ADLINK

User Guide

Release 2.0.14

Contents

1

Introduction

Vortex Lite offers full DDSI rev2.1 interoperability with enhanced real-time performance. Vortex Lite brings real-time data sharing to resource constrained embedded devices.

Vortex Lite is the small footprint, lowest end to end latency DDS implementation of the Vortex product suite.

At the heart of Vortex Lite is the Vortex OpenSplice DDSI implementation, giving the advantage of a robust and battle-hardened codebase.

Vortex Lite's main benefits are:

- Minimal resource-consumption with regard to CPU- and Memory-usage.
- Allows variability on functionalities, transport and support of underlying OS / BSP
- Deterministic data delivery: data urgency / importance based network-scheduling
- Networking efficiency: configurable *networkPartitions* allowing to partition the physical network
- Vortex Lite comes with 3 APIs:
  - C99 API, a new easy to use C API for DDS
  - ISO-C++ DCPS API, the latest C++ specification for DDS.
  - Classic DCPS C++ API, the original C++ specification for DDS

These APIs are compliant with the Vortex OpenSplice C++ APIs to allow for easier migration of code. As of Lite version 2, the updated ISO C++ V2 API is supported (also available in OpenSplice as of version 6.6).

This User Guide will get you started with Vortex Lite development.

Detailed information about ADLINK's product support services, general support contacts and enquiries are described on the ADLINK Support page reached via the ADLINK Home page at http://ist.adlinktech.com/.

2

Installation and Licensing

2.1 Installing the software

2.1.1 Installation for UNIX and Windows Platforms

Install Vortex Lite by running the installation wizard for your particular installation, using:

P<platform_code>-VortexLite-<version>-installer.<ext>

where

- <platform_code> - the platform code, specific to each host/target architecture
- <version> - the Vortex Lite version number, for example 1.0.0
- <ext> - the platform executable extension, either empty for unix systems, .exe for windows or .tar.gz where an installation package cannot be created.

For a tar.gz, just unpack that to the required location.

The directories in the Vortex Lite distribution are named after the installation package they contain. Each package consists of an archive and its installation procedure.

2.1.2 Setting the User Environment

Within the Vortex Lite install directory there will be a setup file that can be sourced. In addition to this, on Windows platforms we provide a batch file.

1. Go to the install directory
2. Source the setup file or run the setup batch file.

Unix:

```
source ./setup
```

Windows:

```
setup.bat
```

Please note on Windows platforms the installer will automatically set LITE_HOME globally. If installing multiple versions of Vortex Lite you will need to ensure that this is set to the correct location when building the examples.

2.2 Licensing

This section describes how to install a license file for a Vortex product and how to use the license manager.

Vortex uses Reprise License Manager (RLM) to manage licenses.

The licensing software is automatically installed on the host machine as part of the Vortex distribution. The software consists of two parts:

- Vortex binary files, which are installed in <Lite_Install_Dir>/bin where <Lite_Install_Dir> is the directory where Vortex Lite is installed.
- License files which determine the terms of the license. These will be supplied by ADLINK.

ADLINK supplies a Vortex license file, license.lic. This file is not included in the software distribution, but is sent separately by ADLINK.

2.2.1 Development and Deployment Licenses

Development licenses for Vortex Device products (Enterprise, Café, Web and Lite) are provided on a 'per user' basis. This means that each developer using the product requires a separate valid license to use the product. Vortex is physically licensed for development purposes.

Vortex Cloud and Gateway are also physically licensed, and each product requires a valid deployment license to use it in an operational or a production environment.

2.2.2 Installing the License File

Copy the license file to <Vortex_Install_Dir>/license/license.lic where <Vortex_Install_Dir> is the directory where Vortex is installed, on the machine that will run the license manager. This is the recommended location for the license file but you can put the file in any location that can be accessed by the license manager rlm.

Lite will additionally search the following hierarchy for the license file:

On Unix platforms:

- ADLINK_LICENSE environment variable
- RLM_LICENSE environment variable
- <Lite_Install_Dir>/etc

On Windows platforms:

- ADLINK_LICENSE environment variable
- RLM_LICENSE environment variable
- <Lite_Install_Dir>etc

The environment variables RLM_LICENSE and ADLINK_LICENSE must be set to the full path and file name of the license file. (Note that either variable can be used; there is no need to set both.) For example:

ADLINK_LICENSE=/my/lic/dir/license.lic

If licenses are distributed between multiple license files, the RLM_LICENSE or ADLINK_LICENSE variable can be set to point to the directory which contains the license files.

2.2.3 Running the License Manager Daemon

It is only necessary to run the License Manager Daemon for floating or counted licenses. In this case, the license manager must be running before Vortex can be used. The license manager software is responsible for allocating licenses to developers and for ensuring that the allowed number of concurrent licenses is not exceeded. For node-locked licenses, as is the case with all evaluation licenses, then it is not necessary to run the License Manager Daemon but the RLM_LICENSE or ADLINK_LICENSE variable must be set to the correct license file location. To run the license manager, use the following command:

```
rlm -c <location>
```

where <location> is the full path and filename of the license file. If licenses are distributed between multiple files, <location> should be the path to the directory that contains the license files.

The rlm command will start the ADLINK vendor daemon prismtech, which controls the licensing of the Vortex software.

To obtain a license for Vortex from a License Manager Daemon that is running on a different machine, set either the RLM_LICENSE or ADLINK_LICENSE environment variable to point to the License Manager Daemon, using the following syntax:

```
RLM_LICENSE=<port>@<host>
```

where <port> is the port the daemon is running on and <host> is the host the daemon is running on.

The port and host values can be obtained from the information output when the daemon is started. The format of this output is as shown in the following example:

```
07/05 12:05 (rlm) License server started on rhel4e
07/05 12:05 (rlm) Server architecture: x86_l2
07/05 12:05 (rlm) License files:
07/05 12:05 (rlm) license.lic
07/05 12:05 (rlm)
07/05 12:05 (rlm) Web server starting on port 5054
07/05 12:05 (rlm) Using TCP/IP port 5053
07/05 12:05 (rlm) Starting ISV servers:
07/05 12:05 (rlm) ... prismtech on port 35562
07/05 12:05 (prismtech) RLM License Server Version 9.1BL3 for ISV "prismtech"
07/05 12:05 (prismtech) Server architecture: x86_l2
Copyright (C) 2006-2011, Reprise Software, Inc. All rights reserved.
RLM contains software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http//www.openssl.org)
Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
Copyright (c) 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved
07/05 12:05 (prismtech)
07/05 12:05 (prismtech) Server started on rhel4e (hostid: 0025643ad2a7) for:
07/05 12:05 (prismtech) opensplice_product1 opensplice_product2
07/05 12:05 (prismtech)
07/05 12:05 (prismtech) License files:
07/05 12:05 (prismtech) license.lic
07/05 12:05 (prismtech)
```

The <server> value should be taken from the first line of the output. The <port> value should be taken from the line reading "... prismtech on port xxxxx". From this example, the value for RLM_LICENSE or ADLINK_LICENSE would be '35562@rhel4e'.

2.2.4 Utilities

A utility program, rlmutil, is available for license server management and administration. One feature of this utility is its ability to gracefully shut down the license manager. To shut down the license manager, preventing the checkout of licenses for the Vortex software, run either of the following commands:

```
% rlmutil rlmdown -vendor prismtech
% rlmutil rlmdown -c <location>
```

where <location> is the full path and filename of the license file.

3

Getting Started with Vortex Lite

3.1 Documentation

To access the product documentation, view the index.html file in your installation directory. This contains details of the:

- Release Notes
- API Reference Guide
- User Guide
- Configuration Guide
- Examples

3.2 Examples

To get started with Vortex Lite a set of useful examples have been provided.

On Unix we provide makefiles which can be used to re-build the examples, in addition to this we also provide Visual Studio solution files for Windows.

For more information, see the *examples* directory.

The throughput and roundtrip examples are useful for benchmarking and have been designed to interoperate with the other Vortex product family members; Vortex OpenSplice, Vortex Cafe and Vortex Web.

Please note when running the examples that the working directory must be writeable. For Windows users installing in Program Files we recommend copying the example tree to another location.

3.3 Configuration

Vortex Lite is configured using an XML configuration file. It is advisable to use the *liteconf* tool (UNIX) or Vortex Lite Configurator Tool (Windows Start Menu) to edit your xml files. The configurator tool provides explanations of each attribute and also validates the input. The default configuration file is lite.xml located in <Lite_Install_Dir>/etc (alternative configuration files may also be available in this directory, to assist in other scenarios). The *LITE_URI* environment variable should be set to reference this configuration file. The default value of the environment variable LITE_URI is set to this configuration file. The use of a configuration file is not mandatory.

See the Configuration Guide */docs/confguide/html/index.html* for more information.

3.4 Using the C IDL Compiler

The *dds_idlc* tool parses IDL files defining DDS types and for each one a set of C functions and data is generated in one header (.h) and one code (.c) file.

The command line syntax of *dds_idlc* is as follows:

```
dds_idlc [options] file(s).idl
```

Where the available options are:

-help	Show brief help
-version	Print compiler version
-d *directory*	Output directory for generated files
-I path	Add directory to #include search path
-D *macro*	Define conditional compilation symbol
-E	Preprocess only, to standard output
-allstructs	All structs are Topics
-notopics	Generate type definitions only
-dll name[,file]	Generate DLL linkage declarations
-noxml	Do not generate XML Topic descriptors
-nostamp	Do not timestamp generated code
-lax	Skip over structs containing unsupported datatypes
-quiet	Suppress console output other than error messages
-map_wide	Map the unsupported wchar and wstring types to char and string
-map_longdouble	Map the unsupported long double type to double

The language accepted by *dds_idlc* is IDL, as defined in the OMG CORBA specification. However, many of the constructs available in IDL are not relevant to DDS and these are ignored by the compiler. The primary element of concern is the *struct* definition which is used to define the data types to be supported in the form of DDS topics. The *module* element is also supported to give scoping.

dds_idlc also takes into account all C pre-processor directives that are common to ANSI-C, like *#include*, *#define*, *#ifdef*, etc.

For each struct defined in the specified idl file and marked with a *#pragma keylist* directive, a DDS topic is generated. If the *allstructs* option is used, topics are generated regardless of keylist pragmas. If the *notopics* option is given, topics are not generated, only type definitions. Structures, enumerations and typedefs in idl files incorporated by *#include* may be used, but code is not generated for those.

Not all IDL datatypes are valid for struct members in DDS. *Objects*, the *Any* type and the *ValueBase* type are unsupported. Additionally in Vortex Lite, wide characters, wide strings and the long double type are unsupported. Options are provided to map these to narrower types.

3.4.1 Code generation

For a topic `ExampleModule::Topic` the following items are generated

- A struct `ExampleModule_Topic` defining the data structure for the topic.
- A macro `ExampleModule_Topic__alloc` which allocates memory to hold a single instance of the topic.
- A macro `ExampleModule_Topic_free (d, o)` which deallocates memory. `d` is a pointer to the memory to be released and `o` should be one of DDS_FREE_CONTENTS (just free the topic) or DDS_FREE_ALL (in addition free the elements of contained sequences and arrays).
- Metadata used internally by Vortex Lite.

For a sequence `ExampleModule::seq` the following are generated

- A struct `ExampleModule_seq` which holds a pointer to a buffer of the contained type.

- A macro `ExampleModule_seq_allocbuf (l)` which allocates a buffer of length `l` suitable for use in the struct.

3.4.2 Key definition

The IDL compiler provides a mechanism to mark a struct as a topic, and optionally specify which fields constitute its key. The syntax for the definition is:

```
#pragma keylist <data-type-name> <key>*
```

The identifier <*data-type-name*> is the identification of a struct definition. The identifier <*key*> is a member of the struct. If no keys are specified, the struct is marked as an unkeyed topic.

Example

```
module HelloWorldData
{
  struct Msg
  {
    long userID;
    string message;
  };
  #pragma keylist Msg userID
};
```

Here we define a Topic, `HelloWorldData::Msg`, which has one keyfield, the userID. This will result in the following declaration in the generated header file:

```
typedef struct HelloWorldData_Msg
{
  int32_t userID;
  char * message;
} HelloWorldData_Msg;
```

The generated metadata will include a key description which sets userID as the only field in the key.

Supported Key Types

All fixed size basic types (short, long etc), string types (bounded and unbounded) and arrays of basic type are supported as key types. These key types may be embedded within structs contained within the main topic type.

3.4.3 Building Windows DLLs

If the *-dll name* option is specified, additional code is generated to set the linkage specifiers on the generated functions. These functions will then be dll-exported when the generated C file is built, and will have dll-import linkage when application code includes the generated header file.

If building code which will be linked into the same dll as the generated code, or which will be built to an executable, again also containing the generated code, the functions need to be simply defined as 'extern', as they would be in a Unix-like environment. In order to enable such definitions, when building the application code set the preprocessor macro DDS_BUILD_name_DLL where *name* is the parameter given to the -dll argument.

If the *file* option is specified with -dll, the named file will be included from the generated header file.

3.5 Using the C++ IDL Compiler

The *dds_idlcpp* tool parses IDL files defining DDS types and for each one a set of C++ functions and data is generated in a number of header and code files. These represent the topics specified and the support functions required by Vortex, and in addition the specialized interfaces for *TypeSupport* and the *DataReader* and *DataWriter*.

The command line syntax of *dds_idlcpp* is as follows:

```
dds_idlcpp [options] file(s).idl
```

Where the available options are:

-help	Show brief help
-version	Print compiler version
-isocpp	ISO C++ code generation
-classic	Classic C++ code generation
-d *directory*	Output directory for generated files
-I path	Add directory to #include search path
-D *macro*	Define conditional compilation symbol
-E	Preprocess only, to standard output
-dll name	Generate DLL linkage declarations
-noxml	Do not generate XML Topic descriptors
-nostamp	Do not timestamp generated code
-quiet	Suppress console output other than error messages

These options correspond to their counterparts in *dds_idlc*. The *-classic* and *-isocpp* options select the C++ API for which to compile, with isocpp being the default.

3.6 DDS Link Libraries

All core functionality is contained within a single *dds* library (usually shared), this has no dependencies apart from the standard C runtime and system libraries.

Additional security related features (encryption and ssl support over tcp) are supported via an additional *dds_ssl* library which itself has a dependency on the OpenSSL *crypto* and *ssl* libraries. To use these features applications should be linked with both the *dds* and *dds_ssl* libraries and the ssl runtime support installed with a call to the ssl plugin function:

```
dds_ssl_plugin ();
```

Durability support (for persistent data) is also supported via a pluggable client side library that interacts with the OpenSplice durability service. To enable this an application should link with the *dds_dur* library and install client side durability support with a call to the durability plugin function:

```
dds_durability_plugin ();
```

3.7 Installing Target Libraries

For platforms where applications are cross compiled for a different target RTOS (such as VxWorks), the target libraries need to be deployed to the target system and the target application configured to use them. The exact mechanism to do this is RTOS dependent but usually covered in the associated support documentation. Please contact support if any problems are encountered in this area.

3.8 Valgrind

The Vortex Lite system uses a small amount of memory which is statically allocated and not freed during the lifetime of the application. When using Valgrind to find memory leaks, this statically allocated memory is shown as "still reachable". A suppressions file can be used to remove these indications. The file is: <Lite_Install_Dir>/etc/valgrind.supp

The suppressions file can be used by adding the option:

```
--suppressions=<Lite_Install_Dir>/etc/valgrind.supp
```

to the valgrind command line.

4

Vortex Lite Usage

4.1 DDS API

Vortex Lite implements a proprietary C API with a reduced set of features defined in DDS, see http://www.omg.org/spec/. We call this the C99 API. For a detailed description of this API see */docs/api/c99/index.html*. All code examples in the guide refer to the C API.

In addition, the DCPS C++ API is implemented. This is a C++ mapping of the DCPS specification given in IDL in the DDS standard. We call this the Classic C++ API. For a detailed description of this API see */docs/api/classic_cpp/index.html*.

As of Lite version 2, the updated ISO C++ V2 API is supported (also available in OpenSplice as of version 6.6). For a detailed description of this API see */docs/api/isocpp/index.html*.

4.2 Network protocol

Vortex Lite uses the Real-Time Publish-Subscribe Wire Protocol (DDSI-RTPS).

4.3 Non-standard Features

4.3.1 Filtered Topics

Filters are considered as a logical attribute of a normal topic (there is no separate entity for a filtered topic). Any topic can have a filter set on it. Two operations are supported to set and get the filter on a topic, *dds_topic_set_filter* and *dds_topic_get_filter*. Filters are not SQL based but are functions that take a sample argument and return whether the sample is to be accepted or rejected. Topic filters are applied before a sample is written to the write cache and before a sample is delivered into the read cache.

4.3.2 Error Handling

All DDS functions that can fail in some way return an *int* error status. A return value of *DDS_SUCCESS* (zero) indicates that the function has worked correctly, a negative return value indicates that the function has failed in some way. Vortex Lite encodes three types of information within an error return value, the error category, the error module and a minor number. Functions and macros are provided to decode the error status:

```
const char * dds_err_str (int err);
const char * dds_err_mod_str (int err);
#define dds_err_no(e) (-(-(e) & DDS_ERR_NO_MASK))
#define dds_err_minor(e) ((-(e) & DDS_ERR_MINOR_MASK) >> 16)
```

The *dds_err_str* function returns a string representing the error type, such as “Type Mismatch”. The *dds_err_mod_str* function returns a string representing the module in which the error occurred, such as “Reader”.

The *dds_err_no* and *dds_err_minor* macros return the error number (as defined in dds/error.h) and the minor number respectively. The combination of error status components means that the value of every error code is unique and can be traced to exactly one point in the source code for debugging and support purposes. Generally a single macro *DDS_ERR_CHECK* can be used to check returned error status. For example:

```
int ret;
dds_entity_t ppant;
ret = dds_participant_create (&ppant, DDS_DOMAIN_DEFAULT, NULL, NULL);
DDS_ERR_CHECK (ret, DDS_CHECK_REPORT | DDS_CHECK_EXIT | DDS_CHECK_FAIL);
```

The *DDS_ERR_CHECK* macro takes the error status as its first argument and a bit field as its second that determines how the error is to be handled. If *DDS_CHECK_REPORT* is set then the error is printed to standard output. If *DDS_CHECK_FAIL* is set then any installed failure handle function is called. If *DDS_CHECK_EXIT* is set then the executable exits.

4.3.3 Failure Handling

Failure conditions are defined as something that prevents the continued functioning of an executable, where there is no obvious recovery strategy. For example failure to initialize a mutex variable or to allocate memory. In these cases a pluggable failure routine is invoked. A default implementation of this routine is provided that prints where the program is failing and aborts. The failure handling functions are:

```
typedef void (*dds_fail_fn) (const char *, const char *);
#define DDS_FAIL(m) (dds_fail (m, __FILE__ ":" DDS_INT_TO_STRING (__LINE__)))
void dds_fail_set (dds_fail_fn fn);
dds_fail_fn dds_fail_get (void);
void dds_fail (const char * msg, const char * where);
```

The *dds_fail_set* and *dds_fail_get* functions are used to get and set the failure handler function of type *dds_fail_fn*. The *DDS_FAIL* macro provides a convenience wrapper (including file name and line number) around the *dds_fail* function that delegates the handling of a failure to the installed failure handler.

4.3.4 Time Handling

All times are represented as a 64-bit signed integer, encoding nanoseconds since the epoch. The date of the epoch is system dependent. Time is used in the DDS APIs to manage delays and time outs. The following time APIs are supported:

```
typedef int64_t dds_time_t;
typedef int64_t dds_duration_t;

dds_time_t dds_time (void);
void dds_sleepfor (dds_duration_t n);
void dds_sleepuntil (dds_time_t n);
```

The *dds_time* function returns the current time. The *dds_sleepfor* and *dds_sleepuntil* functions block the calling thread until a relative or absolute time has passed, respectively. To help manage time values a number of macros are provided to do some common time conversions:

```
#define DDS_NSECS_IN_SEC 1000000000LL
#define DDS_NSECS_IN_MSEC 1000000LL
#define DDS_NSECS_IN_USEC 1000LL

#define DDS_NEVER ((dds_time_t) INT64_MAX)
#define DDS_INFINITY ((dds_duration_t) INT64_MAX)
#define DDS_SECS(n) ((n) * DDS_NSECS_IN_SEC)
#define DDS_MSECS(n) ((n) * DDS_NSECS_IN_MSEC)
#define DDS_USECS(n) ((n) * DDS_NSECS_IN_USEC)
```

4.3.5 Instance Handles

Currently only a subset of writer instance handle functionality is supported:

- Writer instance handles must be explicitly created with the *dds_instance_register* function.
- Writer instance handles must either be explicitly deleted with the *dds_instance_unregister* function or implicitly with a *dds_instance_dispose* or *dds_instance_writedispose*.
- Writer instance handles can be used with the following functions:
  - *dds_instance_lookup*
  - *dds_instance_get_key*
- Writer instance handles cannot be used to read or take samples.

4.3.6 Reporting and Tracing

Lite can produce highly detailed traces of all traffic and internal activities. It allows enabling individual categories of information, as well as having a simple verbosity level that enables fixed sets of categories. The categorisation of tracing output is incomplete and hence most of the verbosity levels and categories are not of much use in the current release. This is an ongoing process and here we describe the target situation rather than the current situation. Tracing can be configured in the XML configuration file, by default a log file "lite.log" is generated if any reporting is enabled.

The Tracing element configuration has the following sub elements:

- Verbosity: selects a tracing level by enabled a pre-defined set of categories. The list below gives the known tracing levels, and the categories they enable:
  - *none*
  - *severe*: *error* and *fatal*
  - *warning*: *info*, *severe* and *warning*
  - *config*: *info* and *config*
  - *fine*: *config* and *discovery*
  - *finer*: *fine*, *traffic*, *timing* and *info*
  - *finest*: *fine* and *trace*
- EnableCategory: a comma-separated list of keywords, each keyword enabling individual categories. The following keywords are recognised:
  - *fatal*: all fatal errors, errors causing immediate termination
  - *error*: failures probably impacting correctness but not necessarily causing immediate termination
  - *warning*: abnormal situations that will likely not impact correctness
  - *config*: full dump of the configuration
  - *info*: general informational notices
  - *discovery*: all discovery activity
  - *data*: include data content of samples in traces
  - *radmin*: receive buffer administration
  - *timing*: periodic reporting of CPU loads per thread
  - *traffic*: periodic reporting of total outgoing data

In addition, the keyword *trace* enables all but *radmin*.

- OutputFile: the file to which to write the log (defaults to lite.log)

- AppendToFile: boolean, set to *true* to append to the log instead of replacing the file.

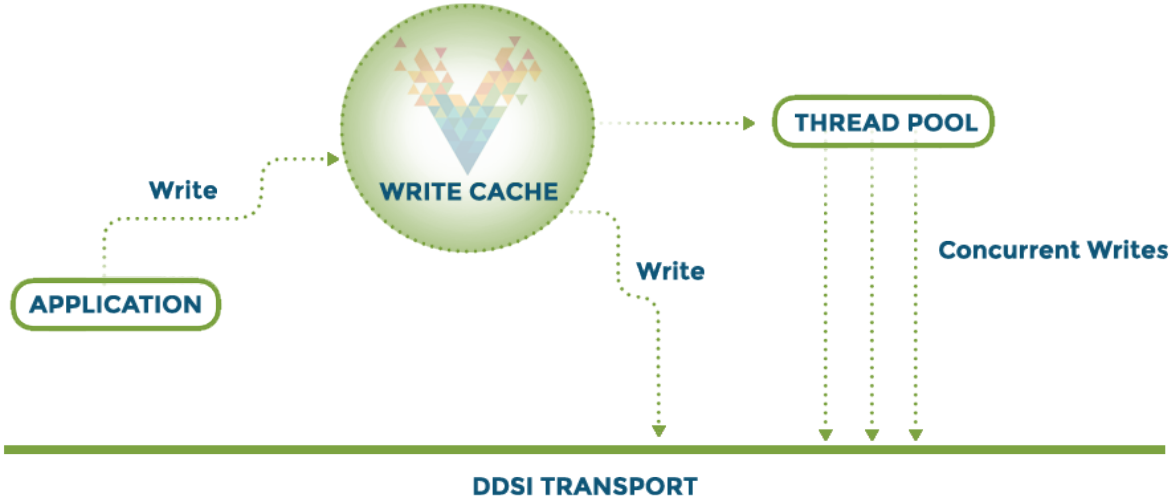
Currently, the useful verbosity settings are *config* and *finest*. *Config* writes the full configuration to the log file as well as any warnings or errors, which can be a good way to verify everything is configured and behaving as expected. *Finest* provides a detailed trace of everything that occurs and is an indispensable source of information when analysing problems; however, it also requires a significant amount of time and results in huge log files. Whether these logging levels are set using the verbosity level or by enabling the corresponding categories is immaterial.

4.4 Threading Model

4.4.1 Publication

Usually no internal threads are used to implement write functionality. By default each application thread that calls a write operation, writes directly through the write cache and out onto the underlying DDSI transport. However if the *WriteBatch* configuration property is set to *true* then a write operation may simply write to the write cache, a subsequent write call may then flush the write cache out to the transport. This behaviour allows multiple small data packets to be aggregated in the write cache into a larger consolidated packet, which optimises throughput at the expense of latency. Note that when running in this mode, data may be left unsent in the write cache after a sequence of write operations. To deal with this scenario a user application can call the *dds_write_flush* function to flush data from the write cache.

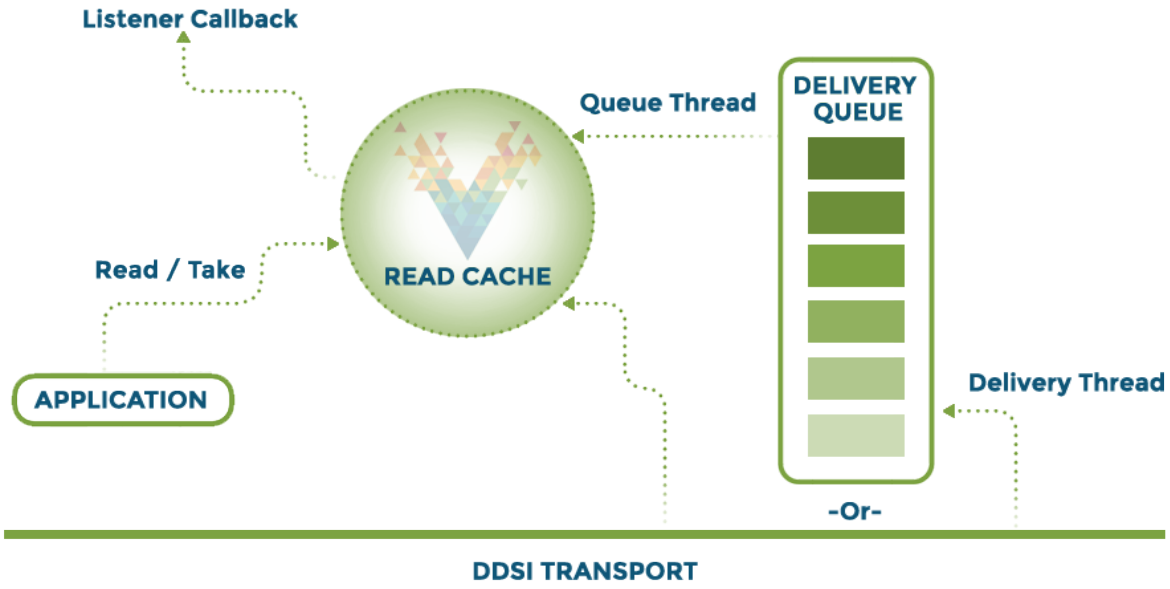
When DDSI has to deliver data to multiple unicast subscribers, the usual behavior is for the delivery thread to deliver to each one in turn. However as this can scale badly it is also possible to configure a DDSI thread pool, where pool threads are used to deliver concurrently to multiple subscribers. This is not usually the case when running a multicast enabled transport such as *udp*, but may give improved performance with unicast based transports such as *tcp*. To enable the DDSI delivery thread pool the *ThreadPool* configuration options are used.



4.4.2 Subscription

A single thread is used to receive incoming packets from the DDSI transport. Under normal operation this thread delivers samples into the read cache and calls onto any registered listeners. However DDSI can be configured to disable this direct delivery and indirect via a delivery queue. When this queue is used a second delivery queue thread takes data from the head of the delivery queue and adds it to the read cache. The delivery thread is also used to make the call back on any registered listeners. This behaviour can be enabled by setting the *SynchronousDeliveryPriorityThreshold* and *SynchronousDeliveryLatencyBound* configuration properties to disable synchronous data delivery. Note that the thread that delivers data to the read cache may block if the data is reliable and the read cache is full (resource limits are set). Data is finally delivered to the application either via a *read/take* from

an application thread or by a *read/take* from a listener function called by the read cache delivery thread (either delivery queue thread or DDSI receive thread).



4.4.3 Application Threads

Each application thread requires some implementation thread specific storage for thread state management. Two functions are provided to manage this state, *dds_thread_init* and *dds_thread_fini*. Every thread created by the application (as opposed to callback threads) should use these functions:

```
application_thread (void * args)
{
  dds_thread_init ("ThreadName");
  ...
  dds_thread_fini ();
}
```

The thread name passed to the *dds_thread_init* function should be unique and is used to identify different threads in generated log files.

5

DDSI Concepts

The DDSI 2.1 standard is very intimately related to the DDS 1.2 standard, with a clear correspondence between the entities in DDSI and those in DCPS. However, this correspondence is not one-to-one. In this section we give a high-level description of the concepts of the DDSI specification, with hardly any reference to the specifics of the Vortex Lite implementation, DDSI2E, which are addressed in the subsequent sections. This division was chosen to aid the reader interested in interoperability in understanding where the specification ends and the Vortex Lite implementation begins.

5.1 Mapping of DCPS Domains to DDSI Domains

In DCPS, a domain is uniquely identified by a non-negative integer, the domain id. DDSI maps this domain id to UDP/IP port numbers to be used for communicating with the peer nodes. These port numbers are particularly important for the discovery protocol, and this mapping of domain ids to UDP/IP port numbers ensures accidental cross-domain communication is impossible with the default mapping.

DDSI does not communicate the DCPS port number in the discovery protocol; it assumes each domain ids maps to unique port numbers. While it is unusual to change the mapping, the specification requires this to be possible, and this means that two different DCPS domain ids can be mapped to a single DDSI domain.

5.2 Mapping of DCPS Entities to DDSI Entities

Each DCPS domain participant in a domain is mirrored in DDSI as a DDSI participant. These DDSI participants drive the discovery of participants, readers and writers in DDSI via the discovery protocols. By default each DDSI participant has a unique address on the network in the form of its own UDP/IP socket with a unique port number.

Any data reader or data writer created by a DCPS domain participant is mirrored in DDSI as a DDSI reader or writer. In this translation, some of the structure of the DCPS domain is lost, because DDSI has no knowledge of DCPS Subscribers and Publishers. Instead, each DDSI reader is the combination of the corresponding DCPS data reader and the DCPS subscriber it belongs to; and similarly, each DDSI writer is a combination of the corresponding DCPS data writer and DCPS publisher. This corresponds to the way the DCPS built-in topics describe the DCPS data readers and data writers, as there are no built-in topics for describing the DCPS subscribers and publishers either.

In addition to the application-created readers and writers (referred to as ‘endpoints’), DDSI participants have a number of DDSI built-in endpoints used for discovery and liveliness checking/asserting. The most important ones are those absolutely required for discovery: readers and writers for the discovery data concerning DDSI participants, DDSI readers and DDSI writers. Some other ones exist as well, and a DDSI implementation can leave out some of these if it has no use for them. For example, if a participant has no writers, it doesn’t strictly need the DDSI built-in endpoints for describing writers, nor the DDSI built-in endpoint for learning of readers of other participants.

5.3 Reliable Communication

Best-effort communication is simply a wrapper around UDP/IP: the packet(s) containing a sample are sent to the addresses at which the readers reside. No state is maintained on the writer. If a packet is lost, the reader will simply drop the sample and continue with the next one.

When reliable communication is used, the writer does maintain a copy of the sample, in case a reader detects it has lost packets and requests a retransmission. These copies are stored in the writer history cache (or WHC) of the DDSI writer. The DDSI writer is required to periodically send Heartbeats to its readers to ensure that all readers will learn of the presence of new samples in the WHC even when packets get lost.

If a reader receives a Heartbeat and detects it did not receive all samples, it requests a retransmission by sending an AckNack message to the writer, in which it simultaneously informs the writer up to what sample it has received everything, and which ones it has not yet received. Whenever the writer indicates it requires a response to a Heartbeat the readers will send an AckNack message even when no samples are missing. In this case, it becomes a pure acknowledgement.

The combination of these behaviours in principle allows the writer to remove old samples from its WHC when it fills up too far, and allows readers to always receive all data. A complication exists in the case of unresponsive readers, readers that do not respond to a Heartbeat at all, or that for some reason fail to receive some samples despite resending it. The specification leaves the way these get treated unspecified.

Note that while this Heartbeat/AckNack mechanism is very straightforward, the specification actually allows suppressing heartbeats, merging of AckNacks and retransmissions, &c. The use of these techniques is required to allow for a performant DDSI implementation, whilst avoiding the need for sending redundant messages.

5.4 DDSI transient-local Behaviour

DCPS specifies four types of data durability ‘volatile’, ‘transient-local’, ‘transient’ and ‘persistent’. Of these, the DDSI specification currently only covers ‘transient-local’, and this is the only form of durable data available when interoperating across vendors.

In DDSI, transient-local data is implemented using the WHC that is normally used for reliable communication. For transient-local data, samples are retained even when all readers have acknowledged them. With the default history setting of KEEP_LAST with history_depth = 1, this means that late-joining readers can still obtain the latest sample for each existing instance.

Naturally, once the DCPS writer is deleted (or disappears for whatever reason), the DDSI writer disappears as well, and with it, its history. For this reason, transient data is generally much to be preferred over transient-local data.

5.5 Discovery of Participants and Endpoints

DDSI participants discover each other by means of the ‘Simple Participant Discovery Protocol’, or ‘SPDP’ for short. This protocol is based on periodically sending a message containing the specifics of the participant to a set of known addresses. By default, this is a standardised multicast address (239.255.0.1; the port number is derived from the domain id) that all DDSI implementations listen to.

Particularly important in the SPDP message are the unicast and multicast addresses at which the participant can be reached. Typically, each participant has a unique unicast address, which in practice means all participants on a node all have a different UDP/IP port number in their unicast address. In a multicast-capable network, it doesn’t matter what the actual address (including port number) is, because all participants will learn them through these SPDP messages.

The protocol does allow for unicast-based discovery, which requires listing the addresses of machines where participants may be located, and ensuring each participant uses one of a small set of port numbers. Because of this, some of the port numbers are derived not only from the domain id, but also from a ‘participant index’, which

is a small non-negative integer, unique to a participant within a node. (The DDSI2 service adds an indirection and uses at most one participant index regardless of how many DCPS participants it handles.)

Once two participants have discovered each other, and both have matched the DDSI built-in endpoints their peer is advertising in the SPDP message, the 'Simple Endpoint Discovery Protocol' or 'SEDP' takes over, exchanging information on the DCPS data readers and data writers in the two participants.

The SEDP data is handled as reliable, transient-local data. Therefore, the SEDP writers send Heartbeats, the SEDP readers detect they have not yet received all samples and send AckNacks requesting retransmissions, the writer responds to these and eventually receives a pure acknowledgement informing it that the reader has now received the complete set.

Note that the discovery process necessarily creates a burst of traffic each time a participant is added to the system: all existing participants respond to the SPDP message, following which all start exchanging SEDP data.

6

Vortex Lite DDSI2E Implementation

DDSI2E is an implementation of the DDSI protocol, version 2.1. The 'E' in the name indicates that it also support a number of proprietary 'Extensions'. DDSI2E adds three major additional extensions:

- Multiple channels: parallel processing of independent data streams, with prioritisation based on the transport priority setting of the data writers, supporting traffic-shaping of outgoing data. See *Multiple Channels*.
- Network partitions: use of special multicast addresses for some partition-topic combinations as well as allowing ignoring data. See *Network Partitioning*.
- Encryption: encrypting all traffic for a certain network partition. See *Encryption*.

These extended features are described more fully at the end of this section. Data encryption is supported using the OpenSSL crypto library.

6.1 Discovery Behaviour

6.1.1 Proxy Participants and Endpoints

DDSI2E is what the DDSI specification calls a 'stateful' implementation. Writers only send data to discovered readers and readers only accept data from discovered writers. (There is one exception: the writer may choose to multicast the data, and anyone listening will be able to receive it, if a reader has already discovered the writer but not vice-versa, it may accept the data even though the connection is not fully established yet.) Consequently, for each remote participant and reader or writer, DDSI2E internally creates a proxy participant, proxy reader or proxy writer.

In the discovery process, writers are matched with proxy readers, and readers are matched with proxy writers, based on the topic and type names and the QoS settings.

Proxies have the same natural hierarchy that 'normal' DDSI entities have: each proxy endpoint is owned by some proxy participant, and once the proxy participant is deleted, all its proxy endpoints are deleted as well. Participants assert their liveliness periodically, and when nothing has been heard from a participant for the lease duration published by that participant in its SPDP message, the lease becomes expired triggering a clean-up.

Under normal circumstances, deleting endpoints simply triggers disposes and unregisters in SEDP protocol, and, similarly, deleting a participant also creates special messages that allow the peers to immediately reclaim resources instead of waiting for the lease to expire.

6.1.2 Lingering Writers

When an application deletes a reliable DCPS data writer, there is no guarantee that all its readers have already acknowledged the correct receipt of all samples. In such a case, DDSI2E lets the writer (and the owning participant if necessary) linger in the system for some time, controlled by the *Internal/WriterLingerDuration* option. The writer is deleted when all samples have been acknowledged by all readers or the linger duration has elapsed, whichever comes first. The writer linger duration setting is currently not applied when DDSI2E is requested to terminate, so in this case data may be lost.

6.1.3 Start-up mode

When starting DDSI2E, discovery takes time, and when data is written immediately after DDSI2E has started, it is likely that the discovery process hasn't completed yet and some remote readers have not yet been discovered. This would cause the writers to throw away samples for lack of interest, even though matching readers already existed at the time of starting. For best-effort writers, this is perhaps surprising but still acceptable; for reliable writers, however, this is counter-intuitive.

Hence the existence of the so-called 'start-up mode', during which all volatile reliable writers are treated as-if they are transient-local writers. Transient-local data is meant to ensure samples are available to late-joining readers, the start-up mode uses this same mechanism to ensure late-discovered readers will also receive the data. This treatment of volatile data as-if it were transient-local happens entirely within DDSI2E and is invisible to the outside world, other than the availability of some samples that would not otherwise be available.

Once DDSI2E has completed its initial discovery, it has built up its view of the network and can locally match new writers against already existing readers, and consequently keeps any new samples published in a writer history cache because these existing readers have not acknowledged them yet. Hence why this mode is tied to the start-up of the DDSI2E, rather than to that of an individual writer.

Unfortunately it is impossible to detect with certainty when the initial discovery process has been completed and therefore the time DDSI2E remains in this start-up mode is controlled by an option: *General/StartupModeDuration*.

While in general, this start-up mode is beneficial, it is not always so. There are two downsides: the first is that during the start-up period, the writer history caches can grow significantly larger than one would normally expect; the second is that it does mean large amounts of historical data may be transferred to readers discovered relatively late in the process.

6.2 Writer Throttling

The DDSI specification heavily relies on the notion of a writer history cache (WHC) within which a sequence number uniquely identifies each sample. Writer throttling is based on the WHC size using a simple bang-bang controller. Once the WHC contains *Internal/Watermarks/WhcHigh* bytes in unacknowledged samples, it stalls the writer until the number of bytes in unacknowledged samples drops below *Internal/Watermarks/WhcLow*. The writer is stalled until either the WHC shrinks to the low water mark or the *max_blocking_time* specified for the writer reliability QoS is in which case a TIMEOUT status is returned from the write operation.

6.3 Unresponsive Readers and Head-of-Stream Blocking

For reliable communications, DDSI2E must retain sent samples in the WHC until they have been acknowledged. Especially in case of a KEEP_ALL history kind, but also in the default case when the WHC is not aggressively dropping old samples of instances *Internal/AggressiveKeepLast1Whc*, a reader that fails to acknowledge the samples timely will cause the WHC to run into resource limits.

One particular case where this can easily occur is if a reader becomes unreachable, for example because a network cable is unplugged. While this will eventually cause a lease to expire, allowing the proxy reader to be removed and the writer to no longer retain data for it, in the meantime the writer can easily run into a WHC limit, causing it to block then time out. The presence of unacknowledged data in a WHC as well as the existence of unresponsive readers will force the publication of Heartbeats, and so unplugging a network cable will typically induce a stream of Heartbeats from some writers.

Another case where this can occur is with a very fast writer, and a reader on a slow host, and with large buffers on both sides: then the time needed by the receiving host to process the backlog can become longer than this responsiveness time out, causing the writer to time out. A writer that times out can delay then write again, allowing the reader to catch up, at which point it once again acknowledges data promptly until a new backlog builds up.

6.4 Networking and Discovery

6.4.1 Networking Interfaces

DDSI2E uses a single network interface, the 'preferred' interface, for transmitting its multicast packets and advertises only the address corresponding to this interface in the DDSI discovery protocol.

To determine the default network interface, DDSI2E ranks the eligible interfaces by quality, and then selects the interface with the highest quality. If multiple interfaces are of the highest quality, it will select the first enumerated one. Eligible interfaces are those that are up and have the right kind of address family (IPv4 or IPv6). Priority is then determined as follows:

- interfaces with a non-link-local address are preferred over those with link-local
- multicast-capable is preferred, or if none is available
- non-multicast capable but neither point-to-point, or if none is available
- to-point, or if none is available
- loopback

If this procedure doesn't select the desired interface automatically, it can be overridden by setting *General/NetworkInterfaceAddress* to either the name of the interface, the IP address of the host on the desired interface, or the network portion of the IP address of the host on the desired interface. An exact match on the address is always preferred and is the only option that allows selecting the desired one when multiple addresses are tied to a single interface.

The default address family is IPv4, setting *General/UseIPv6* will change this to IPv6. Currently, DDSI2E does not mix IPv4 and IPv6 addressing. Consequently, all DDSI participants in the network must use the same addressing mode. When interoperating, this behaviour is the same, i.e., it will look at either IPv4 or IPv6 addresses in the advertised address information in the SPDP and SEDP discovery protocols.

IPv6 link-local addresses are considered undesirable because they need to be published and received via the discovery mechanism, but there is in general no way to determine to which interface a received link-local address is related.

If IPv6 is requested and the preferred interface has a non-link-local address, DDSI2E will operate in a 'global addressing' mode and will only consider discovered non-link-local addresses. In this mode, one can select any set of interfaces for listening to multicasts. Note that this behaviour is essentially identical to that when using IPv4, as IPv4 does not have the formal notion of address scopes that IPv6 has.

If instead only a link-local address is available, DDSI2E will run in a 'link-local addressing' mode. In this mode it will accept any address in a discovery packet, assuming that a link-local address is valid on the preferred interface. To minimise the risk involved in this assumption, it only allows the preferred interface for listening to multicasts.

When a remote participant publishes multiple addresses in its SPDP message (or in SEDP messages, for that matter), it will select a single address to use for communicating with that participant. The address chosen is the first eligible one on the same network as the locally chosen interface, else one that is on a network corresponding to any of the other local interfaces, and finally simply the first one. Eligibility is determined in the same way as for network interfaces.

Multicasting

DDSI2E allows configuring to what extent multicast is to be used:

- whether to use multicast for data communications
- whether to use multicast for participant discovery
- on which interfaces to listen for multicasts

It is advised to allow multicasting to be used. However, if there are restrictions on the use of multicasting, or if the network reliability is dramatically different for multicast than for unicast, it may be attractive to disable

multicast for normal communications. In this case, setting *General/AllowMulticast* to false will force DDSI2E to use unicast communications for everything except the periodic distribution of the participant discovery messages.

If at all possible, it is strongly advised to leave multicast-based participant discovery enabled, because that avoids having to specify a list of nodes to contact, and it furthermore reduces the network load considerably. However, if need be, one can disable the participant discovery from sending multicasts by setting *Internal/SuppressSpdpMulticast* to true.

To disable incoming multicasts, or to control from which interfaces multicasts are to be accepted, one can use the *General/MulticastRecvInterfaceAddresses* setting. This allows listening on no interface, the preferred, all or a specific set of interfaces.

Discovery

Discovery Addresses

The DDSI discovery protocols, SPDP for the domain participants and SEDP for their endpoints, usually operate well without any explicit configuration. Indeed, the SEDP protocol never requires any configuration.

DDSI2E by default uses the domain id as specified in *Domain/Id* but allows overriding it for special configurations using the *Discovery/DomainId* setting. The domain id is the basis for all UDP/IP port number calculations, which can be tweaked when necessary using the configuration settings under *Discovery/Ports*. It is however rarely necessary to change the standardised defaults.

The SPDP protocol periodically sends, for each domain participant, an SPDP sample to a set of addresses, which by default contains just the multicast address, which is standardised for IPv4 (239.255.0.1), but not for IPv6 (it uses ff02::ffff:239.255.0.1). The actual address can be overridden using the *Discovery/SPDPMulticastAddress* setting, which requires a valid multicast address.

In addition (or as an alternative) to the multicast-based discovery, any number of unicast addresses can be configured as addresses to be contacted by specifying peers in the *Discovery/Peers* section. Each time an SPDP message is sent, it is sent to all of these addresses.

Default behaviour of DDSI2E is to include each IP address several times in the set, each time with a different UDP port number (corresponding to another participant index), allowing at least several applications to be present on these hosts. Obviously, configuring a number of peers in this way causes a large burst of packets to be sent each time an SPDP message is sent out, and each local DDSI participant causes a burst of its own. Most of the participant indices will not actually be used, making this rather wasteful behaviour. DDSI2E allows explicitly adding a port number to the IP address, formatted as IP:PORT, to avoid this waste, but this requires manually calculating the port number. In practice it also requires fixing the participant index using *Discovery/ParticipantIndex* to ensure that the configured port number indeed corresponds to the remote DDSI2E (or other DDSI implementation).

Asymmetric Discovery

On reception of an SPDP packet, DDSI2E adds the addresses advertised in that SPDP packet to this set, allowing asymmetrical discovery. In an extreme example, if SPDP multicasting is disabled entirely, host A has the address of host B in its peer list and host B has an empty peer list, then B will eventually discover A because of an SPDP message sent by A, at which point it adds A's address to its own set and starts sending its own SPDP message to A, allowing A to discover B. This takes a bit longer than normal multicast based discovery, though.

Timing of SPDP Packets

The interval with which the SPDP packets are transmitted is configurable as well, using the *Discovery/SPDPInterval* setting. A longer interval reduces the network load, but also increases the time discovery takes, especially in the face of temporary network disconnections.

Endpoint Discovery

Although the SEDP protocol never requires any configuration, the network partitioning of DDSI2E does interact with it: so-called 'ignored partitions' can be used to instruct DDSI2E to completely ignore certain DCPS topic and partition combinations, which will prevent DDSI2E from forwarding data for these topic/partition combinations to and from the network.

6.4.2 Combining Multiple Participants

The *Internal/SquashParticipants* configuration option can be used to simulate the existence of only one participant, which owns all endpoints on that process. This reduces the background messages because far fewer liveliness assertions need to be sent. Clearly, the liveliness monitoring features that are related to domain participants will be affected if multiple DCPS domain participants are combined into a single DDSI domain participant. When interoperability with another vendor is not needed, enabling the SquashParticipants option is often a good choice.

6.4.3 Controlling Port Numbers

The port numbers used by DDSI2E are determined as follows, where the first two items are given by the DDSI specification and the third is unique to DDSI2E as a way of serving multiple participants by a single DDSI instance:

- Two 'well-known' multicast ports: B and B+1
- Two unicast ports at which only this instance of DDSI2E is listening: B+PG*PI+10 and B+PG*PI+11
- One unicast port per domain participant it serves, chosen by the kernel from the anonymous ports, i.e., >= 32768

Where:

- B is *Discovery/Ports/Base* (7400) + *Discovery/Ports/DomainGain* (250) * *Domain/Id*
- PG is *Discovery/Ports/ParticipantGain*
- PI is *Discovery/ParticipantIndex*

The default values, taken from the DDSI specification, are in parentheses. There are actually even more parameters, here simply turned into constants as there is absolutely no point in ever changing these values—but they are configurable and the interested reader is referred to the DDSI 2.1 specification, section 9.6.1.

PI is the most interesting, as it relates to having multiple instances of DDSI2E in the same domain on a single node. Its configured value is either 'auto', 'none' or a non-negative integer. This setting matters:

- When it is 'auto' (which is the default), DDSI2E probes UDP port numbers on start-up, starting with PI = 0, incrementing it by one each time until it finds a pair of available port numbers, or it hits the limit. The maximum PI it will ever choose is currently still hard-coded at 9 as a way of limiting the cost of unicast discovery.
- When it is 'none' it simply ignores the 'participant index' altogether and asks the kernel to pick two random ports (>= 32768). This eliminates the limit on the number of deployments on a single machine and works just fine with multicast discovery while complying with all other parts of the specification for interoperability. However, it is incompatible with unicast discovery.
- When it is a non-negative integer, it is simply the value of PI in the above calculations. If multiple instances of DDSI2E on a single machine are needed, they will need unique values for PI, and so for standalone deployments this particular alternative is hardly useful.

Clearly, to fully control port numbers, setting *Discovery/ParticipantIndex* (= PI) to a hard-coded value is the only possibility. By fixing PI, the port numbers needed for unicast discovery are fixed as well. This allows listing peers as IP:PORT pairs, significantly reducing traffic, as explained in the preceding subsection.

The other non-fixed ports that are used are the per-domain participant ports, the third item in the list. These are used only because some DDSI implementations exist that assume each domain participant advertises a unique port number as part of the discovery protocol, and hence that there is never any need for including an explicit destination

participant id when intending to address a single domain participant by using its unicast locator. DDSI2E never makes this assumption, instead opting to send a few bytes extra to ensure the contents of a message are all that is needed. With other implementations, you will need to check.

If all DDSI implementations in the network include full addressing information in the messages, then the per-domain participant ports serve no purpose at all. The default false setting of *Compatibility/ManySocketsMode* disables the creation of these ports. This setting has a few other side benefits, as there will generally be more participants using the same unicast locator, improving the chances for requiring a single unicast, even when addressing multiple participants in a node. The obvious case where this is beneficial is when one host has not received a multicast.

6.5 Multiple Channels

DDSI2E allows defining channels, which are independent data paths within the DDSI service. Vortex Lite chooses a channel based by matching the transport priority QoS setting of the data writer with the threshold specified for the various channels. Because each channel has a set of dedicated threads to perform the processing and the thread priorities can all be configured, it is straightforward to guarantee that samples from high-priority data writers will get precedence over those from low-priority data throughout the service stack.

A second aspect to the use of channels is to avoid the head-of-line blocking mentioned previously. Unresponsive readers & head-of-stream blocking is per channel, guaranteeing that a high-priority channel will not be disrupted by an unresponsive reader of low-priority data.

The channel-specific threads perform essentially all processing (serialisation, writer history cache management, deserialisation, delivery to DCPS data readers, &c.), but there still is one shared thread involved. This is the receive thread ('recv') that demultiplexes incoming packets and implements the protocol state machine. The receive thread only performs minimal work on each incoming packet, and never has to wait for the processing of user data.

When configuring multiple channels, it is recommended to set the CPU priority of the receive thread to at least that of the threads of the highest priority channel, to ensure the receive thread will be scheduled in promptly. If no channels are defined explicitly, a single, default channel is used.

6.5.1 Transmit Side

For each discovered local data writer, DDSI2E determines the channel to use. This is the channel with the lowest threshold priority of all channels that have a threshold priority that is higher than the writer's transport priority. If there is no such channel, i.e., the writer has a transport priority higher than the highest channel threshold, the channel with the highest threshold is used.

Each channel has its own network queue into which the OpenSplice kernel writes samples to be transmitted and that DDSI2E reads. The size of this queue can be set for each channel independently by using *Channels/Channel/QueueSize*, with the default taken from the global *Sizing/NetworkQueueSize*.

Bandwidth limiting and traffic shaping are configured per channel as well. The following parameters are configurable:

- bandwidth limit
- auxiliary bandwidth limit
- IP QoS settings

The traffic shaping is based on a leaky bucket algorithm: transmit credits are added at a constant rate, the total transmit credit is capped, and each outgoing packet reduces the available transmit credit. Outgoing packets must wait until enough transmit credits are available.

Each channel has two separate credits: data and auxiliary. The data credit is used strictly for transmitting fresh data (i.e., directly corresponding to writes, disposes, &c.) and control messages directly caused by transmitting that data. This credit is configured using the *Channels/Channel/DataBandwidthLimit* setting. By default, a channel is treated as if it has infinite data credit, disabling traffic shaping.

The auxiliary credit is used for everything else: asynchronous control data & retransmissions, and is configured using the *Channels/Channel/AuxiliaryBandwidthLimit* configuration setting. When an auxiliary bandwidth limit has been set explicitly, or when one explicitly sets, e.g., a thread priority for a thread named 'tev.channel-name', an independent event thread handles the generation of auxiliary data for that channel. But if neither is given, the global event thread instead handles all auxiliary data for the channel.

The global event thread has an auxiliary credit of its own, configured using *Internal/AuxiliaryBandwidthLimit*. This credit applies to all discovery related traffic, as well as to all auxiliary data generated by channels without their own event thread. Generally, it is best to simply specify both the data and the auxiliary bandwidth for each channel separately, and set *Internal/AuxiliaryBandwidthLimit* to limit the network bandwidth the discovery & liveliness protocols can consume.

6.5.2 Receive Side

On the receive side, the single receive thread accepts incoming data and runs the protocol state machine. Data ready for delivery to the local data readers is queued on the delivery queue for the channel that best matches the proxy writer that wrote the data, according to the same criterion used for selecting the outgoing channel for the data writer.

The delivery queue is emptied by the delivery thread, 'dq.channel-name', which deserialises the data and updates the data readers. Because each channel has its own delivery thread with its own scheduling priority, once the data leaves the receive thread and is enqueued for the delivery thread, higher priority data once again takes precedence over lower priority data.

6.5.3 Discovery Traffic

DDSI discovery data is always transmitted by the global timed-event thread ('tev'), and always processed by the special delivery thread for DDSI built-in data ('dq.builtin'). By explicitly creating a timed-event thread, one effectively separates application data from all discovery data. One way of creating such a thread is by setting properties for it, another is by setting a bandwidth limit on the auxiliary data of the channel.

6.5.4 Interoperability

DDSI2E channels are fully compliant with the wire protocol. One can mix & match DDSI2E with different sets of channels and with other vendors' implementation.

6.6 Network Partitioning

Network partitions introduce alternative multicast addresses for data. In the DDSI discovery protocol, a reader can override the default address at which it is reachable, and this feature of the discovery protocol is used to advertise alternative multicast addresses. The DDSI writers in the network will (also) multicast to such an alternative multicast address when multicasting samples or control data.

The mapping of a DCPS data reader to a network partition is indirect: DDSI2E first matches the DCPS data reader partitions and topic against a table of 'partition mappings', partition/topic combinations to obtain the name of a network partition, then looks up the network partition. This makes it easier to map many different partition/topic combinations to the same multicast address without having to specify the actual multicast address many times over. If no match is found, DDSI2E automatically defaults to standardised DDSI multicast address.

6.6.1 Matching Rules

Matching of a DCPS partition/topic combination proceeds in the order in which the partition mappings are specified in the configuration. The first matching mapping is the one that will be used. The '*' and '?' wildcards are available for the DCPS partition/topic combination in the partition mapping.

DDSI2E can be instructed to ignore all DCPS data readers and writers for certain DCPS partition/topic combinations through the use of *Partitioning/IgnoredPartitions*. The ignored partitions use the same matching rules as normal mappings, and take precedence over the normal mappings.

6.6.2 Multiple Matching Mappings

A single DCPS data reader can be associated with a set of partitions, and each partition/topic combination can potentially map to a different network partitions. In this case, DDSI2E will use the first matching network partition. This does not affect what data the reader will receive; it only affects the addressing on the network.

6.6.3 Interoperability

DDSI2E network partitions are fully compliant with the wire protocol. One can mix and match DDSI2E with different sets of network partitions and with other vendors' implementation.

6.7 Encryption

DDSI2E encryption support allows defining 'security profiles', named combinations of (symmetrical block) ciphers and keys. These can be associated with subsets of the DCPS data writers via the network partitions: data from a DCPS data writer matching a particular network partition will be encrypted if that network partition has an associated security profile.

The encrypted data will be tagged with a unique identifier for the network partition, in cleartext. The receiving nodes use this identifier to lookup the network partition and the associated encryption key and cipher. Clearly, this requires that the definition of the encrypted network partitions must be identical on the transmitting and the receiving sides. If the network partition cannot be found, or if the associated key or cipher differs, the receiver will ignore the encrypted data. It is therefore not necessary to share keys with nodes that have no need for the encrypted data. The encryption is performed per-packet; there is no chaining from one packet to the next.

6.7.1 Adding Encryption Support

Encryption support is included in the optional *dds_ssl* library. This is required to install encryption components. This must be linked with applications that wish to use encryption and installed by calling the ssl plugin function:

```
dds_ssl_plugin ();
```

6.8 Enabling Encryption

```
<DDSI2E>
  <Partitioning>
    <NetworkPartitions>
      <NetworkPartition Name="part_1" Address= "224.0.0.1"  SecurityProfile="Security1"/>
    </NetworkPartitions>
    <PartitionMappings>
      <PartitionMapping NetworkPartition="part_1" DCPSPartitionTopic="TestPartition_1.TestTopic_1
    </PartitionMappings>
  </Partitioning>
  <Security>
    <SecurityProfile Cipher="aes128" CipherKey="ABCDEFABCDEFABCDABCDEFABCDEFABCD" Name="Security1
  </Security>
 </DDSI2E>
```

6.8.1 Interoperability

Encryption is not yet a standardised part of DDSI, but the standard does allow vendor-specific extensions. DDSI2E encryption relies on a vendor-specific extension to marshal encrypted data into valid DDSI messages, but they cannot be interpreted by implementations that do not recognise this particular extension.

6.9 Instance Management

For a sequence of inputs, there is a deterministic mapping of sequence of operations to the state read by the application. However, if the reader history contains sequences of register/unregister, then the messages are merged into one exisiting/following the actual sample.

Here is an example:

Publisher:

```
dds_write (writer, inst1-sample1);
dds_write (writer, inst1-sample2);
dds_unregister (writer, inst1);
dds_write (writer, inst1-sample3);
```

Subscriber:

- if a third sample arrives before reading the sample information corresponding to unregister (invalid data), then the instance state is set to ALIVE and no_writers_generation_count is set to 1 to indicate that there were not any writers preceding this sample.
- if unregister is read, the instance state is set to NOT_ALIVE_NO_WRITERS and the instance is freed from memory. When a third sample arrives, the instance is treated as NEW and the sample state is set to ALIVE with all the counters set to 0.

6.10 Entity Deletion

Applications should take care of releasing all associated resources before deleting an entity. If an entity is deleted without proper cleanup, Vortex Lite will delete regardless and may assert in debug mode (no error code is returned). This results in undefined behaviour that might lead to accessing resources which are already deleted. For instance, if a reader has some associated conditions and waitsets attached, application should ensure that all the conditions are detached and deleted from waitsets before calling delete_datareader or delete_contained_entities.

6.11 Changeable QoS

QoS for an entity can be set only at the time of creation. Dynamic update of QoS is not supported. To change QoS for a particular entity, it should be deleted and recreated with a new set of QoS values.

Here is the code snippet for changing Ownership strength on a writer:

```
dds_qos_get (writer, qos);
dds_entity_delete (writer);
dds_qset_ownership_strength (qos, <new value>);
dds_writer_create (ppant, writer, topic, qos, NULL);
```

7

Using Vortex Lite with TCP

In addition to UDP, Vortex Lite supports the TCP protocol as a transport layer for its DDSI stack. Using Vortex Lite with TCP requires the setting of a number of configuration properties. This chapter describes the main configuration steps that must be applied to use TCP. Other configuration options available for fine-tuning of the TCP configuration are described in */docs/configuration.html*.

7.1 Enabling the TCP protocol

By default, Vortex Lite uses the UDP protocol. Switching to TCP simply requires enabling it in the XML configuration:

```
<DDSI2E>
  <TCP>
    <Enable>true</Enable>
  </TCP>
</DDSI2E>
```

7.2 Identifying Peers

Each DDS participant should know the other participants' (peers) endpoints, in order to connect and discover them. A peer is identified by an IP address or a host name, with a port number. To identify remote peers, they must be added to the configuration XML:

```
<DDSI2E>
  <Discovery>
    <Peers>
      <Peer address="10.1.0.40:7405"/>
      <Peer address="10.1.0.41:3542"/>
    </Peers>
  </Discovery>
</DDSI2E>
```

7.3 Setting the participant public locator

In order to be reached by a remote participant *via* TCP, each participant can indicate how it can be connected *via* TCP (tcp listener) This is done by configuring the Port on which the transport listens for new connections:

```
<DDSI2E>
  <TCP>
    <Enable>true</Enable>
    <Port>7400</Port>
  </TCP>
</DDSI2E>
```

This port is placed in the endpoints published by the participant in DDSI discovery, and is used by peers to establish connections with the participant. If the port is not set, then peers cannot connect directly to the participant, however if peers are configured then the participant will be able to establish connections with those peers, who will then be able to communicate with the participant via these connections. No TCP communication is possible if neither the port or peers are configured. If the Port is configured to be zero then a dynamic port is created, not configuring the port or setting it to -1 disables its usage.

7.4 Using a Discovery Service (Optional)

For a stand alone system consisting of TCP enabled participants, each participant must be able to connect to each peer with which it needs to communicate. This can be achieved by configuring each participant with the set of peers to which it needs to interact; however this is a static configuration that scales badly. To give greater scalability and add dynamism to the system, a discovery service can be deployed. Here each participant simply needs to be configured with the peer address of the discovery service, which then brokers communication endpoints via the DDSI discovery protocol. The discovery service is part of the Vortex Cloud product suite, see *Using Vortex Lite with Vortex Cloud*.

8

Using Vortex Lite with SSL

Vortex Lite supports secure communications inside a DDS domain by using the SSL protocol. SSL is supported using OpenSSL. For details on how to manage SSL keys and certificates the O'Reilly book "Network Security with OpenSSL" is recommended. Vortex Lite supports version 3 of SSL also known as TLS.

8.1 Adding SSL Support

SSL support is included in the optional *dds_ssl* library. This must be linked with applications that wish to use ssl and installed by calling the ssl plugin function:

```
dds_ssl_plugin ();
```

8.2 Enabling SSL

SSL is layered over the TCP transport. Both must be enabled in the configuration XML:

```
<DDSI2E>
  <TCP>
    <Enable>true</Enable>
  </TCP>
  <SSL>
    <Enable>true</Enable>
  </SSL>
</DDSI2E>
```

8.3 Providing a Key Store

SSL is supported using the OpenSSL libraries. Keys and certificates are managed in a key store *.pem* file. The file must be specified as part of the SSL XML configuration as well as the passphrase used to encrypt the store:

```
<DDSI2E>
  <SSL>
    <Enable>true</Enable>
    <KeystoreFile>/home/dds/ssl/server.pem</KeystoreFile>
    <KeyPassphrase>secure</KeyPassphrase>
  </SSL>
</DDSI2E>
```

8.4 Certificate Management

By default all SSL certificates (held in a key store) are validated and self signed certificates are rejected (ones not signed by a recognised authority). However both verification and acceptance of self signed certificates can be

configured:

```
<DDSI2E>
  <SSL>
    <CertificateVerification>true</CertificateVerification>
    <SelfSignedCertificates>true</SelfSignedCertificates>
  </SSL>
</DDSI2E>
```

8.5 Creating Keys and Certificates

All OpenSSL keys and certificates are managed with the *openssl* utility. It is recommended that OpenSSL reference material is consulted to fully understand the mechanisms and options available for creating keys and certificates when setting up a secure SSL configuration.

Examples

- Creating a private key and a CA certificate for self signing:

```
openssl genrsa -des3 -out privkey.pem 2048
openssl req -new -x509 -key privkey.pem -out cacert.pem -days 1095
```

- Creating a SSL server key store containing a server key and self signed certificate:

```
openssl req -newkey rsa:1024 -sha1 -keyout serverkey.pem -out serverreq.pem -days 90
openssl x509 -req -in serverreq.pem -sha1 -CA cacert.pem -CAkey privkey.pem -CAcreateserial -out
cat servercert.pem serverkey.pem cacert.pem > server.pem
```

- Creating a SSL client key store containing a client key and self signed certificate:

```
openssl req -newkey rsa:1024 -sha1 -keyout clientkey.pem -out clientreq.pem -days 90
openssl x509 -req -in clientreq.pem -sha1 -CA cacert.pem -CAkey privkey.pem -CAcreateserial -out
cat clientcert.pem clientkey.pem cacert.pem > client.pem
```

- Verifying the generated client and server certificates:

```
openssl verify -CAfile cacert.pem servercert.pem
openssl verify -CAfile cacert.pem clientcert.pem
```

8.6 Interoperating with Vortex Café

When SSL is used to communicate between Vortex Lite and Votex Café, SSL certificates must be managed either in Java (using the keytool utility) or in OpenSSL (using the openssl utility). To share certificates between both SSL implementations the certificates must be copied from one format to another.

To convert from OpenSSL format (pem files) to Java key store format, the Java keytool utility can be used. To convert Java key store format files to OpenSSL pem files, a number of open source utilities can be used, such as Portecle (available from http://sourceforge.net/projects/portecle/). For exact details on how to transform keys and certificates please refer to the specific tool

9

Using Vortex Lite with Vortex Cloud

If Vortex Lite is deployed in a LAN or a private cloud (multicast-enabled cloud) and Vortex Cloud is configured to discover user applications using UDP-multicast in the LAN/private cloud, then Vortex Lite does not need to be configured with any particular configuration property. Vortex Cloud will discover and communicate with Vortex Lite using the standardized multicast IP address and UDP ports. The only constraint is that both Vortex Lite and Vortex Cloud are configured to participate in the same DDS domain (*DomainID*).

If Vortex Lite is deployed in a WAN (without multicast support), then it needs to be configured to use TCP transport. It also needs to be configured with the peers of one or several discovery services of the Vortex Cloud system. If multiple discovery services are available (for fault-tolerance purposes), Vortex Lite may be configured with several discovery service peers in a peers group. Vortex Lite will try to connect to the first discovery service of the group, and then with the others only if the first cannot be connected.

Example with a single discovery service locator:

```
<DDSI2Service>
  <General>
    .....
  </General>
  <TCP>
    <Enable>True</Enable>
  </TCP>
  <Discovery>
    <Peers>
      <Peer address="1.1.1.1:7400"/>
    </Peers>
  </Discovery>
```

Example with two discovery service locators:

```
<Discovery>
  <Peers>
    <Peer address="1.1.1.1:7400"/>
    <Peer address="2.2.2.2:7400"/>
  </Peers>
</Discovery>
```

ℹ Unlike when deployed on a multicast LAN (where the domainID determines the multicast-address used for discovery), the addressed DS of Vortex Cloud will discover (and subsequently match) data from any DDS-domain utilized by Vortex Lite. Utilizing multiple Vortex Cloud discovery services in the WAN (i.e. one per domain) allows for domain-specific discovery and subsequent routing if required.

10

Using Durability with Vortex Lite

In its standalone configuration Vortex Lite does not support durable (persistent) Topics, these are Topics with a durability kind of TRANSIENT or PERSISTENT. However Vortex Lite can be deployed in conjunction with Vortex OpenSplice to take advantage of its durability service to support these topic types via *client side durability*.

For OpenSplice to act as a durability server for Vortex Lite it must be configured to run an instance of the durability service in the same domain as Lite and use a DDSI networking service (see the Vortex OpenSplice documentation for details).

10.1 Adding Durability Support

Durability support is included in the optional *dds_dur* library. This must be linked with applications that wish to use durability and installed by calling the durability plugin function:

```
dds_durability_plugin ();
```

10.2 Durability Configuration

Client side durability can be configured via the configuration tool. The XML configuration is within a *Durability* section:

```
<Lite>
  <Durability>
    <Encoding>cdr_client</Encoding>
  </Durability>
</Lite>
```

Also server side durability support must be configured in the OpenSplice XML configuration:

```
<DurabilityService name="durability">
  <ClientDurability enabled="true">
  </ClientDurability>
</DurabilityService>
```

10.3 Durable Publishers

A publisher application for durable data should create TRANSIENT or PERSISTENT topic instances and then simply write samples as normal. If samples are to be durable after the writer is deleted (or the application terminated), then the Writer Data Lifecycle QoS should be set to false, to ensure that the data instance is not automatically disposed.

```
dds_init (argc, argv);
...
dds_entity_t topic;
```

```
dds_qos_t * qos = dds_qos_create ();
dds_qset_durability (qos, DDS_DURABILITY_PERSISTENT);
dds_topic_create (participant, &topic, &Msg_desc, "Msg", qos, NULL);
dds_qos_delete (qos);
...
dds_entity_t writer;
qos = dds_qos_create ();
dds_qset_reliability (qos, DDS_RELIABILITY_RELIABLE, DDS_SECS (1));
dds_qset_writer_data_lifecycle (qos, false);
dds_writer_create (participant, &writer, topic, qos, NULL);
...
dds_write (writer, &sample);
```

10.4 Durable Subscribers

A subscriber application for durable data should install the durability plugin, create TRANSIENT or PERSISTENT topic instances, then for each durable reader call the *dds_reader_wait_for_historical_data* function to wait for any historic data to be delivered. Historic data samples can then be read or taken as normal. A time out can be provided to *dds_reader_wait_for_historical_data* so that the client does not block if no durable data is available.

```
dds_durability_plugin ();
dds_init (argc, argv);
...
dds_entity_t topic;
dds_qos_t * qos = dds_qos_create ();
dds_qset_durability (qos, DDS_DURABILITY_PERSISTENT);
dds_topic_create (participant, &topic, &Msg_desc, "Msg", qos, NULL);
dds_qos_delete (qos);
...
dds_entity_t reader;
qos = dds_qos_create ();
dds_qset_durability (qos, DDS_DURABILITY_PERSISTENT);
dds_reader_create (participant, &reader, topic, qos, NULL);
...
dds_reader_wait_for_historical_data (reader, DDS_SECS (10));
dds_read (reader, samples, MAX_SAMPLES, info, states);
```

10.5 Durability Service QoS

Durable publishers and subscribers need not set the Durability Service QoS and just work with default values. However if a publisher or subscriber does explicitly set the Durability Service QoS, then this must be set the same for all durable publishers and subsribers for that topic.

11

Using Vortex Lite with VxWorks

Vortex Lite supports VxWorks as shared/static RTP libraries or kernel AMP/SMP libraries. Typically it is compiled for a specific target configuration or BSP. Please contact ADLINK for specific platform availability. For evaluation purposes the generic Pentium 4, PowerPC 32 and VxWorks simulator targets are supported "out of the box". Also note that by default only the GNU compiler toolchain is supported, both Diab and the Intel toolchains can be supported on request.

11.1 VxWorks7 Support

VxWorks 7 is different from previous versions of VxWorks in that RTP builds are no longer for generic CPU targets, but are built against a specific VSB (which may either target a specific BSP, or a specific CPU.) We include the vsb.config file in the etc directory of the distribution so that you can generate a matching VSB (with the same versions of all layers etc.) to build your VIP project, and program code against and this is required for stable operation on vxworks.

This is explained in the VxWorks 7 Programmer's Guide, See Caution section under "RTP Applications for UP and SMP Configurations of VxWorks" currently on page 24.

11.2 Hosts and Targets

VxWorks is a target platform in that applications are cross compiled and built for a VxWorks target on a host. For VxWorks supported host types are Linux and Windows. Vortex Lite currently supports a variety of PPC, ARM and X86 targets. The *LITE_TARGET* environment variable is used identify the target build system for lite and *LITE_HOST* for the host. For VxWorks the following configurations are currently supported:

LITE_HOST	LITE_TARGET	Build Configuration
linux_gcc_x86	vxworks_gnu_x86_linux	Linux x86 host build for VxWorks x86 target
linux_gcc_x86	vxworks_gnu_arm_linux	Linux x86 host build for VxWorks ARM target
linux_gcc_x86	vxworks_gnu_ppc_linux	Linux x86 host build for VxWorks PPC target
win32_cl_x86	vxworks_gnu_x86_win32	Windows 7 x86 host build for VxWorks x86 target
win32_cl_x86	vxworks_gnu_arm_win32	Windows 7 x86 host build for VxWorks ARM target
win32_cl_x86	vxworks_gnu_ppc_win32	Windows 7 x86 host build for VxWorks PPC target

11.3 Unsupported Features

Currently not all Lite features are supported on VxWorks:

- TCP over SSL
- The ISO C++ binding (supported on some VxWorks 7 builds.)

11.4 VxWorks Kernel Requirements

Lite has been tested with VxWorks development kernels. The core requirements are for POSIX, RTP and network support. The following kernel components are required:

- PROFILE_DEVELOPMENT
- BUNDLE_POSIX
- INCLUDE_POSIX_PTHREAD_SCHEDULER
- BUNDLE_RTP_DEPLOY
- INCLUDE_CPLUS (optional for c++ support)
- INCLUDE_CPLUS_LANG (optional for c++ support)
- INCLUDE_CPLUS_IOSTREAMS (optional for c++ support)
- INCLUDE_IPV6 (optional for ipv6 support)

Lite will run with default configuration settings, but to support using an XML configuration file, some sort of file system supports (ROMFS/NFS etc) must also be provided by the kernel.

11.5 Building with Make

By default support is provided for building VxWorks examples using GNU make. This is the native make system on Linux systems and can be provided by CygWin on windows systems. Once the user environment has been configured (see: *Setting the User Environment*) the examples can be re-compiled simply by typing *make*, or *make debug* (for debug executables) in the appropriate directory.

For kernel space builds of the examples, the tool lite_vx_wrapper is automatically used by the makefiles to generate (source code of) a wrapper function with the name of the example which takes the arguments as one string, and then passes them to the main function in the example, using C argc, argv style parameters. The main function in the example is renamed using objcopy to <examplename>_main.

Support for building the config xml file into the examples binaries is also provided. To do this set the VL_BUILDIN_CONFIG variable to "1" and set the LITE_URI before building the example(s) then the config will be builtin. This can then be referenced by the uri litecfg://<filename of original xml> e.g.

```
export LITE_URI=file://$LITE_HOME/etc/lite.xml
make VL_BUILDIN_CONFIG=1
```

Then on the target you can use:

```
putenv ("LITE_URI=litecfg://lite.xml")
```

The tool liteconf2c can be used to generate a C source file which can be built, and linked into your executable to achieve the above with your own makefiles.

It options are as follows:

```
liteconf2c [-] | [ [-x] [-u <URI>]... [-e <env=var> ]... [-o <file>]
```

The -x option exludes XML configuration, just sets environment variables. The -u option(s) can be used to include one or more URI file into the generated code. The -e option(s) allow environment variable settings to be built in too. The -o option allows the output file name to be changed from the default "lite_config.c"

```
liteconf2c -u file://$LITE_HOME/etc/lite.xml -e LITE_URI=litecfg://lite.xml
```

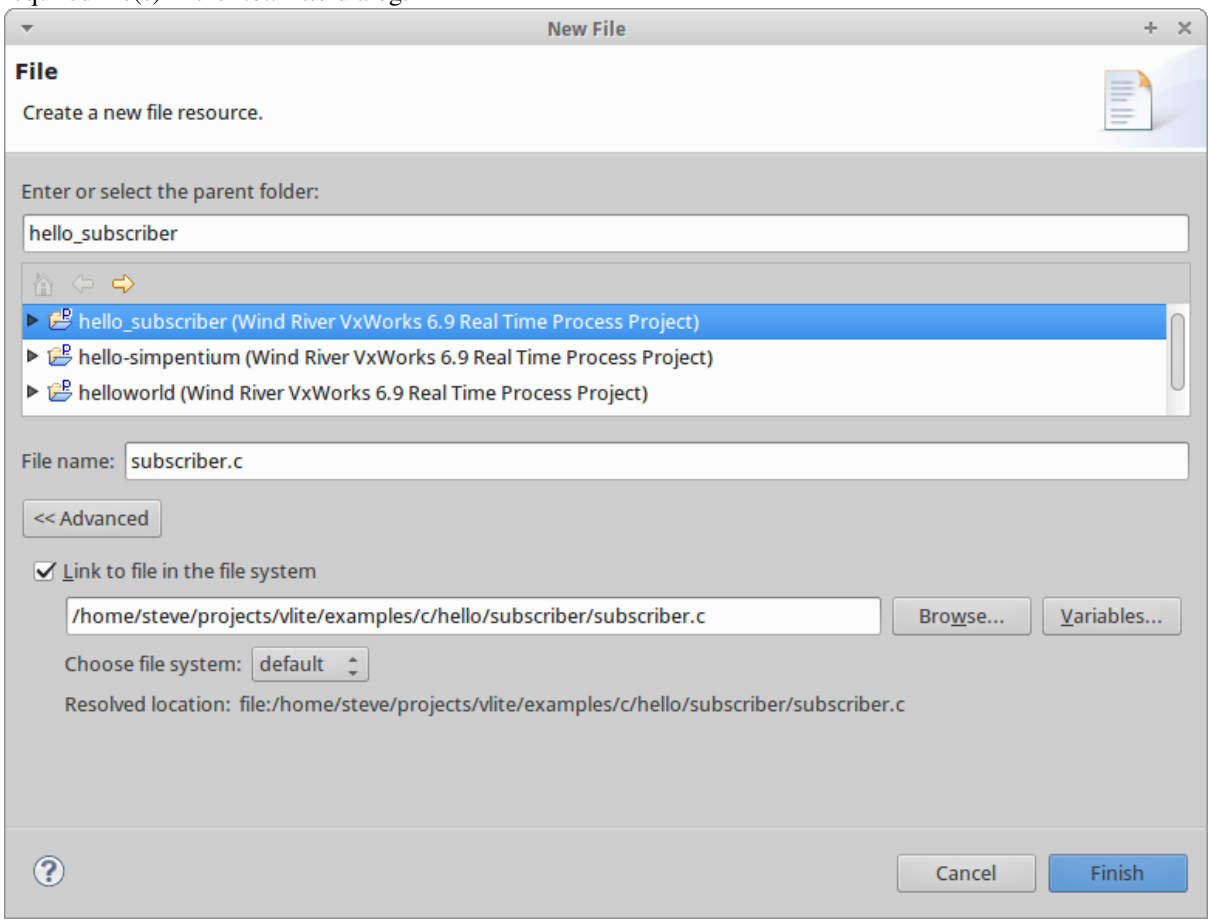
11.6 Building with WorkBench

Vortex Lite examples for RTP can be built as new WorkBench projects or Lite can be integrated into existing application projects. The steps for both are essentially the same:

- Create a new or use an existing WorkBench project
- Add Lite source files to the project
- Add Lite include directories to the project
- Add Lite libraries to the project
- Add Lite definitions to project compile flags

11.6.1 Adding Project Source Files

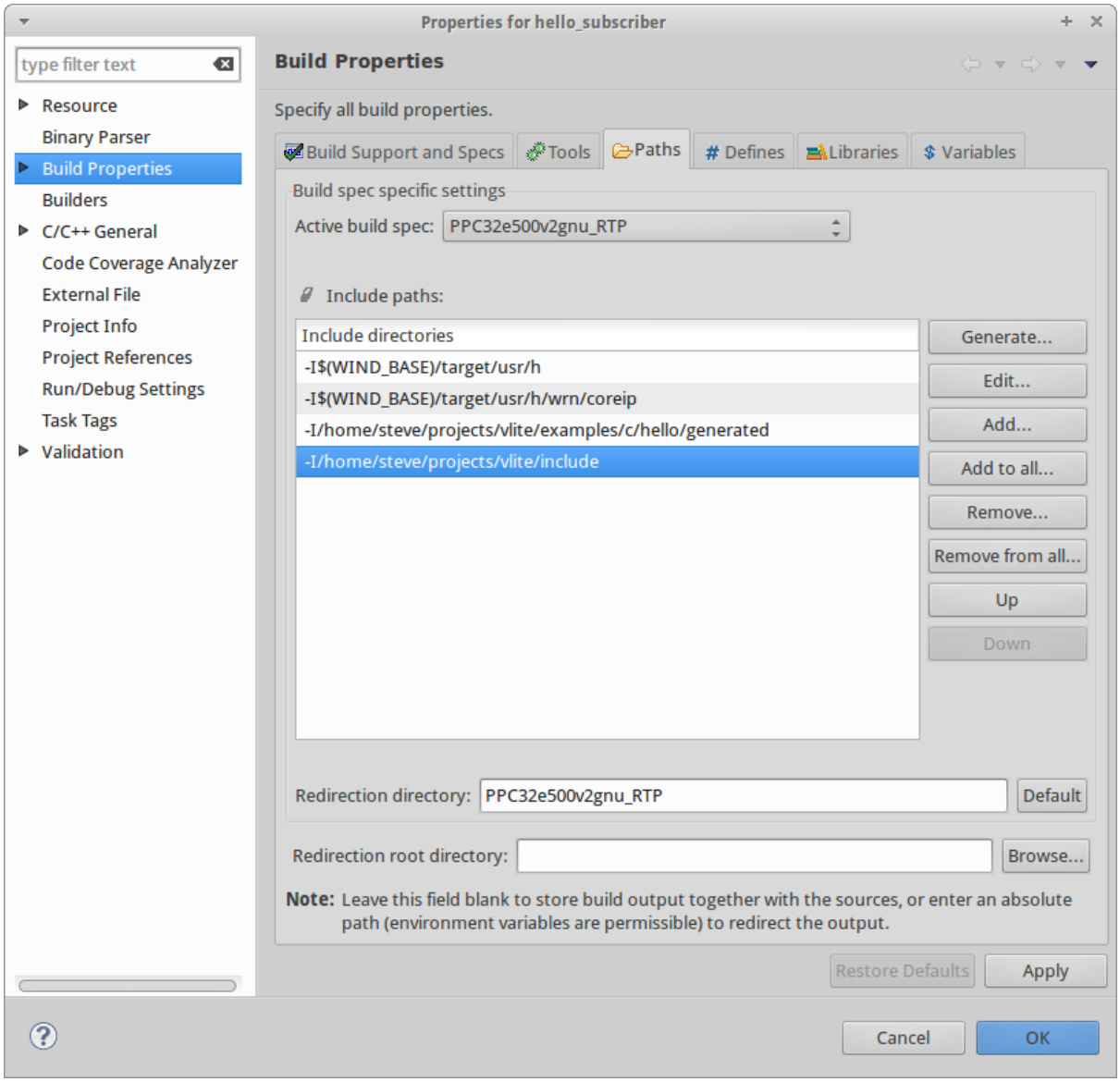
Lite source files can be added to a project by right clicking the project, selecting *New->File* and selecting the required file(s) in the *New File* dialog:



11.6.2 Adding Project Include Directories

The *$LITE_HOME/include* directory should always be included.

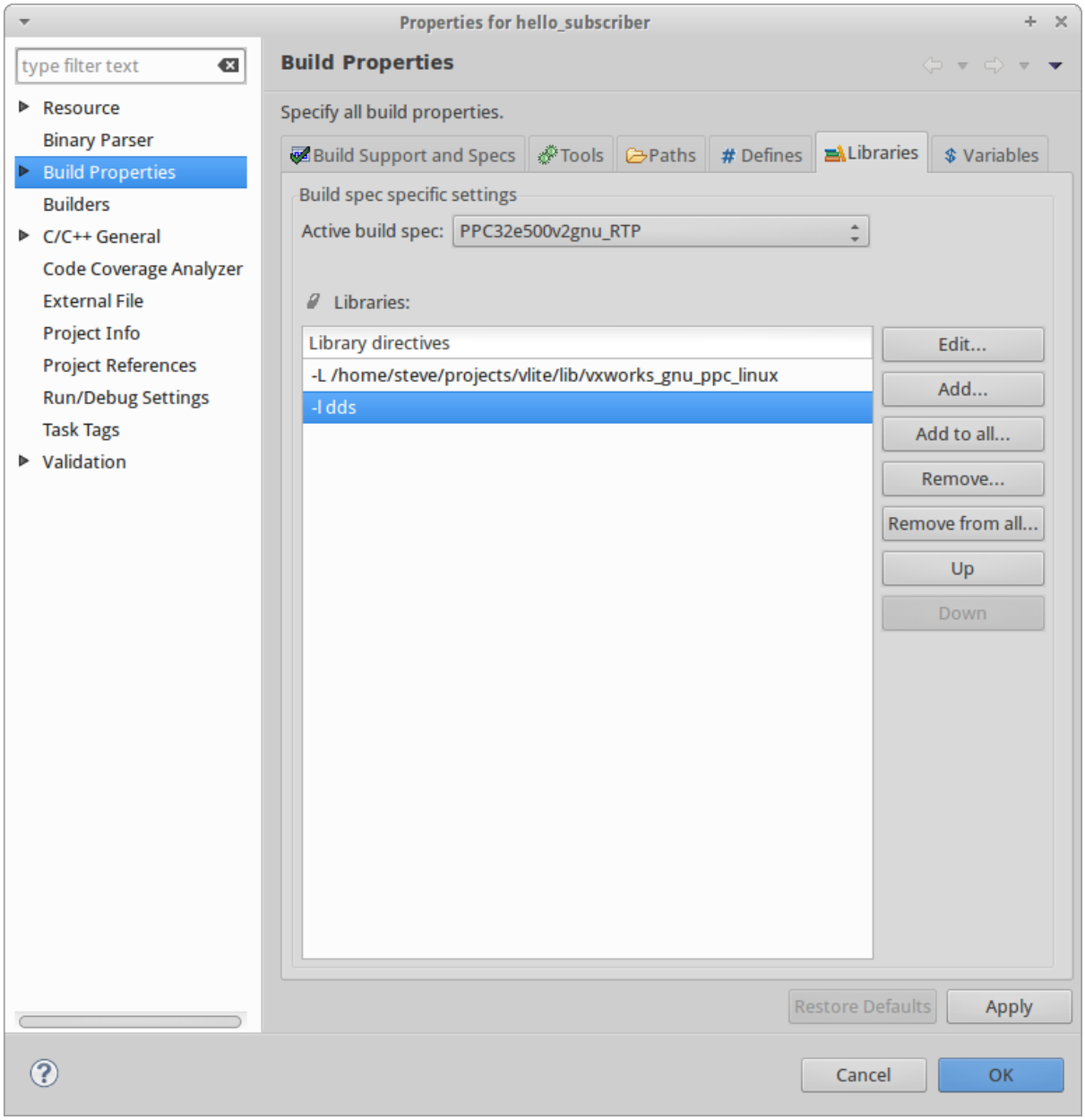
Lite include directories can be added to a project by right clicking the project, selecting *Properties*, selecting the *Paths* tab of the *Build Properties* dialog and selecting with the *Add...* button:



11.6.3 Adding Project Libraries

The *$LITE_HOME/lib/$LITE_TARGET* directory should always be set as a library include directory and the *dds* library always set as a link libraries (additional libraries may also be required). For a debug project the *_g* variants of the Lite libraries should be used.

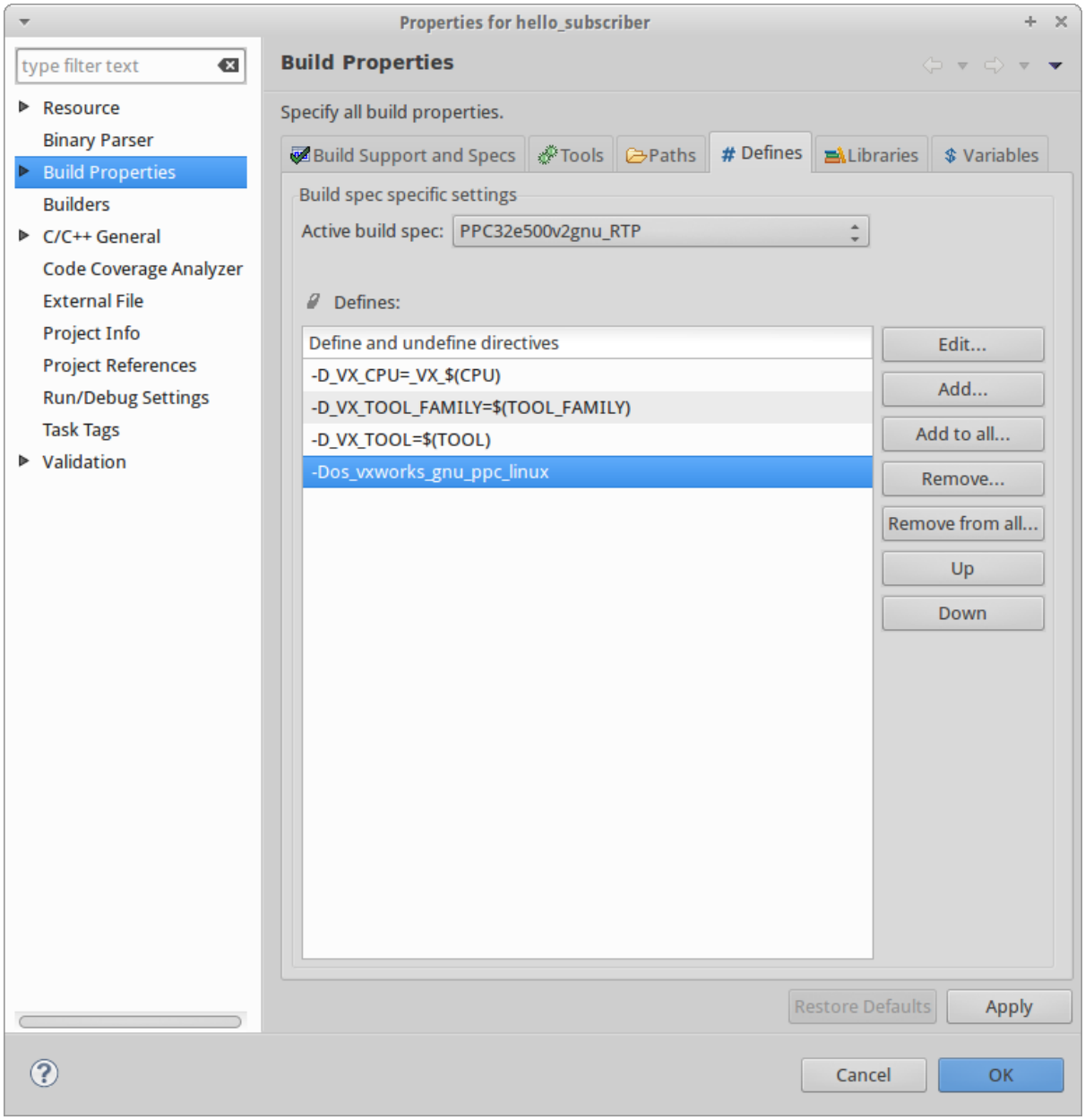
Lite libraries can be added to a project by right clicking the project, selecting *Properties*, selecting the *Libraries* tab of the *Build Properties* dialog and selecting with the *Add...* button:



11.6.4 Adding Project Definitions

To identity the build target *os_$LITE_TARGET* should always be defined.

Lite compiler defines can be added to a project by right clicking the project, selecting *Properties*, selecting the *#Defines* tab of the *Build Properties* dialog and selecting with the *Add...* button:



11.7 Using the VxWorks Simulator

VxWorks provides a simulation environment which can be used to run Lite applications. See the *Wind River VxWorks Simulator User's Guide* for full details of the target board and network simulators.

A simple way to use the simulator is to build a kernel, run the kernel in the board simulator (vxsim), and run RTPs from the kernel console. The RTPs can be accessed directly from the host file system, which is visible from the simulator.

A script to build a suitable kernel image is provided in *$LITE_HOME/etc/vxworks/build-sim-image.sh*. The script takes one parameter which is the location of a directory in which to build the kernel:

```
$LITE_HOME/etc/vxworks/build-sim-image.sh <projdir>
```

The image will be *<projdir>/default/vxWorks*.

11.7.1 Using the Network Simulator

The board simulator can make use of a simulated network, which is provided by the network simulator daemon (vxsimnetd). By using this the simulation can be kept separate from the host machine's network.

The simulated network can be defined in a configuration file. A simple configuration file is provided in *$LITE_HOME/etc/vxworks/simnet-config*. This defines a network which has the address 192.168.200.0, configured for broadcast and multicast.

11.7.2 Running the examples

First start the network simulator daemon:

```
vxsimnetd -f $LITE_HOME/etc/vxworks/simnet-config &
```

Then start two board simulators in different windows:

```
vxsim -hostname <host> -h <host IP> -d simnet -e <board IP> -p <n> -f <kernel>
```

where:

- *host* is the name of the host on which the simulation is running
- *host IP* is the IP address of the host
- *board IP* is the IP address of the simulated board in the simulated network
- *n* is the "processor number", to distinguish simulator instances
- *kernel* is the full path to the kernel image

11.7.3 For RTP examples

This will display the VxWorks console in each window. Now run the required RTP. For a static build:

```
cmd
<path to .vxe file>
```

For a build using shared libraries the LD_LIBRARY_PATH must be set. It must include the path to the LITE libraries, and the path to the VxWorks simulator libraries:

```
cmd
set env "LD_LIBRARY_PATH=$LITE_HOME/lib/vxworks_gnu_x86_linux;/usr/local/VxWorks69/vxworks-6.9/ta
<path to .vxe file>
```

For example, to run the static throughput publisher:

```
cmd
$LITE_HOME/examples/c/throughput.vxe 100 100 100 10 "Throughput example"
```

11.7.4 For Kernel space examples

This will display the VxWorks console in each window. Now load and run the required DKM. For a static build:

```
ld 1,0,"<path to .out file>"
<example_name> <example arguments>
```

For example, to run the static throughput publisher:

```
ld 1,0,"$LITE_HOME/examples/c/throughput.out"
throughput "100 100 100 10 Throughput_example"
```

12

Troubleshooting

Detailed information about ADLINK's product support services, general support contacts and enquiries are described on the ADLINK Support page reached via the ADLINK Home page at http://ist.adlinktech.com/.

12.1 Enabling Logging

By default only log messages of severity warning or greater are logged. Additionally, more verbose levels of logging can be enabled by setting the tracing verbosity in the XML configuration:

```
<DDSI2E>
  <Tracing>
    <Verbosity>finest</Verbosity>
  </Tracing>
</DDSI2E>
```

12.2 Analysing Error Codes

All C99 API raw error codes have bit fields identifying the core error classification, module and minor code for every error. Each generated error code from the C99 API is unique and can be used by support to identify the exact cause of any error. An error checking macro is provided that can generate error report strings to aid in diagnostics:

```
ret = dds_subscriber_create (participant, &subscriber, NULL, NULL);
DDS_ERR_CHECK (ret, DDS_CHECK_REPORT);
```

12.3 Using Debug Libraries

All libraries are provided with both runtime and debug versions, all debug libraries are named <*library*>_*g*. The normal runtime library is optimised for size and speed, whereas the debug library is not optimised, compiled with debug and contains extra runtime checking. It is recommended to develop with debug libraries and deploy with non debug libraries. In the event of a crash, use of a debugger (such as gdb) with the debug libraries may help locate the problem.

12.4 Using Wireshark

The Wireshark protocol analysis tool can be used to capture and display DDSI packets directly off an interface. However if this is not possible, or Lite is installed on a machine without Wireshark support, it is possible to configure Lite to generate Wireshark capture files that can be analysed on another system. To generate Wireshark pcap files, simply set the capture file name in the XML configuration:

```
<DDSI2E>
  <Tracing>
    <PacketCaptureFile>ddsi.pcap</PacketCaptureFile>
  </Tracing>
</DDSI2E>
```

12.5 Managing Application Threads

Application threads must call dds_thread_init() for creating thread specific storage. Any call to reader/writer from the thread function without initialization will result in segmentation fault. Please refer to section 4.4.3 - 'Application Threads' in Lite Usage for details.

13

Supported Features

13.1 Supported DDS Profiles

Profile	Support	Limitations
Minimum profile	Partial	**Global** : **Participant** : *lookup_topicdescription*, *create_multitopic* and *delete_multitipic* are not supported **Publisher** : *wait_for_acknowledgments* is not supported **DataWriter** : *wait_for_acknowledgments*, *get_matched_subscriptions* and *get_matched_subscription_data* are not supported **Subscriber** : *get_datareaders* and *lookup_datareaders* are not supported **DataReader** : *get_matched_publications* and *get_matched_publication_data* are not supported
Content-subscription profile	**YES**	Supported via C filter functions as opposed to SQL filter statements
Persistence profile	NO	
Ownership profile	**YES**	

13.2 Supported QoS Policies

QoS Category	*QoS Policy*	*Supported*
Data Availability		
	DURABILITY	VOLATILE, TRANSIENT_LOCAL with KEEP_LAST with depth 1, TRANSIENT and PERSISTENT (as a client only)
	DURABILITY_SERVICE	**YES**
	LIFESPAN	NO
	HISTORY	**YES** (Reader Only)
	WRITER_DATA_LIFECYCLE	**YES**
	READER_DATA_LIFECYCLE	NO
Data Delivery		
	PRESENTATION	NO
	RELIABILITY	**YES**
	PARTITION	**YES**
	DESTINATION_ORDER	**YES** (Reader Only)
	OWNERSHIP	**YES**
	OWNERSHIP_STRENGTH	**YES**
Data Timeliness		
	DEADLINE	NO
	LATENCY_BUDGET	NO
	TRANSPORT_PRIORITY	NO
Resources		
	TIME_BASED_FILTER	NO
	RESOURCE_LIMITS	**YES** (Reader Only)
Configuration		
	USER_DATA	**YES**
	TOPIC_DATA	**YES**
	GROUP_DATA	**YES**
	ENTITY_FACTORY	NO
System Availability		
	LIVELINESS	AUTOMATIC only

13.3 Supported DDS Statuses

Entity	**Status Name**	**Supported**
Topic	INCONSISTENT_TOPIC	**YES**
Subscriber	DATA_ON_READERS	**YES**
DataReader	SAMPLE_REJECTED	**YES**
DataWriter	LIVELINESS_CHANGED	Partial
	REQUESTED_DEADLINE_MISSED	NO
	REQUESTED_INCOMPATIBLE_QOS	**YES**
	DATA_AVAILABLE	**YES**
	SAMPLE_LOST	**YES**
	SUBSCRIPTION_MATCHED	**YES**
	LIVELINESS_LOST	NO
	OFFERED_DEADLINE_MISSED	NO
	OFFERED_INCOMPATIBLE_QOS	**YES**
	PUBLICATION_MATCHED	**YES**

14

Contacts & Notices

14.1 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

14.2 Notices