Node.js DCPS API Guide

Release 6.x

Contents

1 Preface

1.1 About the Node.js DCPS API Guide

The Node.js DCPS API Guide is a starting point for anyone using, developing or running Node.js applications with Vortex OpenSplice.

This guide contains:

- Node.js DCSP API setup instructions
- location of Node.js DCPS API dds module documentation
- overview of general DDS concepts and Node.js API for Vortex DDS
- a listing of examples, and how to run them
- detailed information on how to specify DDS entity Quality of Service (QoS)
- how to register DDS topics

This reference guide is based on the OMG's Data Distribution Service Specification.

Please note that this guide is not intended to provide a detailed explanation of the aforementioned OMG specifications or the Vortex OpenSplice product. It provides an introduction to the essential concepts and enables users to begin using the Node.js DCPS API as quickly as possible.

1.2 Intended Audience

The Node.js Reference Guide is intended to be used by JavaScript programmers who are using Vortex OpenSplice to develop Node.js applications.

2 Introduction

The Node.js DCPS API provides users with Node.js classes to model DDS communication using JavaScript and pure DDS applications.

The Node.js DCPS API is a native JavaScript binding that supports DDS functionality. The language binding consists of Node.js classes and wrapper implementations of the C99 API (C API for DDS). It makes use of ECMAScript 2015 (ES6) JavaScript language features and leverages ease of use by providing a higher level of abstraction.

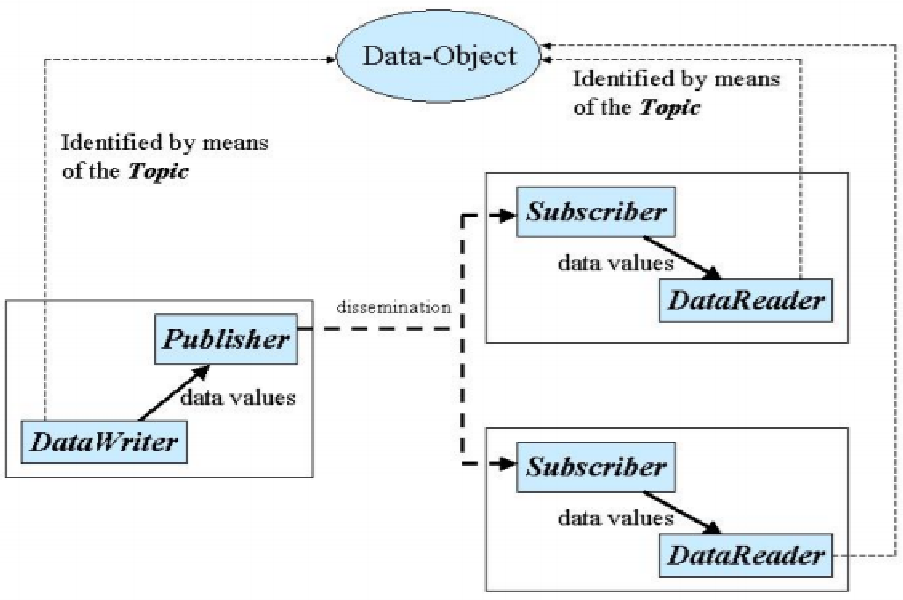
Please see *Limitations of Node.js Support* for limitations of Node.js DDS API support.

2.1 DDS

What is DDS?

"The Data Distribution Service (DDS™) is a middleware protocol and API standard for data-centric connectivity from the Object Management Group® (OMG®). It integrates the components of a system together, providing low-latency data connectivity, extreme reliability, and a scalable architecture that business and mission-critical Internet of Things (IoT) applications need."

"The main goal of DDS is to share the right data at the right place at the right time, even between time-decoupled publishers and consumers. DDS implements global data space by carefully replicating relevant portions of the logically shared dataspace." DDS specification



Further Documentation

http://portals.omg.org/dds/

http://ist.adlinktech.com/

3 Installation

This section describes the procedure to install the Node.js DCPS API on a Linux or Windows platform.

3.1 Dependencies

The Node.js DCPS API has several dependencies that must be installed.

3.1.1 Linux 64-bit

- Node.js LTS 8.11.1 or later (preferably Node.js LTS 8.x version)
- npm (node package manager) version 5.6.0 or later (typically included with a Node.js install)
- Python 2.7 (v3.x.x is not supported)
- make
- C/C++ compiler toolchain like GCC

3.1.2 Windows 64-bit

- Node.js LTS 8.11.1 or later (preferably Node.js LTS 8.x version)
- npm (node package manager) version 5.6.0 or later (typically included with a Node.js install)
- Python 2.7 (v3.x.x is not supported)
- Visual C++ build tools (VS 2015 or VS 2017)

3.1.2.1 Python and Visual C++ build tools install

- Install all the required tools and configurations using Microsoft's windows-build-tools by running the following from a command prompt as an administrator:

```
npm install --global --production windows-build-tools
```

(or)

- Install tools and configuration manually
  - Visual C++ build tools (VS 2015 or VS 2017)
  - Python 2.7 (v3.x.x is not supported)

More detailed information on installing Python and Visual C++ build tools on Windows can be found at: https://github.com/Microsoft/nodejs-guidelines/blob/master/windows-environment.md#compiling-native-addon-modules

3.2 OpenSplice (OSPL) and Node.js DCPS API Installation

Steps:

1. Install OSPL. Choose an HDE type installer which is a Host Development Environment. This contains all of the services, libraries, header files and tools needed to develop applications using OpenSplice. The Node.js DCPS API is included in the following installers:

Platform	Platform Code
Ubuntu1404 64 bit	P704
Ubuntu1604 64 bit	P768
Windows10 64 bit	P738

   Example installer:

   P704-VortexOpenSplice-6.x.x-HDE-x86.linux-gcc4.1.2-glibc2.5-installer.run

2. Setup OSPL license. Copy the license.lic file into the appropriate license directory

   /INSTALLDIR/Vortex_v2/license

3. Node.js DCPS API files are contained in a tools/nodejs folder

   Example: *$OSPL_HOME/tools/nodejs*

3.3 Installing Node.js DCPS API in a Node.js application

1. Start a command shell. Setup OSPL environment variables by running release.com or release.bat which can be found in

   Linux

   /INSTALLDIR/ADLINK/Vortex_v2/Device/VortexOpenSplice/6.x.x/HDE/x86_64.linux/

   Windows

   \INSTALLDIR\ADLINK\Vortex_v2\Device\VortexOpenSplice\6.x.x\HDE\x86_64.windows\

2. Create a node project folder, if not created

   mkdir <project_name>

   cd <project_name>

   npm init

3. Change directory to node project folder

4. Install the Node.js DCPS API to your project by executing:

   Linux

   npm install $OSPL_HOME/tools/nodejs/vortexdds-x.y.z.tgz

   Windows

   npm install %OSPL_HOME%\tools\nodejs\vortexdds-x.y.z.tgz

3.4 Examples and Documentation

1. Examples directory:

   $OSPL_HOME/tools/nodejs/examples

2. Node.js DCPS API documentation directory:

   $OSPL_HOME/docs/nodejs/html

3. Node.js DCPS User Guide (HTML and PDF) directory:

   $OSPL_HOME/docs

4 Examples

Examples are provided to demonstrate the Node.js DCPS features.

The examples can be found in the following directory:

> *$OSPL_HOME/tools/nodejs/examples*

WHERE:

> Linux
>
> > OSPL_HOME = *INSTALLDIR/ADLINK/Vortex_v2/Device/VortexOpenSplice/6.9.x/HDE/x86_64.linux*
>
> Windows
>
> > OSPL_HOME = *INSTALLDIR\ADLINK\Vortex_v2\Device\VortexOpenSplice\6.9.x\HDE\x86_64.win64*

4.1 Example Files

Each subfolder consists of a Node.js example project.

4.1.1 Examples

Example	Description
HelloWorld	Demonstrates how to read and write a simple topic in DDS
jsshapes	Demonstrates how to read and write to a DDS application
IoTData	Demonstrates reading and writing an IoTData topic
PingPong	Demonstrates reading, writing, and waitsets
GetSetQoSExample	Demonstrates getting and setting QoS using QoS Provider and DCPS api
ListenerExample	Demonstrates how to attach listener to a DDS entity

4.1.2 File Types

The examples directory contains files of different types.

File Type	Description
js	A program file or script written in JavaScript
package.json	Lists the packages that a project depends on
xml	An XML file that contains one or more Quality of Service (QoS) profiles for DDS entities
idl	An interface description language file used to define topic(s)

4.2 Running Examples

To run a Node.js example:

1. Setup OSPL environment variables

   Linux

   - For each example javascript file to be run, open a Linux terminal
   - Navigate to directory containing release.com file (OSPL_HOME)

     /INSTALLDIR/ADLINK/Vortex_v2/Device/VortexOpenSplice/6.9.x/HDE/x86_64.linux

   - Run release.com (". release.com")

   Windows

   - For each example javascript file to be run, open a command prompt
   - Navigate to directory containing release.bat file (OSPL_HOME)

     \INSTALLDIR\ADLINK\Vortex_v2\Device\VortexOpenSplice\6.9.x\HDE\x86_64.win64

   - Run release.bat ("release.bat"), if OSPL environment variables are not set system wide

2. Change directory to the example folder

   Example:

   $OSPL_HOME/tools/nodejs/examples/HelloWorld

3. Run npm install to install the Node.js DCPS API (vortexdds-x.y.z.tgz) and all other dependencies

   npm install

Note: If you are running the examples from another directory outside the OSPL install, then you will need to manually install the Node.js DCPS API first and then run npm install for the example dependencies as follows:

npm install $OSPL_HOME/tools/nodejs/vortexdds-x.y.z.tgz

npm install

---

4. Run .js file(s) in a terminal/command shell

   Example:

   node HelloWorldSubscriber.js

5

Node.js API for Vortex DDS

The Node.js DCPS API provides users with Node.js classes to model DDS communication using JavaScript and pure DDS applications.

The Node.js DCPS API consists of one module.

- vortexdds

This section provides an overview of the main DDS concepts and Node.js API examples for these DDS concepts.

Note:

- The Node.js DCPS API documentation can be found in the following directory:

  $OSPL_HOME/docs/nodejs/html

5.1 API Usage Patterns

The typical usage pattern for the Node.js DCPS API for Vortex DDS is the following:

- Model your DDS topics using IDL and generate Node.js topic classes from IDL.
- AND/OR generate Node.js topic classes for topics that already exist in the DDS system.
- Start writing your Node.js program using the Node.js API for Vortex DDS.

The core classes are `Participant`, `Topic`, `Reader` and `Writer`. `Publisher` and `Subscriber` classes can be used to adjust the Quality of Service (QoS) defaults.

For details on setting QoS values with the API, see *Quality of Service (QoS)*.

The following list shows the sequence in which you would use the Vortex classes:

- Create a `Participant` instance.
- Create one or more `Topic` using the Participant instance.
- If you require publisher or subscriber level non-default QoS settings, create `Publisher` and/or `Subscriber` using the Participant instance. (The most common reason for changing publisher/subscriber QoS is to define non-default partitions.)
- Create `Reader` and/or `Writer` classes using the `Topic` instances that you created.
- If you required data filtering, create a `QueryCondition` using the Reader instance.
- Create the core of program, writing and/or reading data and processing it.

5.1.1 Asynchronous Aspects of the API

Some DDS operations can take a long time. In order to not block the execution of other asynchronous JavaScript operation, the Vortex DDS API for NodeJS uses asynchronous operations the return standard JavaScript Promise objects.

In addition, at creation, some entities accept a 'listener' object, which defines one or more 'callback' methods, which are called asynchronously by the NodeJS engine when the appropriate event occurs. Entities supporting listeners are: Topic, Reader and Writer. See their factory methods for documentation on these listeners.

5.1.2 Releasing DDS resources

Many DDS objects have associated with them resources obtained from the DDS system. The NodeJS engine does not reclaim these resources, even if it garbage collects an object the represents such a resource.

In order to avoid leaking of DDS resources, you should take care to call the appropriate delete() method when you are finished with a DDS object. Once the delete() method is called on an object, it is no longer usable.

Many DDS objects are organized into a hierarchy. DDS objects created by factory methods on another DDS object are implicitly 'owned' by that 'parent' object. If a parent object is deleted, then all directly and indirectly owned objects are also deleted. The only DDS types that are not 'owned' by other objects are:

- Participant, which sit at the top of the ownership hierarchy
- Waitset instances.
- GuardCondition instances.
- QoSProvider instances.

To completely clean-up DDS resources, at a minimum, your program must explicitly delete all instances of the above types. Instances of other DDS objects may be explicitly deleted by your program once you no longer require them. Explicitly deleted such objects will reduce the footprint of your DDS application.

5.1.3 Exceptions

Most APIs will throw exceptions, rather than return 'error codes'.

The most common exception thrown is DDSError, which represents an error within the DDS system. Frequently, additional information is written to the file dds-error.log. All methods that throw DDSError have this fact explicitly documented.

Most methods also type-check their arguments. If errors are found in these arguments, then a standard JavaScript TypeError is typically throw. The documentation does not explicitly document when TypeError is thrown.

5.1.4 Key API References

- To import data types defined in an Vortex DDS compliant IDL file, see importIDL().
- To connect to a DDS domain, create a Participant.
- To register a DDS topic, use Participant.createTopic().
- To create a DDS data reader, use Participant.createReader() or Subscriber.createReader().
- To create a DDS data writer, use Participant.createWriter() or Subscriber.createWriter().
- Import a externally defined quality-of-service (QoS) 'profiles', see QoSProvider.
- To programmatically create or examine quality-of-service (QoS) policies on an entity, see QoS and Entity.qos.
- To create a DDS 'waitset' that enables you to wait (asynchronously) for specific conditions, see Waitset.

5.2 Participant

The Node.js `Participant` class represents a DDS domain participant entity.

In DDS - "A domain participant represents the local membership of the application in a domain. A domain is a distributed concept that links all the applications able to communicate with each other. It represents a communication plane: only the publishers and subscribers attached to the same domain may interact."

A DDS domain participant, represents a connection by your program to a DDS Domain.

Typically, your application will create only one participant. All other DDS entities are created through a participant or one of its child entities via factory methods such as `Participant.createPublisher()`.

Note: An explicit `delete()` is required for the `Participant` to release DDS resources. All entities owned directly or indirectly by the `Participant` will be released on `delete()`.

Example: Create new participant

```
const dds = require('vortexdds');

// create domain participant
const participant = new dds.Participant();
```

Example: Create new participant on the default domain with a QoS profile

```
const dds = require('vortexdds');
const path = require('path');

const QOS_XML_PATH = __dirname + path.sep + 'DDS_Get_Set_QoS.xml';
const QOS_PROFILE = 'DDS GetSetQosProfile';
const DOMAIN_ID = dds.DDS_DOMAIN_DEFAULT;

//...

async function setup(){

    let participant = null;
    let qp = null;
    try {
        // create a qos provider using qos xml file
        qp = new dds.QoSProvider(QOS_XML_PATH, QOS_PROFILE);

        // get participant qos from qos provider and create a participant
        const pqos = qp.getParticipantQos();

        participant = new dds.Participant(DOMAIN_ID, pqos);

        //...

    } finally {
        console.log('=== Cleanup resources');
        if (qp !== null){
        qp.delete();
        }
        if (participant !== null){
          participant.delete().catch((error) => {
            console.log('Error cleaning up resources: '
              + error.message);
          });
        }
    }
}
```

5.3 Topic

The Node.js `Topic` class represents a DDS topic type.

A `Topic` represents a globally named data type, along with quality-of-service policies. The `Topic` is available to all participants in a DDS domain, and must be registered with the same data type definition and QoS policies.

You must create a `Topic` before you can create `Reader` or `Writer` instances.

Class instances are created by the `Participant.createTopic()` function.

```
createTopic(
        topicName,
        typeSupport,
        qos = null,
        listener = null
)
```

In order to create a `Topic`, a `TypeSupport` instance must be created from an IDL file.

Step 1 - Generate TypeSupport objects from IDL file

The function `importIDL(idlPath)` is provided to generate a `TypeSupport` instance for every topic defined in an IDL file. This function returns a promise, therefore should be called from an async function.

Step 2 - Create Topic instance using TypeSupport

The `createTopic` method in the `Participant` class can then be used to create a topic instance.

Example: Create a topic

```
const dds = require('vortexdds');
const path = require('path');

//...

async function publishData(){

  console.log('=== HelloWorldPublisher start');

  let participant = null;
  try {
    participant = new dds.Participant();

    const topicName = 'HelloWorldData_Msg';
    const idlName = 'HelloWorldData.idl';
    const idlPath = __dirname + path.sep + idlName;

    const typeSupports = await dds.importIDL(idlPath);
    const typeSupport = typeSupports.get('HelloWorldData::Msg');

    const tqos = dds.QoS.topicDefault();

    tqos.durability = {kind: dds.DurabilityKind.Transient};
    tqos.reliability = {kind: dds.ReliabilityKind.Reliable};

    const topic = participant.createTopic(
        topicName,
        typeSupport,
        tqos
    );

    //...
  } finally {
    console.log('=== Cleanup resources');
    if (participant !== null){
      participant.delete().catch((error) => {
        console.log('Error cleaning up resources: '
          + error.message);
      });
    }
  }
};
```

5.4 Publisher

The Node.js `Publisher` class represents a DDS publisher entity.

A `Publisher` typically owns one or more Writer instances created via the `Publisher.createWriter()` method.

Class instances are created by the `Participant.createPublisher()` method.

Use of the `Publisher` class is optional. You typically create a `Publisher` because you require quality-of-service parameters not available on the 'default publisher'. Frequently, you wall want to specify a QoS.partition policy so that non-default partitions are used by the contained Writer instances.

Note: An explicit `delete()` is not required for the `Publisher`. When `delete()` is called on the owning `Participant`, the `Publisher delete()` is triggered.

Example: Create a Publisher

```
//...

const pub = participant.createPublisher();
```

Create a publisher with a participant and QoS profile. Consider the code snippet below taken from example: GetSetQoSExample/GetSetQoSExample.js file.

```
//...

// get publisher qos from previously created qos provider: qp
const pubqos = qp.getPublisherQos();

// create a publisher
const publisher = participant.createPublisher(pubqos);
```

5.5 Writer

The Node.js `Writer` class represents a DDS data writer entity.

A `Writer` may be owned by either a `Participant` or `Publisher`. Using a `Publisher` to create a `Writer` allows the writer to benefit from quality-of-service policies assigned the the `Publisher`, in particular the QoS.partition policy.

Class instances are created by the `Participant.createWriter()` or `Publisher.createWriter()` methods.

Note: An explicit `delete()` is not required for the `Writer`. When `delete()` is called on the owning `Participant`, the `Writer delete()` is triggered.

Example: Create a Writer and write sample

```
const dds = require('vortexdds');

//...

async function writeData(publisher, topic){
    const wqos = dds.QoS.writerDefault();

    wqos.durability = {kind: dds.DurabilityKind.Transient};
    wqos.reliability = {kind: dds.ReliabilityKind.Reliable};
    const writer = publisher.createWriter(topic, wqos);

    // send one message
    const msg = {userID: 1, message: 'Hello World'};
    await writer.writeReliable(msg);

    //writer will be deleted on participant delete
}
```

5.6 Subscriber

The Node.js `Subscriber` class represents a DDS subscriber entity.

A `Subscriber` typically owns one or more `Reader` instances created via the `Subscriber.createReader()` method.

Class instances are created by the `Participant.createSubscriber()` method.

You typically create a `Subscriber` because you require quality-of-service parameters not available on the 'default subscriber'. Frequently, you will want to specify a QoS.partition policy so that non-default partitions are used by the contained `Reader` instances.

Note: An explicit `delete()` is not required for the `Subscriber`. When `delete()` is called on the owning `Participant`, the `Subscriber delete()` is triggered.

Example: Create a Subscriber

```
//...

//create Subscriber
const sub = participant.createSubscriber();
```

Create a subscriber with participant and QoS profile.

```
//...

// get subscriber qos from previously created qos provider: qp
const subqos = qp.getSubscriberQos();

// create a subscriber
const subscriber = participant.createSubscriber(subqos);
```

5.7 Reader

The Node.js `Reader` class represents a DDS data reader entity.

A `Reader` may be owned by either a `Participant` or `Subscriber`. Using a `Subscriber` to create a `Reader` allows the reader to benefit from quality-of-service policies assigned the the `Subscriber`, in particular the QoS.partition policy.

Class instances are created by the `Participant.createReader()` or `Subscriber.createReader()` methods.

Note: An explicit `delete()` is not required for the `Reader`. When `delete()` is called on the owning `Participant`, the `Reader delete()` is triggered.

Example: Create a reader and take data*

```
const dds = require('vortexdds');

//...

function readData(subscriber, topic) {
    const rqos = dds.QoS.readerDefault();

    rqos.durability = {kind: dds.DurabilityKind.Transient};
    rqos.reliability = {kind: dds.ReliabilityKind.Reliable};
    const reader = subscriber.createReader(topic, rqos);

    let takeArray = reader.take(10);

    //...

}
```

5.7.1 QueryCondition

`QueryCondition` class represents a condition on `Reader` input based on samples, instances and view state AND on values found in the actual samples.

A query condition is attached to the `Reader` entity and created using the `Reader.createQueryCondition()` function.

Note: An explicit `delete()` is not required for the `QueryCondition`. When `delete()` is called on the owning `Reader`, the `QueryCondition delete()` is triggered.

Example: Read using a QueryCondition

```
const dds = require('vortexdds');

//...

async function readBlueCircleWriteRedSquare(
  circleReader,
  squareWriter
) {
  let queryCond = null;
  let queryWaitset = null;
  try {
    // set up a waitset on our circle reader for the query condition
    // (shape read = blue circle)
    const mask = dds.StateMask.sample.not_read;
    const sqlExpression = 'color=%0';
    const params = ['BLUE'];

    queryCond = circleReader.createQueryCondition(mask, sqlExpression, params);
    queryWaitset = new dds.Waitset(queryCond);

    for (let i = 0; i < 100; i++) {
      await queryWaitset.wait(10000000000);
      let sampleArray = circleReader.takeCond(1, queryCond);
      if (sampleArray.length > 0 && sampleArray[0].info.valid_data) {
        let sample = sampleArray[0].sample;
        console.log(
          util.format(
            '%s %s of size %d at (%d,%d)',
            sample.color,
            'Circle',
            sample.shapesize,
            sample.x,
            sample.y
          )
        );
        squareWriter.write({
          color: 'RED',
          x: sample.x,
          y: sample.y,
          shapesize: 45,
        });
      }
    }

  } finally {
    if (queryWaitset !== null){
      queryWaitset.delete();
    }
  }
```

5.7.2 ReadCondition

`ReadCondition` represents a condition on `Reader` input based on samples, instances and view state.

`ReadCondition` is used by the Reader to wait for the availablity of data based on a condition. A read condition is attached to the `Reader` entity and created using the `Reader.createReadCondition()` function.

Note: An explicit `delete()` is not required for the `ReadCondition`. When `delete()` is called on the owning `Reader`, the `ReadCondition delete()` is triggered.

Example: Read data with read condition

```
const dds = require('vortexdds');

//...

function readCondition(reader) {

    const maxSamples = 100;
    const cond = reader.createReadCondition(dds.StateMask.sample.not_read);
    const readArray = reader.readCond(maxSamples, cond);

    //...
}
```

5.8 WaitSet

`WaitSet` class represents a collection of Conditions upon which you can wait.

A WaitSet object allows an application to wait until one or more of the attached Condition objects evaluates to true or until the timeout expires.

Note: An explicit `delete()` is required for the `WaitSet` to release DDS resources.

Example Create a WaitSet with a ReadCondition

```
const dds = require('vortexdds');

//...

async function waitExample(reader) {

    let newDataCondition = null;
    let newDataWaitset = null;
    try {
      // create waitset for new data
      newDataCondition = reader.createReadCondition(
        dds.StateMask.sample.not_read
      );
      newDataWaitset = new dds.Waitset(newDataCondition);

      await newDataWaitset.wait();

      //...

      } finally {
          if (newDataWaitset !== null){
            newDataWaitset.delete();
          }
      }
}
```

5.9 StatusCondition

`StatusCondition` class represents a condition on an Entities 'communication statuses'.

It is created using the `createStatusCondition()` function on a dds entity instance.

Note: An explicit `delete()` is not required for the `StatusCondition`. When `delete()` is called on the owning `Entity`, the `StatusCondition delete()` is triggered.

Example: Create a StatusCondition

```
const dds = require('vortexdds');

//...

const condition = squareWriter.createStatusCondition(
    dds.StatusMask.publication_matched
);

//...
```

5.10 GuardCondition

`GuardCondition` class is a user-triggerable condition for interrupting `Waitsets`.

A `GuardCondition`, when attached to a `Waitset`, enables you to interrupt an asynchronous `Waitset.wait()` operation by calling the `GuardCondition.trigger()` method.

Note: An explicit `delete()` is required for the `GuardCondition` to release DDS resources.

Example

Create a guard condition and attach it to a waitset.

```
const dds = require('vortexdds');

const ws = new dds.Waitset();
const guard = new dds.GuardCondition();
ws.attach(guard);
// ws.attach(other conditions);
ws.wait()
.then(triggeredConds => {
    for(const cond of triggeredConds) {
        if(cond === guard) {
            // guard was triggered
        }
    }
})
.catch(err => {
    // wait set error, including timeout
});

// sometime later, cause the wait set to trigger.
guard.trigger();
```

6 Listener

The Listener provides a generic mechanism (a callback function) for DDS to notify an application of relevant asynchronous status change events, such as incoming data, a rejected sample or a missed deadline.

The Listener is related to changes in communication status. Each DCPS entity type supports its own specialized listener.

Note: Listeners are currently not supported for the Participant, Publisher and Subscriber entities in the NodeJS DCPS API.

6.1 Attaching a listener to an entity

On entity creation, a listener can be specified. By default the listener is set to null.

A listener is defined using a javascript object. One or more listener properties can be added. For each property, the listener property name and its callback function are defined. Each entity type supports its own listener functionality.

Please see ListenerExample for a runnable code example.

Example: Creating a writer with listener for onPublicationMatched

```
const writer = participant.createWriter(
  topic,
  null,
  {
    onPublicationMatched: function(entity, status) {
        console.log('== onPublicationMatched listener triggered');
        console.log('    status.totalCount: ', status.totalCount);
        console.log('    status.totalCountChange: ', status.totalCountChange);

        //...
    },
  }
);
```

Example: Creating a reader with listener with multiple properties

```
const reader = participant.createReader(
  topic,
  null,
  {
    onSubscriptionMatched: function(entity, status) {
        console.log('== onSubscriptionMatched listener triggered');
        //...
    },
    onDataAvailable: function(entity) {
        console.log('== onDataAvailable listener triggered');
        //...
    },
  }
);
```

6.2 Listeners

This section outlines the different listener properties supported for each entity type.

For more detailed information, please refer to the Node.js DCPS API documentation, found in:

$OSPL_HOME/docs/nodejs/html

6.2.1 Topic Entity

onInconsistentTopic: function(topic, status)

This function is called when another topic exists with the same name but different characteristics.

6.2.2 Reader Entity

onRequestedDeadlineMissed: function(reader, status)

This function is called when the deadline that the Reader was expecting through its QoS policy was not respected for a specific instance.

onRequestedIncompatibleQos: function(reader, status)

This function is called when a QoS policy requested is incompatible with the offered QoS policy by the Writer entity.

onSampleRejected: function(reader, status)

This function is when a received sample was rejected. This can happen when the ResourceLimits Qos is active and the History determined by History QoS of the Reader entity is full.

onLivelinessChanged: function(reader, status)

This function is called when the liveliness of one or more Writers that were writing instances read through the Reader has changed. Some Writer have become "active" or "inactive."

onDataAvailable: function(reader)

This function is called when new data is available for this Reader.

onSubscriptionMatched: function(reader, status)

This function is called when the Reader has found a Writer that matches the Topic and has compatible QoS, or has ceased to be matched with a Writer that was previously considered to be matched.

onSampleLost: function(reader, status)

This function is called when a sample has been lost (never received).

6.2.3 Writer Entity

onOfferedDeadlineMissed: function(writer, status)

This function is called when the deadline that the Writer has committed through its Qos policy DEADLINE was not respected for a specific instance.

onOfferedIncompatibleQos: function(writer, status)

This function is called when a Qos policy value was incompatible with what was requested.

onLivelinessLost: function(writer, status)

This function is called when the liveliness that the Writer has committed through its Qos policy LIVELINESS was not respected; thus Reader entities will consider the Writer as no longer "active".

onPublicationMatched: function(writer, status)

This function is called when the Writer has found a Reader that matches the Topic and has compatible QoS, or has ceased to be matched with a Reader that was previously considered to be matched.

7

Quality of Service (QoS)

The following section explains how to set the Quality of Service (QoS) for a DDS entity.

Users have two options available to set the QoS for an entity or entities. They can define the QoS settings using an XML file, or they can use the Node.js DCPS APIs. Both of these options are explained.

If a QoS setting for an entity is not set using an xml file or the Node.js DCPS APIs, the defaults will be used. This allows a user the ability to override only those settings that require non-default values.

7.1 Setting QoS Using QoS Provider XML File

QoS for DDS entities can be set using XML files based on the XML schema file QoSProfile.xsd. These XML files contain one or more QoS profiles for DDS entities. OSPL includes an XSD (XML schema), that is located in $OSPL_HOME/etc/DDS_QoSProfile.xml. This can be used with XML schema core editors to help create valid XML files.

Sample QoS Profile XML files can be found in the examples directory. Typically you will place the qos files in a subdirectory of your Node.js application.

7.1.1 QoS Profile

A QoS profile consists of a name and optionally a base_name attribute. The base_name attribute allows a QoS or a profile to inherit values from another QoS or profile in the same file. The file contains QoS elements for one or more DDS entities.

A skeleton file without any QoS values is displayed below to show the structure of the file.

```
<dds xmlns="http://www.omg.org/dds/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="file:DDS_QoSProfile.xsd">
    <qos_profile name="DDS QoS Profile Name">
        <datareader_qos></datareader_qos>
        <datawriter_qos></datawriter_qos>
        <domainparticipant_qos></domainparticipant_qos>
        <subscriber_qos></subscriber_qos>
        <publisher_qos></publisher_qos>
        <topic_qos></topic_qos>
    </qos_profile>
</dds>
```

Example: Persistent QoS XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns="http://www.omg.org/dds/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="file:DDS_QoSProfile.xsd">
<qos_profile name="DDS PersistentQosProfile">
     <domainparticipant_qos>
          <user_data>
```

```
                <value></value>
            </user_data>
            <entity_factory>
                <autoenable_created_entities>true</autoenable_created_entities>
            </entity_factory>
        </domainparticipant_qos>
        <subscriber_qos name="subscriber1">
            <presentation>
                <access_scope>INSTANCE_PRESENTATION_QOS</access_scope>
                <coherent_access>true</coherent_access>
                <ordered_access>true</ordered_access>
            </presentation>
            <partition>
                <name>partition1</name>
            </partition>
            <group_data>
                <value></value>
            </group_data>
            <entity_factory>
                <autoenable_created_entities>true</autoenable_created_entities>
            </entity_factory>
        </subscriber_qos>
        <subscriber_qos name="subscriber2">
            <presentation>
                <access_scope>INSTANCE_PRESENTATION_QOS</access_scope>
                <coherent_access>false</coherent_access>
                <ordered_access>false</ordered_access>
            </presentation>
            <partition>
                <name></name>
            </partition>
            <group_data>
                <value></value>
            </group_data>
            <entity_factory>
                <autoenable_created_entities>true</autoenable_created_entities>
            </entity_factory>
        </subscriber_qos>
        <publisher_qos>
            <presentation>
                <access_scope>INSTANCE_PRESENTATION_QOS</access_scope>
                <coherent_access>true</coherent_access>
                <ordered_access>true</ordered_access>
            </presentation>
            <partition>
                <name>partition1</name>
            </partition>
            <group_data>
                <value></value>
            </group_data>
            <entity_factory>
                <autoenable_created_entities>true</autoenable_created_entities>
            </entity_factory>
        </publisher_qos>
        <datawriter_qos>
            <durability>
                <kind>PERSISTENT_DURABILITY_QOS</kind>
            </durability>
            <deadline>
                <period>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
                </period>
```

```xml
         </deadline>
         <latency_budget>
              <duration>
                   <sec>0</sec>
                   <nanosec>0</nanosec>
              </duration>
         </latency_budget>
         <liveliness>
              <kind>AUTOMATIC_LIVELINESS_QOS</kind>
              <lease_duration>
                   <sec>DURATION_INFINITE_SEC</sec>
                   <nanosec>DURATION_INFINITE_NSEC</nanosec>
              </lease_duration>
         </liveliness>
         <reliability>
              <kind>RELIABLE_RELIABILITY_QOS</kind>
              <max_blocking_time>
                   <sec>0</sec>
                   <nanosec>100000000</nanosec>
              </max_blocking_time>
        </reliability>
        <destination_order>
             <kind>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</kind>
         </destination_order>
         <history>
              <kind>KEEP_LAST_HISTORY_QOS</kind>
              <depth>100</depth>
         </history>
         <resource_limits>
              <max_samples>LENGTH_UNLIMITED</max_samples>
              <max_instances>LENGTH_UNLIMITED</max_instances>
              <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
         </resource_limits>
         <transport_priority>
              <value>0</value>
         </transport_priority>
         <lifespan>
              <duration>
                   <sec>DURATION_INFINITE_SEC</sec>
                   <nanosec>DURATION_INFINITE_NSEC</nanosec>
              </duration>
         </lifespan>
         <user_data>
              <value></value>
        </user_data>
        <ownership>
             <kind>SHARED_OWNERSHIP_QOS</kind>
         </ownership>
         <ownership_strength>
              <value>0</value>
         </ownership_strength>
         <writer_data_lifecycle>
              <autodispose_unregistered_instances>true</autodispose_unregistered_instances>
         </writer_data_lifecycle>
    </datawriter_qos>
    <datareader_qos>
         <durability>
             <kind>PERSISTENT_DURABILITY_QOS</kind>
         </durability>
         <deadline>
              <period>
                   <sec>DURATION_INFINITE_SEC</sec>
                   <nanosec>DURATION_INFINITE_NSEC</nanosec>
```

```
               </period>
          </deadline>
          <latency_budget>
               <duration>
                    <sec>0</sec>
                    <nanosec>0</nanosec>
               </duration>
          </latency_budget>
          <liveliness>
               <kind>AUTOMATIC_LIVELINESS_QOS</kind>
               <lease_duration>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
               </lease_duration>
          </liveliness>
          <reliability>
               <kind>BEST_EFFORT_RELIABILITY_QOS</kind>
               <max_blocking_time>
                    <sec>0</sec>
                    <nanosec>100000000</nanosec>
               </max_blocking_time>
          </reliability>
          <destination_order>
                <kind>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</kind>
          </destination_order>
          <history>
               <kind>KEEP_ALL_HISTORY_QOS</kind>
          </history>
          <resource_limits>
               <max_samples>LENGTH_UNLIMITED</max_samples>
               <max_instances>LENGTH_UNLIMITED</max_instances>
               <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
          </resource_limits>
          <user_data>
               <value></value>
          </user_data>
          <ownership>
               <kind>SHARED_OWNERSHIP_QOS</kind>
          </ownership>
          <time_based_filter>
               <minimum_separation>
                    <sec>0</sec>
                    <nanosec>0</nanosec>
               </minimum_separation>
          </time_based_filter>
          <reader_data_lifecycle>
               <autopurge_nowriter_samples_delay>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
               </autopurge_nowriter_samples_delay>
               <autopurge_disposed_samples_delay>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
               </autopurge_disposed_samples_delay>
          </reader_data_lifecycle>
     </datareader_qos>
     <topic_qos>
          <topic_data>
               <value></value>
          </topic_data>
          <durability>
               <kind>PERSISTENT_DURABILITY_QOS</kind>
          </durability>
          <durability_service>
               <service_cleanup_delay>
                    <sec>3600</sec>
                    <nanosec>0</nanosec>
               </service_cleanup_delay>
               <history_kind>KEEP_LAST_HISTORY_QOS</history_kind>
               <history_depth>100</history_depth>
               <max_samples>8192</max_samples>
               <max_instances>4196</max_instances>
               <max_samples_per_instance>8192</max_samples_per_instance>
          </durability_service>
          <deadline>
               <period>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
               </period>
          </deadline>
          <latency_budget>
               <duration>
                    <sec>0</sec>
                    <nanosec>0</nanosec>
               </duration>
          </latency_budget>
          <liveliness>
               <kind>AUTOMATIC_LIVELINESS_QOS</kind>
               <lease_duration>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
               </lease_duration>
          </liveliness>
          <reliability>
               <kind>BEST_EFFORT_RELIABILITY_QOS</kind>
               <max_blocking_time>
                    <sec>0</sec>
                    <nanosec>100000000</nanosec>
                    </max_blocking_time>
          </reliability>
          <destination_order>
               <kind>BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</kind>
          </destination_order>
          <history>
               <kind>KEEP_LAST_HISTORY_QOS</kind>
               <depth>1</depth>
          </history>
              <resource_limits>
               <max_samples>LENGTH_UNLIMITED</max_samples>
               <max_instances>LENGTH_UNLIMITED</max_instances>
               <max_samples_per_instance>LENGTH_UNLIMITED</max_samples_per_instance>
          </resource_limits>
          <transport_priority>
               <value>0</value>
          </transport_priority>
          <lifespan>
               <duration>
                    <sec>DURATION_INFINITE_SEC</sec>
                    <nanosec>DURATION_INFINITE_NSEC</nanosec>
              </duration>
          </lifespan>
          <ownership>
               <kind>SHARED_OWNERSHIP_QOS</kind>
          </ownership>
     </topic_qos>
</qos_profile>
```

```
</dds>
```

7.1.2 Applying QoS Profile

To set the QoS profile for a DDS entity using the Node.js DCPS API and an XML file, the user specifies the qos file URI or file path and the QoS profile name as parameters.

Example: Using QoSProvider

```
const dds = require('vortexdds');
const path = require('path');

const QOS_XML_PATH = __dirname + path.sep + 'DDS_Get_Set_QoS.xml';
const QOS_PROFILE = 'DDS GetSetQosProfile';
const DOMAIN_ID = dds.DDS_DOMAIN_DEFAULT;

//...

async function setup(){

    let participant = null;
    let qp = null;
    try {
        // create a qos provider using qos xml file
        qp = new dds.QoSProvider(QOS_XML_PATH, QOS_PROFILE);

        // get participant qos from qos provider and create a participant
        const pqos = qp.getParticipantQos();

        participant = new dds.Participant(DOMAIN_ID, pqos);

        // get publisher qos from qos provider
        const pubqos = qp.getPublisherQos();

        // get publisher qos policies
        const pubScope = pubqos.presentation;
        const publisher = participant.createPublisher(pubqos);

        //...

    } finally {
        console.log('=== Cleanup resources');
        if (qp !== null){
           qp.delete();
        }
        if (participant !== null){
          participant.delete().catch((error) => {
            console.log('Error cleaning up resources: '
              + error.message);
          });
        }
    }
}
```

7.2 Setting QoS Using Node.js DCPS API Classes

QoS settings can also be set by using the Node.js classes alone. (No XML files required.)

Example: Create Reader Using QoS APIS

```
const dds = require('vortexdds');

//...

function createReader(subscriber, topic){

    // get the default reader qos
    const rqos = dds.QoS.readerDefault();

    // modify the reader qos policies
    rqos.partition = {names: 'partition1'};
    rqos.timebasedFilter = {minimumSeparation: 60000};
    rqos.readerDataLifecycle = {
        autopurgeNoWriterSamples: 100,
        autopurgeDisposedSamplesDelay: 500,
    };

    // create a reader with qos
    const reader = subscriber.createReader(topic, rqos);

    //...
}
```

8

Topic Generation and Discovery

A DDS Topic represents the unit for information that can be produced or consumed by a DDS application. Topics are defined by a name, a type, and a set of QoS policies.

The Node.js DCPS API provides several ways of generating Node.js classes to represent DDS topics.

- over the wire discovery
- dynamic generation of Node.js classes using parameters IDL file and topic name

---

Note:

- The *Examples* section provides the examples directory location, example descriptions and running instructions.

8.1 Over the Wire Discovery

Node.js topic classes can be generated for existing DDS topics in the DDS system. These topics are "discovered over the wire".

The Node.js classes are generated when the topic is requested by name.

A code snippet is provided from findTopicExample.js. This example finds a topic registered by another process, and writes a sample to that topic.

Example: findTopic

```
const dds = require('vortexdds');
const dp = someAlreadyCreatedParticipant;
dp.findTopic('MyTopic')
  .then(topic => {
    // do something with the found topic
})
.catch(err => {
    // process any exception
})
.then(_ => {
    // cleanup: do any processing necessary after both
    // success and failure
});
```

8.2 Dynamic Generation of Node.js Topic Classes Using IDL and Name

The Node.js DCPS API supports generation of Node.js topic classes from IDL. This section describes the details of the IDL-Node.js binding.

8.2.1 Dynamic Generation

The Node.js DCPS API provides an asynchronous function that returns a Map of `TypeSupport` objects.

A `TypeSupport` object includes the topic typename, keys and descriptor.

The structure type representation of a topic is created by the `TypeSupport` object. However, the usage of the structure type is internal to the Node.js DCPS API.

In order to create a `Topic`, a topic name and a `TypeSupport` are passed into the `Participant.createTopic` function. (Qos and listener parameters are optional)

Example: Import a Topic from IDL

```
const dds = require('vortexdds');
const path = require('path');

//...

async function createTopic(participant) {

  const topicName = 'IoTData';
  const idlName = 'dds_IoTData.idl';
  const idlPath = path.resolve(idlName);

  //wait for dds.importIDL to return a map of typeSupports
  let typeSupports = await dds.importIDL(idlPath);
  let typeSupport = typeSupports.get('DDS::IoT::IoTData');

  return participant.createTopic(topicName, typeSupport);
};
```

8.2.2 Generated Artifacts

The following table defines the Node.js artifacts generated from IDL concepts:

IDL Concept	Node.js Concept
module	N/A
enum	enum from npm 'enum' package
enum value	enum value
struct	object
field	object property
union	object (IoTValue from dds_IoTData.idl is the only supported union)

Datatype mappings

The following table shows the Node.js equivalents to IDL primitive types:

IDL Type	Node.js Type
boolean	Boolean
char	Number
octet	Number
short	Number
ushort	Number
long	Number
ulong	Number
long long	Number
ulong long	Number
float	Number
double	Number
string	String
wchar	Unsupported
wstring	Unsupported
any	Unsupported
long double	Unsupported

Implementing Arrays and Sequences in Node.js

Both IDL arrays and IDL sequences are mapped to JavaScript arrays.

8.3 Limitations of Node.js Support

The Node.js binding has the following limitations:

- Listener is not supported for the Participant, Publisher and Subscriber entities.
- Only the IoTValue union from dds_IoTData.idl is supported.
- JavaScript does not currently include standard support for 64-bit integer values. 64-bit integers with more than 53 bits of data are represented by String values to avoid loss of precision. If the value will fit inside a JavaScript Number without losing precision, a Number can be used, otherwise use a String. (Refer to IoTData example which demonstrates the usage and ranges for the unsigned and signed 64 bit integers within nodejs.)

9

Contacts & Notices

9.1 Contacts

ADLINK Technology Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA
Tel: +1 781 569 5819

ADLINK Technology Limited
The Edge
5th Avenue
Team Valley
Gateshead
NE11 0XA
UK
Tel: +44 (0)191 497 9900

ADLINK Technology SARL
28 rue Jean Rostand
91400 Orsay
France
Tel: +33 (1) 69 015354

Web: http://ist.adlinktech.com/

Contact: http://ist.adlinktech.com

E-mail: ist_info@adlinktech.com

LinkedIn: https://www.linkedin.com/company/79111/

Twitter: https://twitter.com/ADLINKTech_usa

Facebook: https://www.facebook.com/ADLINKTECH

9.2 Notices