this example is Big Buck Bunny, (c) copyright 2008, Blender Foundation / www.bigbuckbunny.org Video format is WebM, a royalty-free open source video codec.