NVIDIA Open Data Distribution Service (Linux)

DA-09299-001 | February 28, 2019
NVIDIA CONFIDENTIAL | Prepared and provided under NDA

User Guide

TABLE OF CONTENTS

Using Open Data Distribution Service

Data Distribution Service (DDS) is networking middleware for data exchanges using the publish-subscribe pattern for real time distributed applications. DDS ensures interoperability (across different vendors), portability of applications, and high performance.

DDS enables publisher and subscriber nodes to:

- Send and receive messages
- Send and receive events and commands based on topics

Additionally, DDS handles:

- Addressing
- Marshaling and unmarshaling data
- Internal flow control
- Discovery of services

Applications can specify the Quality of Service (QoS) for discovery and runtime behavior.

OpenDDS is an open source, C++ implementation of the OMG Data Distribution Service specification. OpenDDS is built on the ACE abstraction layer. DDS for DRIVE OS includes:

- OpenDDS and the dependent libraries and sample applications.
- Sample applications leverage DDS-based communication methods.

Applications that use DDS for communication services must:

- Include minimal design for the publish-subscribe model of DDS.
- Define the exchange data types using IDL.
- Identify the QoS needs.
- Invoke DDS portable APIs that are independent of DDS implementations.

For the supported OpenDDS versions, see the *Release Notes*.

Within OpenDDS, the following terminology is used and defined as follows:

CellHeading	**CellHeading**
Domain	Represents a global data space. Each domain is uniquely identified by an integer domain ID. Domains are independent from each other. For two DDS applications to communicate with each other, they must join the same domain.
Domain Participant	A domain participant is the entrypoint for an application to

	interact within a particular domain. The domain participant is a factory for many of the objects involved in writing or reading data.
Topic	A topic is the most basic description of data to be published or subscribed to. Each topic describes a data stream in your system. A topic is identified by its name, which is a string that must be unique in the whole domain.
Publisher	The publisher is responsible for taking the published data and disseminating it to all relevant subscribers in the domain.
Subscriber	The subscriber receives the data from the published data and disseminates it to all relevant subscribers in the domain.
Data Writer	Data writer objects are responsible for sending type-specific data to one or more data readers. A data writer is created with a topic, which gives a name to the data stream and associates the data writer with a data type.
Data Reader	Data reader objects are responsible for receiving type-specific data sent by a data writer. Data writers and readers are associated with a topic.
QoS Policies	The entities of a domain have their own set of Quality of Service policies that determine the behavior of the transfer of data and the compatibility between data writers and readers. The entities include: # Domain participant # Topic # Publisher # Subscriber # Data writer # Data reader
Sample	A sample is a single data update received over DDS.
Interface Definition Language (IDL)	The Interface definition language is used to specify the interface between the client and the server so that the Remote Procedure Call (RPC) mechanism can create the code stubs required to call functions across the network.

Installing OpenDDS

The OpenDDS source and binary files are included as part of the DRIVE OS release package.

Use either the SDK Manager or manually extract in sequence the SDK RUN files.

For the branch and build number, see the *Release Notes*.

Once the package is installed, the directory structure is as follows:

```
<top>
├── drive-oss-src
```

```
│ └── dds
│ ├── install
│ ├── install_static
│ ├── opendds
│ ├── samples
├── drive-t186ref-linux
├── bin-target
├── include
├── lib-target
├── targetfs
│ └── samples
```

Once the package is installed, the directory structure is as follows:

```
<top>
├── drive-oss-src
│ └── dds
│ ├── install
│ ├── install_static
│ ├── opendds
│ ├── samples
│ ├── tao
│ ├── xercesc

├── drive-t186ref-linux
├── bin-target
│ └── dds
├── include
│ └── dds
├── lib-target
│ └── dds
├── targetfs
│ └── samples
```

Where:

- `<PDK_TOP>` is the directory where you installed DRIVE OS software package.
- `drive-oss-src/dds` contains the sources for DDS and TAO.
- `drive-oss-src/dds/install_static` contains the static libraries for DDS and TAO.

- `sample code.drive-t186ref-linux/targetfs/home/nvidia/drive-t186ref-linux/samples/dds` contains the pre-built samples using dynamic libraries.
- `drive-t186ref-linux/targetfs/usr/lib` contains the libraries on the target system. These libraries are flashed at `/usr/lib`.

Using Sample Applications

The complex IDL structures application transmits ten messages from the publisher to the subscriber. The data object transferred is a sample object detection metadata used in computer vision/imaging.

The pre-compiled binaries are available at:

```
<PDK-TOP>/drive-t186ref-linux/home/nvidia/drive-t186ref-linux/samples/dds
```

Standalone binaries are not packaged by default. To compile and package standalone binaries, see **Building Standalone Binaries**.

Running the complex_idl_example Test App

The steps below apply for cross process mode. Create two different sessions for starting the publisher and subscriber so that you can see the subscriber receiving and printing data.

The default configuration file uses RTPS and TCP. For more information, see **Discovery Model** and **Transport Model**.

To run the complex_idl_example application

1. Set the environment for both sessions:

```
export DDS_ROOT=/home/nvidia/drive-t186ref-linux/samples/dds
export PATH=$DDS_ROOT:$PATH
```

2. Start the publisher and subscriber in different sessions:

Publisher session:

```
compidl_publisher -DCPSConfigFile generic_config.ini
```

Subscriber session:

```
compidl_subscriber -DCPSConfigFile generic_config.ini
```

The expected output contains:

- Four lines with vertices of rectangles
- Two transformation matrices are printed 10 times with different values in the subscriber session

Sample output is as follows:

```
Object Detection:
EnableBoundingBoxClipping = 1
```

```
EnableFuseObjects = 1
MaxNumImages = 10
ROIs:
h: = 200 1100
w: = 300 2100
x: = 400 3100
y: = 500 4100
Transformations:
| 225 325 425 |
| 525 625 725 |
| 825 925 1025 |
| 250 350 450 |
| 550 650 750 |
| 850 950 1050 |
```

Complex IDL with Security

Security enabled tests undergo these processes:

- Authentication
- Access control
- Encryption

To run an application with security plugins enabled

1. Update the `rtps_multicast.ini` configuration file to enable security by adding the following line in the common section:

   ```
   DCPSSecurity=1
   ```

2. Set the environment variables for both sessions:

   ```
   export DDS_ROOT=/home/nvidia/drive-t186ref-linux//samples/dds
   export PATH=$DDS_ROOT:$PATH


   Copy the libxerces-c.so and libxerces-c-3.2.so from <top>/drive-oss-src/dds/
   xercesc/install/lib/ to the /usr/lib on target.
   ```

3. Create a directory for the publisher and subscriber certificate files on the target.

   These files can be taken from the source PDK and renamed to the provided file name, then copied into the publisher/subscriber folder. For example:

   ```
   cd <top>/drive-oss-src/dds/opendds/tests
   ```

From	To

security/certs/identity/test_participant_*_cert.pem	cert.pem
security/certs/identity/test_participant_*_private_key.pem	private_key.pem
security/certs/identity/identity_ca_cert.pem	identity_ca_cert.pem
security/certs/permissions/permissions_ca_cert.pem	permissions_ca_cert.pem
DCPS/Messenger/governance_signed.p7s	governance_signed.p7s
DCPS/Messenger/permissions_*_signed.p7s	permissions_signed.p7s
Replace * with different index numbers for publisher and subscriber. To exercise authentication, access control, and encryptions, generate your own governance and permissions signed files from the XML as desired.	

4. Generate a `governance_signed.p7s` file based on the use-case by exercising options available for the particular `governance.xml`.

   Some of the options that can be used are:

   ```
   <allow_unauthenticated_participants>false</allow_unauthenticated_participants>

   <enable_join_access_control>true</enable_join_access_control>

   <metadata_protection_kind>ENCRYPT</metadata_protection_kind>

   <data_protection_kind>ENCRYPT</data_protection_kind>
   ```

5. Once you have the updated desired `governance.xml`, run the following command to generate the corresponding signed file and place it in the respective publisher/subscriber directory:

   ```
   penssl smime -sign -in <file.xml> -text -out <file_signed.p7s> -signer
   <ca_cert.pem> -inkey <private_key.pem>
   ```

6. Update the system date using this format:

   ```
   date MMDDhhmm [[CC]YY][.SS]
   ```

   This is required since the licenses may not be valid for the default date set by the system.

7. Launch the application in the same way for cases without security from the newly created publisher/subscriber directories.

   The same output is expected as for non-security cases. However, additional logs are displayed for publisher and subscriber:

   ```
   (3371081|6) DEBUG: Spdp::attempt_authentication() - Attempting authentication
   (sending request) for participant: 9ef3a9f4.7de6be54.a53f712e.000001c1(28b687be)

   (3371081|1) WARNING: Could not find FQDN. Using "127.0.0.1" as fully qualified
   hostname, please correct system configuration.

   (3371081|3) RPCH 9ef3a9f4.7de6be54.a53f712e.000001c1(28b687be) = 12

   (3371081|3) DWCH 9ef3a9f4.7de6be54.a53f712e.ff0202c3(301467ce) = 13

   (3371081|3) DRCH 97bf6f66.343bdf69.08274081.ff0202c4(624ed7a4) = 3

   (3371081|3) RPCH 9ef3a9f4.7de6be54.a53f712e.000001c1(28b687be) = 12

   (3371081|3) DWCH 9ef3a9f4.7de6be54.a53f712e.ff0202c4(ae70f26d) = 2

   (3371081|3) DRCH 97bf6f66.343bdf69.08274081.ff0202c3(fc2a4207) = 14
   ```

Limitations while using security

- Only RTPS discovery is supported.
- Only rtps_udp transport is supported.
- Origin authentication is not supported.
- For the public key of CA, only 2048 bit RSA key and 256 Elliptic Curve Algo are supported.
- `governance.xml` only supports modification of the following fields:

```
allow_unauthenticated_participants
enable_join_access_control
enabled discovery_protection
enable_liveliness_protection
metadata_protection_kind
data_protection_kind
```

For further details, refer to:

http://download.ociweb.com/OpenDDS/Using_DDS_Security_in_OpenDDS_3_13.pdf

Complex IDL with Static Discovery

Static discovery occurs when predefined endpoints with a predefined IP and port location are specified in the configuration file.

The complex IDL example remains the same, with the same expected output. However, since there are code changes required and a different configuration file is used, different binaries for the publisher and subscriber are needed. The environment setup and compile steps are the same, but the way the applications are launched is as follows:

- Publisher session:

```
static_publisher -DCPSConfigFile static_discovery.ini
```

- Subscriber session:

```
static_subscriber -DCPSConfigFile static_discovery.ini
```

Note: **Static discovery supports rtps_udp as the mode of transport.**

Start the subscriber first, before the publisher.

There is a known issue with the static discovery application. The data transfers successfully, but there is a 60 second timeout and the following message is displayed:

```
ERROR: Subscriber_static.cpp:146: main() - wait failed!
(1663031|1) WARNING: DataLink[101f4100]::~DataLink() - link still in use by 1
entities when deleted!
(1663031|1) ERROR: SubscriberImpl::~SubscriberImpl, 1 datareaders still exist.
```

Use Cases

Use cases have been classified into three types:

- **Intra-SoC or single VM**
- **Inter-VM**
- **Inter-SoC**

OpenDDS supports three types of discoveries and five types of transports, which are changeable in the `generic_config.ini` file.

For more information, see **Discovery Model** and **Transport Model**.

The default configuration file uses RTPS discovery and TCP transport. There are other configuration files in the `<PDK_TOP>/drive-t186ref-linux/targetfs/home/nvidia/drive-t186ref-linux/samples/dds` folder. These can be used instead of the current `generic_config.ini`. Note that the same config file must be used for the publisher and the subscriber.

Single VM/Intra-SoC Use Cases

The following table lists out the configuration files in the package for single VM use cases, and what transport mode and discovery mode they use.

Configuration File	Discovery Mode	Transport Mode
generic_config.ini (default)	RTPS	tcp
static_discovery.ini	Static	rtps_udp
shmem.ini	RTPS	shmem
rtps_multicast.ini	RTPS	rtps_udp

Inter-VM and Inter-SoC Use Cases

DDS can be used to transfer data from one VM to another in a multi-virtual machine environment. The configuration of these use cases is similar to intra-VM use cases, except the two different sessions for publisher and subscriber belong to different VMs or SoCs.

The transport and discovery mode limitation:

- RTPS discovery only works if multicast support is enabled.

The configuration file `rtps_multicast.ini` uses RTPS discovery and `rtps_upd` transport. Modify the following line according to the interface where the Virtual Machine must connect to other Virtual Machines.

```
MulticastInterface=hvX
```

- Where X=0,1,2, depending on the VM's bridge interface to the other VM.

For inter-VM communication, the bridge interface `hv0-hv1` is used between Linux and QNX.

To enable multicast support for RTPS discovery, you must modify the `tegra_t186ref_gnu_linux_defconfig` file (extracted from `oss_src.run`) and change this line:

```
CONFIG_IP_MULTICAST=y
```

Now recompile the kernel. For the steps, see **Building the Flashing Kernel**. If you do not recompile, the DDS application fails to launch with error: "unable to join multicast group".

For an RTPS based discovery mechanism, participants discover each other using Simple Participant Discovery Protocol (SPDP), which is based on multicast-UDP transport. The `rtps_multicast.ini` configuration file has an `InteropMulticastOverride` field to override the default multicast address `239.255.0.1`.

Ensure the kernel IP routing table has an entry against the specified multicast address or is handled using the default gateway. If the multicast address is NOT provided, the DDS middleware reports the following error while sending SDPDP related messages:

```
no route to host
```

The following provides an example for adding an entry in the kernel IP routing table for inter-VM:

```
#route add 239.255.0.0 netmask 255.255.0.0 hv1
```

To verify:

```
#route -n
```

The endpoint discovery can be triggered without multicast, where the following changes can be made in the configuration file:

```
SedpMulticast=0

SedpLocalAddress=<Local ip:port>
```

Use the `complex_idl_example` test application. Follow the steps depending on which VM the publisher and subscriber are started.

For the inter-SoC use case, use the `rtps_multicast.ini` configuration file and follow the steps for inter-VM communication. Change the `MulticastInterface` according to the interfaces that connect.

In all inter-VM/inter-SoC use cases, the transport mode must be specified with a local address. The local address must be an interface that can be pinged from the outside world, or at least from where the subscriber runs.

```
[transport/tcp1]

transport_type=tcp

local_address=IP:Port

## Substitute this with IP and Port for VM/SoC where publisher/subscriber is run
```

Static Discovery in Inter-VM/Inter-SoC

Use the `static_discovery.ini` configuration file for static discovery use cases. Recompiling the kernel is NOT required for static discovery. However, `static_publisher` and `static_subscriber` binaries must be used. Change the IP addresses as per the Virtual Machines or SoCs. Make the changes as follows:

```
[transport/rudp] ## Reader Transport
```

```
transport_type=rtps_udp
use_multicast=0
local_address=IP:Port
## Substitute this with IP and Port for VM/Tegra where subscriber is run

[transport/rtudp] ## Writer Transport
transport_type=rtps_udp
use_multicast=0
local_address=IP:Port
## Substitute this with IP and Port for VM/Tegra where publisher is run
```

For inter-VM communication, the bridge interface hv0-hv1 is used between Linux and QNX.

- Linux fixed IP address is 192.168.2.2
- QNX fixed IP address is 192.168.2.1

For inter-SoC communication, external IPs are used instead.

Building Standalone Binaries

The pre-built binaries present in

```
<TOP>/drive-t186ref-linux/home/nvidia/drive-t186ref-linux/samples/dds
```

are built using dynamic libraries present in

```
<TOP>/drive-t186ref-linux/usr/lib
```

To compile standalone binaries:

8. Change directory to:
   ```
   <TOP>/drive-oss-src/dds/samples/complex_idl_example/build_scripts
   ```
9. Run the `build.sh` script:
   ```
   ./build.sh static
   ```
10. Change directory to:
    ```
    <TOP>/drive-oss-src/dds/samples/
    ```
11. Run the `copy_samples.sh` script:
    ```
    ./copy_samples.sh
    ```

Data Types/IDL

DDS applications send and receive messages that are strongly-typed. These types are defined by the application developer in Interface Definition Language (IDL). For example:

```
struct Data {
```

```
long message_counter;
string stock_name;
double price;
};
```

Since OpenDDS is capable of exchanging messages between processes created in different languages, a structure must be defined that can be translated to all supported languages. For best practices, structures are defined in separate files with the extension `.idl`. These IDL files are written in C++ style, but do not have the same syntax. Since the file is created once, mismatch errors are avoided between publisher and subscriber defined data types. The IDL file is then compiled with OpenDDS tools to create source files for both publishers and subscribers.

The OpenDDS IDL compiler is invoked using the `opendds_idl` executable, located in `$DDS_ROOT`. The IDL compiler:

- Parses a single IDL file.
- Generates the serialization and key support code that OpenDDS requires to marshal and de-marshal the types.
- Generates the type support code for the data readers and writers.

For each IDL file processed, such as `xyz.idl`, three files are generated:

- xyzTypeSupport.idl
- xyzTypeSupportImpl.h
- xyzTypeSupportImpl.cpp

Typically, `opendds_idl` is passed several options and the IDL file name as a parameter.

```
$DDS_ROOT/opendds_idl xyz.idl
```

For more details on options of `opendds_idl`, see the OpenDDS Developer Guide as provided in **References.**

The `xyz.idl` and `xyzTypeSupport.idl` are compiled by the TAO IDL compiler using the `tao_idl` executable. These client stubs are generated:

```
xyzC.h
xyzC.inl
xyzC.cpp
xyzTypeSupportC.h
xyzTypeSupportC.inl
xyzTypeSupportC.cpp
```

These server skeletons are generated:

```
xyzS.h
xyzS.cpp
xyzTypeSupportS.h
xyzTypeSupportS.cpp
```

- Pure client applications require `#include` and link with client stubs.

- Pure server applications require `#include` and link with server skeletons.

  ```
  tao_idl xyz.idl
  ```

  ```
  tao_idl -I<top>/drive-oss-src/dds/opendds xyzTypeSupport.idl
  ```

For more details, see the TAO IDL User's Guide as provided in **References**.

Discovery Model

OpenDDS provides three options for discovery:

- **Information Repository**: A centralized repository style that runs as a separate process allowing publishers and subscribers to discover one another centrally.
- **RTPS Discovery**: A peer-to-peer style of discovery that uses the RTPS protocol to advertise availability and location information.
- **Static Discovery**: A way of discovering which topic and endpoints, as well as QOS policies of all entities, are defined in the configuration file.

Transport Model

Transport Model

Transport mechanisms in OpenDDS are pluggable because they can be replaced by another mechanism through changes in a configuration `.ini` file. The transport framework is extensible to implement customized transports.

OpenDDS transport implementations include:

- tcp
- udp
- multicast
- shmem
- rtps_udp

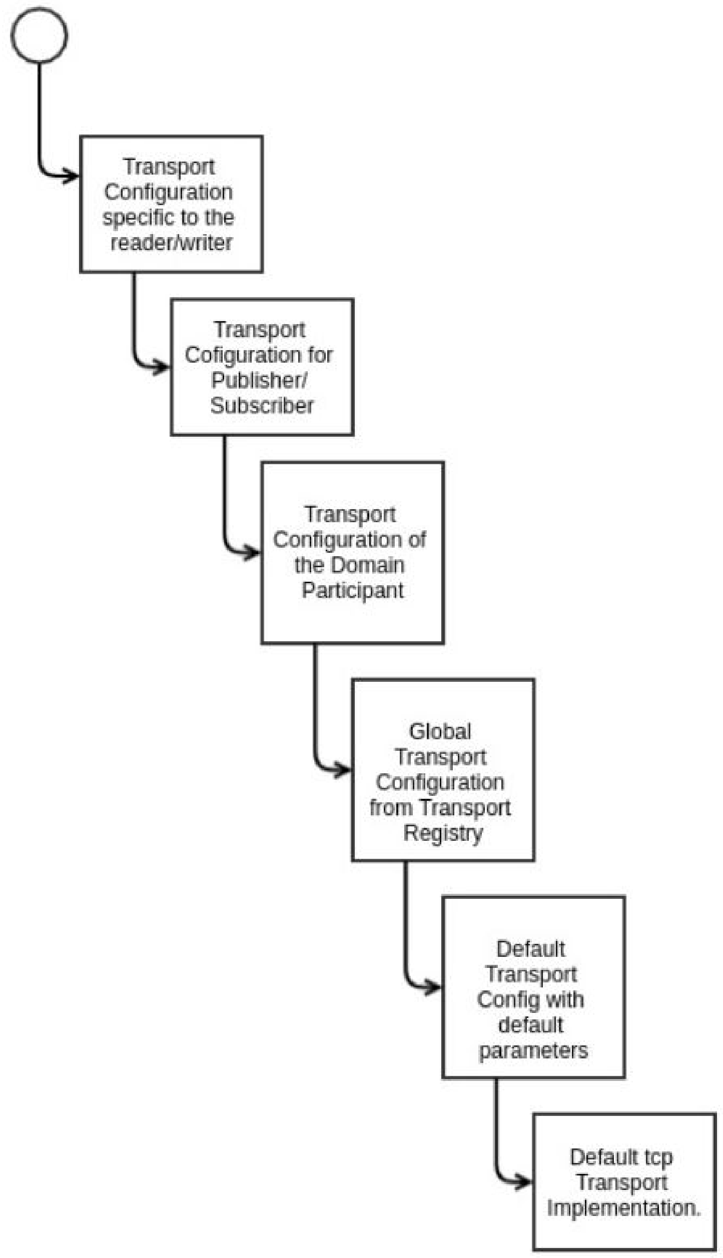
Transport configurations can be specified for:

- Domain participants
- Publishers
- Subscribers
- Data writers
- Data readers

Each transport configuration consists of transport instances, which are pre-configured transport implementations for a channel. The transport configuration and instances are managed in the transport registry, which is created using programming APIs or configuration files.

Transport Selection Hierarchy

Each data writer or reader follows a specific hierarchy in selecting a transport.

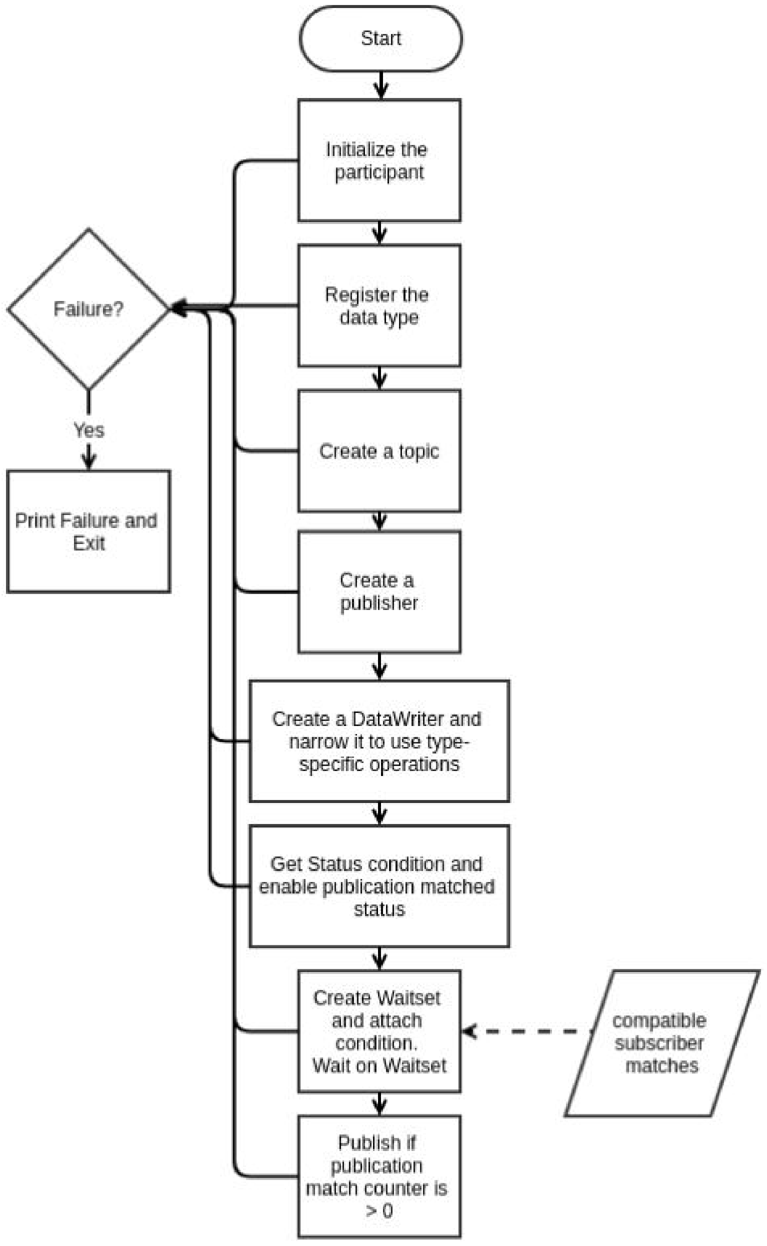

Transport Configuration specific to the reader/writer
Transport Cofiguration for Publisher/ Subscriber
Transport Configuration of the Domain Participant
Global Transport Configuration from Transport Registry
Default Transport Config with default parameters
Default tcp Transport Implementation.

Programming Guidelines

Follow these guidelines for writing and compiling an IDL in Data Types/IDL.

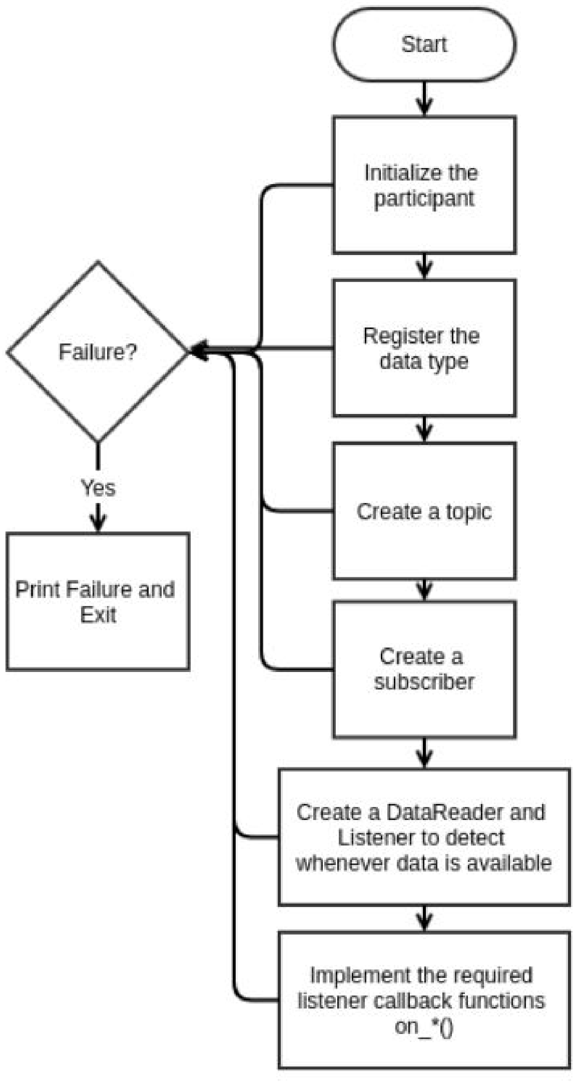
Writing a Publisher

For best practices, follow these guidelines on writing a publisher. Using these guidelines avoids the use of conditions and wait sets for a free-running publisher.


Start
Initialize the participant
Register the data type
Create a topic
Create a publisher
Create a DataWriter and narrow it to use type-specific operations
Get Status condition and enable publication matched status
Create Waitset and attach condition. Wait on Waitset
Publish if publication match counter is > 0
compatible subscriber matches
Failure?
Yes
Print Failure and Exit

For more information about writing a publisher, see the OpenDDS Developer's Guide as provided in **References**.

Writing a Subscriber


Start
Initialize the participant
Register the data type
Create a topic
Create a subscriber
Create a DataReader and Listener to detect whenever data is available
Implement the required listener callback functions on_*()
Failure?
Yes
Print Failure and Exit

For more information about writing a subscriber, see the OpenDDS Developer's Guide as provided in **References**.

Quality of Service for DDS Entities

There are several Quality of Service policies available as a part of OpenDDS. Each specify a defined structure. Different sets of policies are applicable to:

- Domain participant
- Topic
- Publisher
- Subscriber

- Data writer
- Data reader

If the QoS is changed, existing associations are removed if they are no longer compatible and new associations are added if they become compatible.

OpenDDS QoS Policies

The OpenDDS QoS policies are as follows:

CellHeading	CellHeading
LIVELINESS	Determines whether participants are alive and reachable. Implemented using heartbeat or direction assertion.
RELIABILITY	Best effort vs. reliable. # Reliable specifies the service attempts to deliver all samples in its history. # Best effort indicates it is acceptable to not retry propagation of any samples.
HISTORY	Determines the number of samples to retain until the consumer acquires them.
DURABILITY	Determines if the data writers must maintain samples after they have been sent to subscribers.
DURABILITY SERVICE	Controls the deletion of samples in the TRANSIENT or PERSISTENT durability cache and provides a way to specify the history and resource limit for the sample cache.
RESOURCE LIMITS	Determines the amount of resources the service can consume to meet the requested QoS.
PARTITION	Creates a logical partition within a domain using a string name.
DEADLINE	Allows the application to detect when data is not written or read within a specified amount of time.
LIFESPAN	Allows the application to specify when a sample expires. Expired samples are NOT delivered to subscribers.
USER DATA	Can be set to any sequence that can be used to attach information to the created entity, such as security credentials for authentication.
TOPIC DATA	Can be set to attach additional information to the created topic.
GROUP DATA	Can be used to implement matching mechanisms similar to those of the PARTITION policy, except the decision is based on an application-defined policy.
TRANSPORT PRIORITY	Considered a hint to the transport layer to indicate at what priority to send messages. Higher values indicate a higher priority.
LATENCY BUDGET	Considered a hint to the transport layer to indicate the urgency of samples being sent and is used only for monitoring purposes at this time.

ENTITY FACTORY	Controls whether entities are automatically enabled when they are created.
PRESENTATION	Controls how changes to instances by publishers are presented to data readers. It affects the relative ordering of these changes and the scope of this ordering.
DESTINATION ORDER	Controls the order in which samples within a given instance are made available to a data reader.
WRITER DATA LIFECYCLE	Controls the lifecycle of data instances managed by a data writer.
READER DATA LIFECYCLE	Controls the lifecycle of data instances managed by a data reader.
TIME BASED FILTER	Controls how often a data reader may be interested in changes to values in a data instance.
OWNERSHIP	Determines whether more than one data writer can write samples for the same data-object instance.
OWNERSHIP STRENGTH	Used in conjunction with the OWNERSHIP policy, when the OWNERSHIP kind is set to EXCLUSIVE. The data writer with the highest value of strength is considered the owner of the data-object instance.

Recommended Policies for Use-cases

Recommended Policies for Use-cases

- Streaming Data
  - Deadline
  - Reliability
  - Time based filter
  - Durability
  - History
  - Transport priority
- Alarms/Events
  - Reliability
  - Durability
  - Liveliness
  - History
- Large Data
  - Reliability
  - Liveliness
  - Transport priority

Guidelines on integration with other Build Systems

OpenDDS does not support IDL to C mapping, but C++ mappings are supported. The OpenDDS header files expose RTTI functionality directly. If an application built by the NVIDIA build system includes these header files, it can result in build errors.

To overcome build error problems

- Use a C wrapper or a C++ shim layer over the publisher/subscriber application.
- Use OpenDDS build flags while compiling the application.

The C / C++ wrapper can then be built using the regular `tmake` build.

Migrating to Other DDS Implementation

The following sections describe specification compatibility and provide interoperability notes.

Specification Compatibility

OpenDDS states compliance with these specifications:

- Version 1.4 of OMG Data Distribution Service (DDS) for Real-Time Systems specification (formal/2015-04-10)
- Version 2.2 of the Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDSI-RTPS) (formal/2014-09-01).

OpenDDS does NOT implement these specifications:

- DDS Security 1.1b (ptc/17-09-20)
- Extensible and Dynamic Topic Types for DDS Specification Version 1.2 (formal/17-08-01)
- Remote Procedure Calls over DDS Version 1.0 (formal/17-04-01)

Other DDS implementations may include these specifications, and cater to wider use cases than OpenDDS.

Interoperability between OpenDDS and commercially available DDS implementations is not affected, provided that RTPS interoperability protocol is used for discovery.

Design Effort for Migration to Different DDS Implementation

- **Type Representation**: OpenDDS supports OMG IDL spec version 3.1 from the CORBA specification. The latest IDL specification version is 4.2. Other DDS implementations may support a higher IDL version. In addition, other DDS implementations support other languages to represent types as specified in the DDS-XTYPES spec mentioned above, such as XML, XDR, etc., providing flexibility to represent DDS topic type support in a more readable form.

  If migration of type representation format is desired, the effort is proportional to the complexity and number of IDL files migrated.

- **Transport and Discovery Mechanisms**: Commercial distributions support a superset of transport mechanisms that include mandated transports and discovery methods of DDS. Consequently, applications using the recommended transport and discovery methods must work as-is. If a custom transport mechanism is developed, re-implementation may be required. The `DCPSInfoRepodiscovery` mechanism is specific to OpenDDS and must not be used if eventual migration or interoperability is desired.

- **Choosing QoS Policies**: Since QoS policies are specification driven, no change is expected to support these in other DDS implementations. Additional QoS policies may exist. For example, `TypeConsistencyEnforcementQosPolicy`, `DataRepresentationQoSPolicy`, etc.

Coding Effort

- **IDL Compiler**: OpenDDS supports C++ and Java bindings for data objects, while commercial versions may support more languages. Moreover, commercial versions may also have code generators that write boilerplate code for publisher and subscriber and its build environment, over and above compiling the IDL file. The generated publisher and subscriber code might look very different from what NVIDIA writes for OpenDDS, because the class definitions differ. Since the DDS specification platform independent model defines the APIs including the DDS entities, QoS policies, listeners, etc., those remain the same.
- **Publisher/Subscriber Code**: Commercial versions may have a boilerplate publisher, subscriber already written, and extension to that code for your application, if necessary. DDS users may, however, choose not to use the boilerplate code and port your application manually, wherein code refactoring might be required.

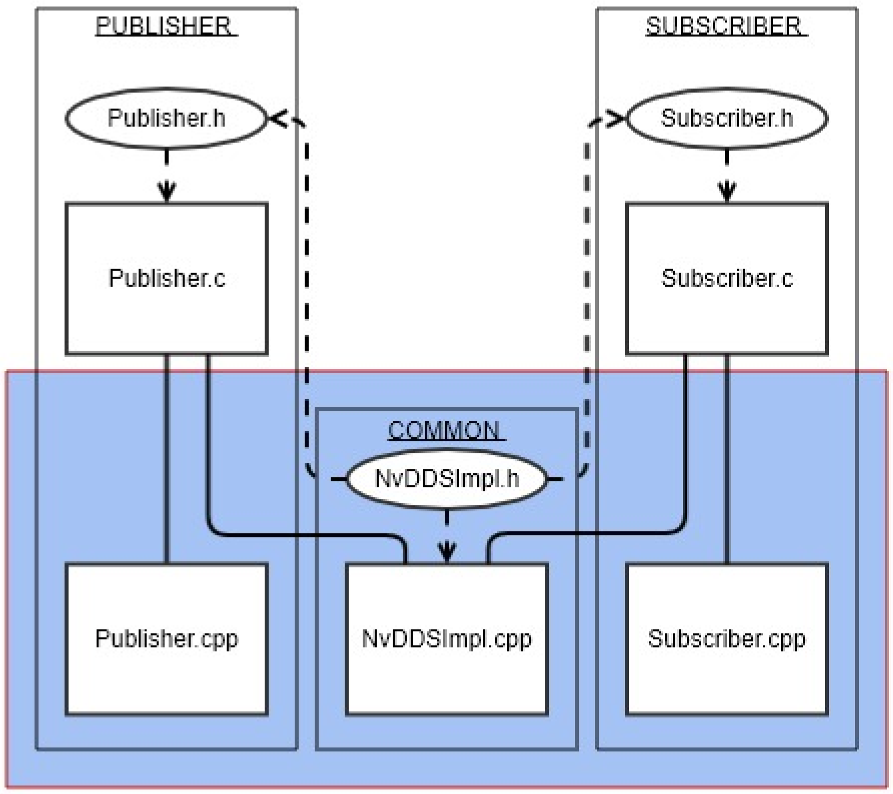
  For example, transport/discovery parameters for OpenDDS can be given via the config file, which may look very different than other distributions—or might not exist—and parameters are passed via the code itself.

Interoperability Issues

Due to a different interpretation of DDS specification by commercial DDS implementations, there may be some interoperability issues. For example, interpretation of CDR. Interoperability demonstrations have been successfully organized between commercial DDS implementations and OpenDDS using a common application, so migration and interoperability is possible with minimal tuning.

Minimal design change and coding adaptation is required to migrate to a commercial version due to a well-defined portable DDS specification.

Shim Layer



The complex_idl example has a shim layer written with the below mentioned directory structure. The application specific definitions and declarations for a given idl have been abstracted into the respective `Publisher.h` and `Subscriber.h` files.

All OpenDDS specific data types and function API calls are extracted into the common folder for NVIDIA's implementation. Based on the requirement, the commercial version can replace the existing open source definitions.

This abstraction allows `Publisher.c` and `Subscriber.c` to be constant over various platforms and implementations. The changes to `Publisher.cpp` and `Subscriber.cpp` are minimal based on the use case.

References

- **DDS Specification Version 1.4**
- **RTPS Specification Version 2.2**
- **DDS Security Specification Version 1.1 Beta 1**
- **OpenDDS Developer's Guide Version 3.12**
- **OpenDDS API Guide**
- **TAO IDL Compiler Users Guide**

Third-Party Licenses

This topic provides license information about the third-party software libraries included in this NVIDIA product.

OpenDDS License

ACE+TAO License

Legal Information