The Data Distribution Service

Angelo Corsaro
Chief Technology Officer
PrismTech
angelo.corsaro@prismtech.com

February 14, 2014

Contents

Chapter 1

Foundations

1.1 The Data Distribution Service

Whether you are an experienced programmer or a newbie, it is highly likely that you have already experienced some form of Publish/Subscribe (Pub/Sub) – an abstraction for one-to-many communication that provides anonymous, decoupled, and asynchronous communication between the publisher and its subscribers. Pub/Sub is the abstraction behind many of the technologies used today to build and integrate distributed applications, such as, social application, financial trading, etc., while maintaining their composing parts loosely coupled and independently evolvable.

Various implementations of the Pub/Sub abstraction have emerged through time to address the needs of different application domains. DDS is an Object Management Group (OMG) standard for Pub/Sub introduced in 2004 to address the data sharing needs of large scale mission- and business-critical applications. Today DDS is one of the hot technologies at the heart of some of the most interesting Internet of Things (IoT) and Industrial Internet (I2) applications.

At this point, to the question ”What is DDS?” I can safely answer that it is a Pub/Sub technology for ubiquitous, polyglot, efficient and secure data sharing. Another way of answering this question is that DDS is Pub/Sub on *steroids*.

1.2 The OMG DDS Standard

The DDS standards family is today composed, as shown in Figure 1.2, by the DDS v1.2 API **??** and the Data Distribution Service Interoperability Wire Protocol (DDSI) (DDSI v2.1) **??**. The DDS API standard guarantees source code portability across different vendor implementations, while the DDSI standard ensures on the wire interoperability between DDS implementations from different vendors. The DDS API standard defines several different profiles (see Figure 1.2) that enhance real-time pub/sub with content filtering and queries, temporal decoupling and automatic fail-over.

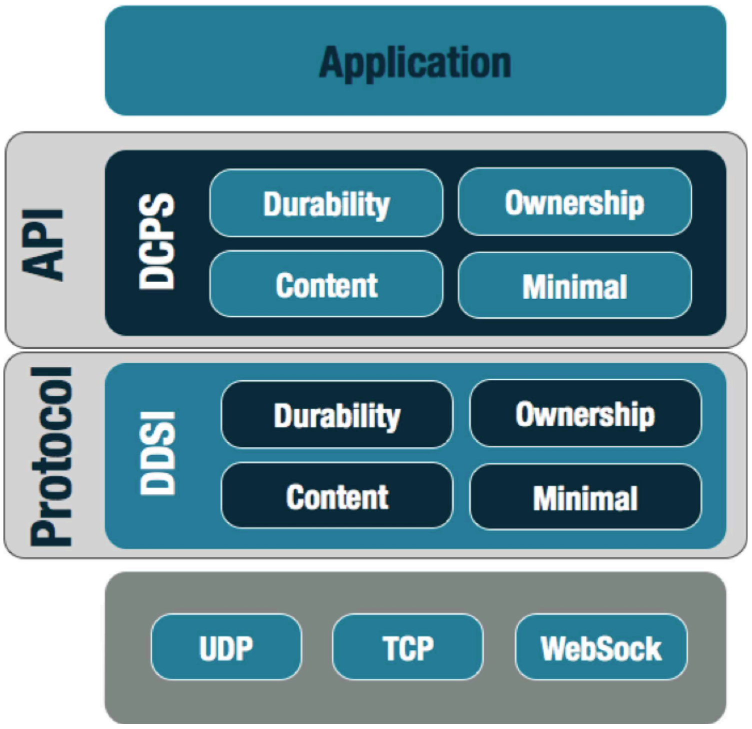

Figure 1.1: The DDS Standard.

The DDS standard was formally adopted by the OMG in 2004 and today it has become the established Pub/Sub technology for distributing high volumes of data, dependably and with predictable low latency in applications such as, Smart Grids, Smart Cities, Financial Trading, Air Traffic Control and Management, High Performance Telemetry and Large Scale Supervisory Systems.

Now that I have given you an overall idea of what DDS is and where it is being used let's try to see how it works.

1.3 DDS in a Nutshell

To explain DDS I will take advantage of a running example that is simple and generic enough that you should easily relate to it. I will describe the example now and then use it to explain the various DDS features throughout this tutorial. The example that I will use is the temperature monitoring and control system for a very large building. Each floor of the building has several rooms, each of which is equipped with a set of temperature and humidity sensors and one or more conditioners. The application is supposed to perform both monitoring for all the elements in the building as well as temperature and humidity control for each room.

This application is a typical distributed monitoring and control application in which you have data telemetry from several sensors distributed over some spatial location and you also have the control action that has to be applied to the actuators–our conditioners.

Now that we've created a task for you to solve, let's see what DDS has to offer.

1.3.1 Global Data Space

The key abstraction at the foundation of DDS is a fully distributed . It is important to remark that the DDS specification requires the implementation of the Global Data Space to be fully distributed to avoid point of failures and single point of bottleneck. Publishers and Subscribers can join or leave the Global Data Space (GDS) at any point in time as they are dynamically discovered. The dynamic discovery of Publisher and Subscribers is performed by the GDS and does not rely on any kind of centralized registry such as those found in other pub/sub technologies such as the Java Message Service (JMS). Finally, I should mention that the GDS also discovers application defined data types and propagates them as part of the discovery process. In essence, the presence of a GDS equipped with dy-

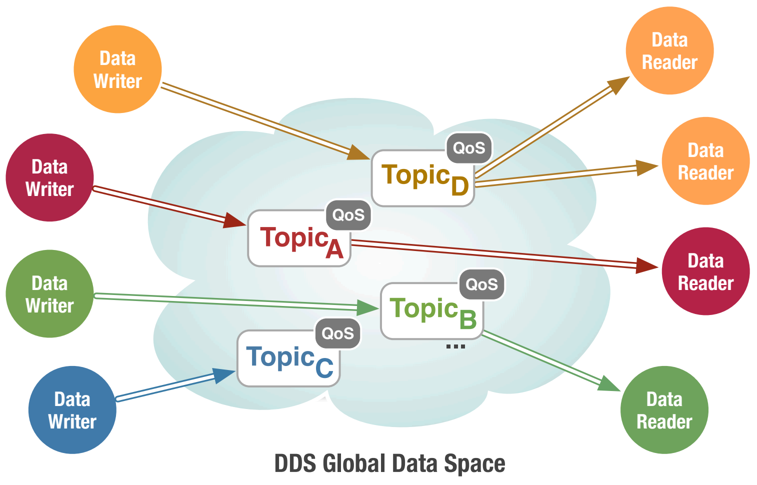

Figure 1.2: The Global Data Space.

namic discovery means that when you deploy a system, you don't have to configure anything. Everything will be automatically discovered and data will begin to flow. Moreover, since the GDS is fully distributed you don't have to fear the crash of some server inducing unknown consequences on the system availability – in DDS there is no single point of failure, although applications can crash and restart, or connect/disconnect, the system as a whole continues to run.

1.3.2 Domain Participants

To do anything useful a DDS applications needs to create a Domain Participant (DP). The DP gives access to the GDS – called domain in DDS applications. Listing 1.1 shows how a DP can be created, notice that domains are identified by integers.

Listing 1.1: Topic creation.

```
// Creates a domain participant in the default domain
DomainParticipant dp1(DomainParticipant::default_domain());

// Creates a domain participant in the domain identified by
// the number 18
DomainParticipant dp2(18);
```

1.3.3 Topics

I've evoked several times this vision of the data flowing from Publishers to Subscribers. In DDS this data belongs to a Topic and represents the unit of information that can be produced or consumed. A Topic is defined as a triad composed of a type, a unique name, and a set of Quality of Service (QoS) policies which, as I'll explain in details later in this tutorial, are used to control the non-functional properties associated with the Topic. For the time being it is enough to say that if the Quality of Service (QoS)s are not explicitly set, then the DDS implementation will use some defaults prescribed by the standard.

Topic Types can be represented with the subset of the OMG Interface Definition Language (IDL) standard that defines struct types, with the limitations that Any-types are not supported. If you are not familiar with the IDL standard you should not worry as essentially, it is safe for you to think that Topic Types are defined with "C-like" structures whose attributes can be primitive types, such as short, long, float, string, etc., arrays, sequences, union and enumerations. Nesting of structures is also allowed. On the other hand, If you are familiar with IDL I am sure you are now wondering how DDS relates to CORBA. The only things that DDS has in common with CORBA is that it uses a subset of IDL; other than this, CORBA and DDS are two completely different Standards and two completely different and complementary technologies.

Now, getting back to our temperature control application, you might want to define topics representing the reading of temperature sensors, the conditioners and perhaps the rooms in which the temperature sensors and the conditioner are installed. Listing 1.2 provides an example of how you might define the topic type for the temperature sensor.

Listing 1.2: The IDL definition for the Temperature Sensor

```
enum TemperatureScale {
  CELSIUS,
  KELVIN,
  FAHRENHEIT
};

struct TempSensorType {
  float temp;
  float hum;
  TemperatureScale scale;
  short id;

};
#pragma keylist TempSensor id
```

As Listing 1.2 reveals, IDL structures really look like C/C++ structures, as a result learning to write Topic Types is usually effortless for most programmers. If you are a "detail-oriented" person you'll have noticed that the Listing 1.2 also includes a suspicious `#pragma keylist` directive. This directive is used to specify keys. The `TempSensorType` is specified to have a single key represented by the sensor identifier (id attribute). At runtime, each key value will identify a specific stream of data, more precisely, in DDS we say that each key-value identifies a Topic instance. For each instance it is possible for you to observe the life-cycle and learn about interesting transitions such as when it first appeared in the system, or when it was disposed. Keys, along with identifying instances, are also used to capture data relationships as you would in traditional entity relationship modeling. Keys can be made up by an arbitrary number of attributes, some of which could also be defined in nested structures.

Once defined the topic type, you can programmatically register a DDS topic using the DDS API by simply instantiating a `Topic` class with proper type and name.

Listing 1.3: Topic creation.

```
Topic<TempSensorType> topic(dp, "TempSensorTopic");
```

1.3.4 Reading and Writing Data

Now that you have seen how to specify topics it is time to explore how you can make this Topics flow between Publishers and Subscribers. DDS uses the specification of user-defined Topic Types to generate efficient encoding and decoding routines as well as strongly typed DataReaders and DataWriters.

Creating a DataReader or a DataWriter is pretty straightforward as it simply requires to construct an object by instantiating a template class with the Topic Type and passing by the desired Topic object. After you've created a DataReader for your ”TempSensorTopic” you are ready to read the data produced by temperature sensors distributed in your system. Likewise after you've created a DataWriter for your “TempSensorTopic” you are ready to write (publish) data. Listing 1.4 and 1.5 show the steps required to do so.

If you look a bit closer to Listing 1.5, you'll see that our first DDS application is using polling to read data out of DDS every second. A sleep is used to avoid spinning in the loop to fast since the DDS read is non-blocking and returns right away if there is no data available. Although polling is a good way to write your first DDS examples it is good to know that DDS supports two ways for informing your application of data availability, listeners and waitsets. Listeners can be registered with readers for receiving notification of data availability as well as several other interesting status changes such as violation in QoS. Waitsets, modeled after the Unix-style select call, can be used for waiting the happening of interesting events, one of which could be the availability of data. I will detail these coordination mechanism later on in this tutorial.

Listing 1.4: Writing Data in DDS.

```
// create a Domain Participant
DomainParticipant dp(default_id());
// create the Topic
Topic<TempSensorType> topic("TempSensorTopic");
// create a Publisher
Publisher pub(dp);
// Create a DataWriter
DataWriter<TempSensorType> dw(dp, topic);

TempSensorType ts = {1, 26.0F, 70.0F, CELSIUS};
// Write Data
dw.write(ts);
// Write some more data using streaming operators
dw << TempSensorType(2, 26.5F, 74.0F, CELSIUS);
```

I think that looking at this code you'll be a bit puzzled since the data reader and the data writer are completely decoupled. It is not clear where they are writing data to or reading it from, how they are finding about each other and so on. This is the DDS magic! As I had explained in the very beginning of this Chapter DDS is equipped with dynamic discovery of both participants as well as user-defined data types. Thus it is DDS that discovers data produces and consumers and takes care of matching them. My strongest recommendation is that you try to compile the code examples available online (see Appendix A) and run them on your machine or even better on a couple of machines. Try running one writer and several readers. Then try adding more writers and see what happens. Also experiment with arbitrary killing (meaning kill -9) readers/writers and restarting them. This way you'll see the dynamic discovery in action.

Listing 1.5: Reading Data in DDS.

```
// create a Domain Participant
DomainParticipant dp(default_id());
// create the Topic
Topic<TempSensorType> topic("TempSensorTopic");
// create a Publisher
Publisher pub(dp);
// create a DataReader
while (true) {
  LoanedSamples<TempSensorType> samples = dr.read();
  std::for_each(dr.begin(),
                dr.end(),
                [](const Sample<TempSensorType>& s) {
                  std::cout << s.data() << std::endl;
                });
  std::this_thread::sleep_for(std::chrono::seconds(1));
 }
```

1.4 Summary

In this first chapter I've explained the abstraction behind DDS and introduced some of its core concepts. I've also shown you how to write your first DDS applications that distributes temperature sensors values over a distributed system. This was an effort taking less than 15 lines of code in total, remarkable, isn't it? In the upcoming chapters I'll introduce you to more advanced concepts provided by DDS and by the end of the series you should be able to use all the DDS features to create sophisticated scalable, efficient and real-time Pub/Sub applications.

Chapter 2

Topics, Domains and Partitions

In the previous chapter I introduced the basics of DDS and walked you through the steps required to write a simple pub/sub application. Now it is time to start looking in more depth at DDS and we have no better place to start than data management.

2.1 Topics Inside Out

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

2.1.1 Topic Types

As DDS is independent of the programming language as well as the **OS!** (**OS!**) it defines its type system [1] along with an space and time efficient binary encoding for its types. Different syntaxes can be used to express DDS topic types, such as IDL, XML, and annotated Java.

In this Tutorial I will be focusing on the subset of IDL that can be used to define a topic type. A topic type is made by an IDL struct plus a key. The struct can contain as many fields as you want and each field can be a primitive type (see Table **??**), a template type (see Table 2.1), or a constructed type (see Table 2.2).

As shown in Table **??**, primitive types are essentially what you'd expect, with just one exception – the int type is not there! This should not worry you since the IDL integral types short, long and long long are equivalent to the C99 int16_t, int32_t and int64_t thus you have all you need. Table 2.1 shows IDL templates types. The `string` and `wstring` can be parametrized only with respect to their maximum length; the sequence type with respect

Primitive Types	Size (bits)
`boolean`	8
`octet`	8
`char`	8
`wchar`	16
`short`	16
`unsigned short`	16
`long`	32
`unsigned long`	32
`long long`	64
`unsigned long long`	64
`float`	32
`double`	64

Template Type	Example

`string<length = UNBOUNDED$>`	`string s1;`
	`string<32> s2;`

`wstring<length = UNBOUNDED>`	`wstring ws1;`
	`wstring<64> ws2;`

`sequence<T,length = UNBOUNDED>`	`sequence<octet> oseq;`
	`sequence<octet, 1024> oseq1k;`
	`sequence<MyType> mtseq;`
	`sequence<MyType, $10>$ mtseq10;`

`fixed<digits,scale>`	`fixed<5,2> fp; //d1d2d3.d4d5`

Table 2.1: IDL Template Types

Constructed Type	Example
`enum`	`enum Dimension {1D, 2D, 3D, 4D};`

`struct`	`struct Coord1D { long x;};`
	`struct Coord2D { long x; long y; };`
	`struct Coord3D { long x; long y; long z; };`
	`struct Coord4D { long x; long y; long z,`
	`unsigned long long t;};`

`union`	`union Coord switch (Dimension) {`
	`case 1D: Coord1D c1d;`
	`case 2D: Coord2D c2d;`
	`case 3D: Coord3D c3d;`
	`case 4D: Coord4D c4d;`
	`};`

Table 2.2: IDL Template Types

to its length and contained type; the fixed type with respect to the total number of digits and the scale. The sequence type abstracts homogeneous random access container, pretty much like the `std::vector` in C++ or `java.util.Vector` in Java. Finally, it is important to point out that when the maximum length is not provided the type is assumed as having an unbounded length, meaning that the middleware will allocate as much memory as necessary to store the values the application provides. Table 2.2 shows that DDS supports three different kinds of IDL constructed types, `enum`, `struct`, and `union`. Putting it all together, you should now realize that a Topic type is a struct that can contain as fields nested structures, unions ,enumerations, template types as well as primitive types. In addition to this, you can define multi-dimensional arrays of any DDS-supported or user-defined type.

This is all nice, but you might wonder how this it ties-in with programming languages such as C++, Java, C#. The answer is not really surprising, essentially there is a language-specific mapping from the IDL-types described above to mainstream programming languages.

2.1.2 Topic Keys, Instances and Samples

Each Topic comes with an associated key-set. This key-set might be empty or it can include an arbitrary number of attributes defined by the Topic Type. There are no limitations on the number, kind, or level of nesting, of attributes used to establish the key.

Listing 2.1: Keyed and Keyless Topics.

```
enum TemperatureScale {
      CELSIUM,
      KELVIN,
        FARENHEIT
};

struct TempSensorType {
        short id;
        float temp;
        float hum;
     TemperatureScale scale;

};
#pragma keylist TempSensorType id
```

If we get back to our running example, the temperature control and monitoring system, we could define a keyless variant of the `TempSensorType` defined in Chapter 1. Listing 2.1 shows our old good `TempSensorType` with the id attribute defined as its key, along with the `KeylessTempSensorType` showing-off an empty key-set as defined in its #pragma keylist directive.

If we create two topics associated with the types declared in Listing 2.1 what would be the exact difference between them?

```
Topic<KeylessTempSensorType> kltsTopic(dp, "KLTempSensorTopic");
Topic<TempSensorType> tsTopic(dp, "TempSensorTopic");
```

The main difference between these two topics is their number of instances. Keyless topics have only once instance, thus can be thought as singletons. Keyed topics have once instance per key-value. Making a parallel with classes in object oriented programming languages, you can think of a Topic as defining a class whose instances are created for each unique value of the topic keys. Thus if the topic has no keys you get a singleton.

Topic instances are runtime entities for which DDS keeps track of whether (1) there are any live writers, (2) the instance has appeared in the system for the first time, and (3) the instance has been disposed–meaning explicitly removed from the system. Topic instances impact the organization of data on the reader side as well as the memory usage. Furthermore, as we will see later on in the series, there are some QoS that apply at an instance-level.

Let me now illustrate what happens when you write a keyless topic versus a keyed topic. If we write a sample for the `KLSensorTopic` this is going to modify the value for exactly the same instance, the singleton, regardless of the content of the sample. On the other, each sample you write for the `TempSensorTopic` will modify the value of a specific topic instance, depending on the value of the key attributes, the id in our example.

Thus, the code below is writing two samples for the same instance, as shown in Figure 2.1.

```
DataWriter<KeylessTempSensorType> kldw(pub, kltsTopic);
TempSensorType ts = {1, 26.0F, 70.0F, CELSIUS};
kldw.write(ts);
ts = {2, 26.0F, 70.0F, CELSIUS};
kldw.write(ts);
```

These two samples will be posted in the same reader queue; the queue associated with the singleton instance, as shown in Figure 2.1. If we write the

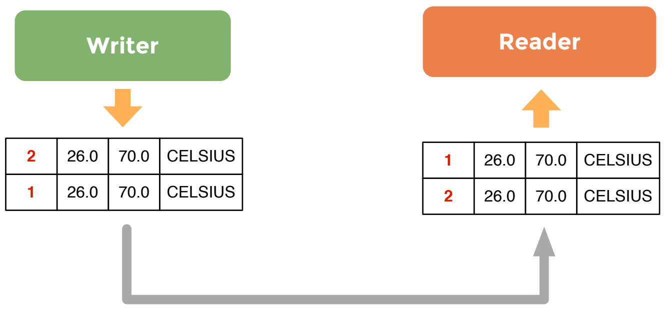
Writer
Reader
2 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
2 | 26.0 | 70.0 | CELSIUS

Figure 2.1: Data Reader queues for a keyless Topics.

same samples for the `TempSensorTopic`, the end-result is quite different. The two samples written in the code fragment below have two different id values, respectively 1 and 2, as a result they are referring to two different instances.

```
DataWriter<EventCountType>  kdw(pub, ecTopic);
TempSensorType ts = {1, 26.0F, 70.0F, CELSIUS};
kdw.write(ts);
ts = {2, 26.0F, 70.0F, CELSIUS};
kdw.write(ts);
```

n this case the reader will see these two samples posted into two different queues, as represented in Figure 2.2, one queue for each instance. In

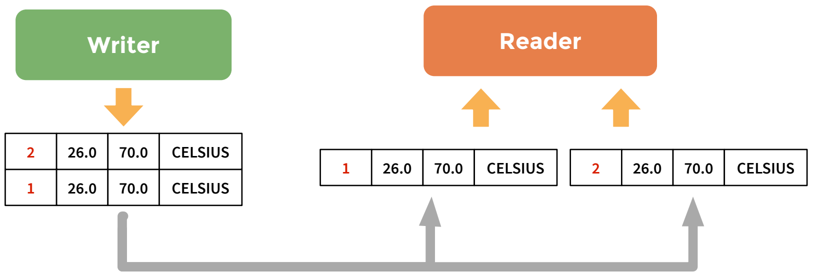
Writer
Reader
2 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
2 | 26.0 | 70.0 | CELSIUS

Figure 2.2: Data Reader queues for a keyed Topics.

summary, you should think of Topics as classes in an object oriented language and understand that each unique key-value identifies an instance. The life-cycle of topic instances is managed by DDS and to each topic instance are associated memory resources, you can think of it as a queue on the reader side. Keys identify specific data streams within a Topic. Thus, in our running example, each id value will identify a specific temperature sensor. Differently from many other Pub/Sub technologies, DDS allows you to use keys to automatic demultiplex different streams of data. Furthermore, since each temperature sensor will be representing an instance of the `TempSensorTopic` you'll be able to track the lifecycle of the sensor by tracking the lifecycle of its associated instance. You can detect when a new sensor is added into the system, just because it will introduce a new instance, you can detect when a sensor has failed, thanks to the fact that DDS can inform you when there are no more writers for a specific instance. You can even detect when a sensor has crashed and then recovered thanks to some information about the state transition that are provided by DDS.

Finally, before setting to rest DDS instances, I want to underline that DDS subscriptions concerns Topics. As a result when subscribing to a topic, you'll receive all the instances produced for that topic. In some cases this is not desirable and some scoping actions are necessary. Let's see then what DDS has to offer.

2.2 Scoping Information

2.2.1 Domain

DDS provides two mechanism for scoping information, domains and partitions. A domain establishes a virtual network linking all the DDS applications that have joined it. No communication can ever happen across domains unless explicitly mediated by the user application.

2.2.2 Partition

Domains can be further organized into partitions, where each partition represent a logical grouping of topics. DDS Partitions are described by names such as "SensorDataPartition", "CommandPartition", "LogDataPartition", etc., and have to be explicitly joined in order to publish data in it or subscribe to the topics it contains. The mechanism provided by DDS for joining a partition is very flexible as a publisher or a subscriber can join by providing its full name, such as "SensorDataPartition" or it can join all the partitions that match a regular expression, such as "Sens*", or "*Data*". Supported regular expressions are the same as those accepted by the POSIX `fnmatch` [2] function.

To recap, partitions provide a way of scoping information. You can use this scoping mechanism to organize topics into different coherent sets. You can equally use partitions to segregate topic instances. Instance segregation can be necessary for optimizing performance or minimizing footprint for those applications that are characterized by a very large number of instances, such as large telemetry systems, or financial trading applications. If we take as an example our temperature monitoring and control system, then we can devise with a very natural partitioning of data that mimics the physical

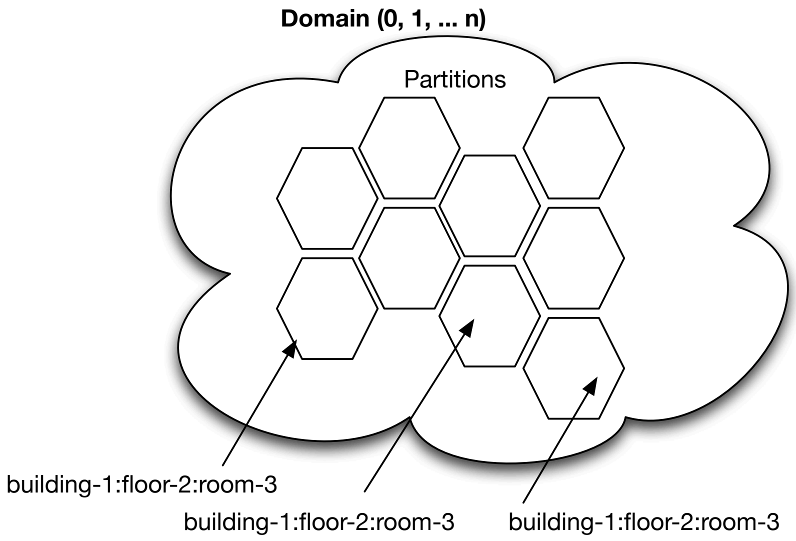
Domain (0, 1, ... n)
Partitions
building-1:floor-2:room-3
building-1:floor-2:room-3
building-1:floor-2:room-3

Figure 2.3: Domain and partitions in DDS.

placement of the various temperature sensors. To do this, we can use a partition naming scheme made of the building number, the floor level and the room number in which the sensor is installed:

```
"building-<number>:floor-<level>:room-<number>"
```

Using this naming scheme, as shown in Figure 2.3, all the topics produced in room 51 on the 15th floor on building 1 would belong to the partition `"building-1:floor-15:room-51"`. Likewise, the partition expression `"building-1:floor-1:room-*"` matches all the partitions for the rooms at the first floor in building.

In a nutshell, you can use partitions to scope information, you can also use naming conversions such as those used for our temperature control applications to emulate hierarchical organization of data starting from flat partitions. Using the same technique you can slice and access data across different dimensions or views, depending on the need of your application.

2.3 Content Filtering

Domains and Partitions are useful mechanisms for structurally organizing data, but what if you need to control the data received based on its content? Content Filtering allows you to create topics that constrain the values their instances might take. When subscribing to a content-filtered topic an application will only receive, among all published values, only those that match the topic filter. The filter expression can operate on the full topic

Constructed Type	**Example**
=	equal
<>	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
BETWEEN	between and inclusive range
LIKE	matches a string pattern

Table 2.3: Legal operators for DDS Filters and Query Conditions

content, as opposed to being able to operate only on headers as it happens in many other pub/sub technologies, such as JMS. The filter expression is structurally similar to a SQL WHERE clause. The operators supported by are listed in Table **??**.

Content-Filtered topics are very useful from several different perspectives. First of all they limit the amount of memory used by DDS to the instances and samples that match the filter. Furthermore, filtering can be used to simplify your application by delegating to DDS the logic that checks certain data properties. For instance, if we consider our temperature control application we might be interested in being notified only then the temperature or the humidity are outside a given range. Thus assuming we wanted to maintain the temperature between 20.5 °C and 21.5 °C and the humidity between 30% and 50%, we could create a Content-Filtered topic that would alert the application when the sensor is producing value outside our desired set point. This can be done by using the filter expression below:

```
((temp NOT BETWEEN (20.5 AND 21.5))
   OR
(hum NOT BETWEEN (30 AND 50)))
```

Listing 2.2 shows the code that creates a content-filtered topic for the TempSensor topic with the expression above. Notice that the content-filtered topic is created starting from a regular topic. Furthermore it is worth noticing that the filter expression is relying on positional arguments %0, %2, etc., whose actual values are passed via a vector of strings.

Listing 2.2: Content Filtered Topic.

```
// Create the TempSensor topic
Topic<TempSensorType> topic(dp,"TempSensor");

// Define the filter predicate
std::string predicate_ =
   "(temp␣NOT␣BETWEEN␣(%0␣AND␣%1))␣\
␣␣␣␣␣OR␣\
␣␣␣␣(hum␣NOT␣BETWEEN␣(%2␣and␣%3))";
```

```
// Define the filter parameters
std::vector<std::string> params =
   {"20.5", "21.5", "30", "50"};

// Create the ContentFilteredTopic
ContentFilteredTopic<TempSensorType>
cftopic("CFTempSensor",
        topic,
        predicate,
        params);

// This data reader will only receive data that
// maches the content filter
DataReader<TempSensorType> dr(sub, cftopic);
```

2.4 Summary

In this chapter I have covered the most important aspects of data management in DDS. By now you've learned all you needed to know about topics-types and topic instances and you've been exposed to the the various mechanism provided by DDS for scoping information. In summary, you can organize structurally your information by means of domains and partitions. Then you can create special views using content-filtered topics and query conditions. At this point my usual suggestion is that you try to compile and run the examples and try to experiment a bit by yourself.

Chapter 3

Reading and Writing Data

In the previous chapter I covered the definition and semantics of DDS topics, topic-instances and samples. I also went through domains and partitions and the role they play in organizing application data flows. In this chapter I'll examine the mechanisms provided by DDS for reading and writing data.

3.1 Writing Data

As you've seen in the first two chapters, writing data with DDS is as simple as calling the write method on the `DataWriter`. Yet, to make you into a fully proficient DDS programmer there are a few more things you should know. Most notably, the relationship between writers and topic-instances life-cycle. To explain the difference between topics and topic instances I drew the analogy between DDS topics/topic-instances and classes/objects in an Object Oriented Programming language, such as Java or C++. Like objects, topic-instances have (1) an identity provided by their unique key value, and (2) a life-cycle. The topic-instance life-cycle can be implicitly managed through the semantics implied by the `DataWriter`, or it can be explicitly controlled via the `DataWriter` API. The topic-instances life-cycle transition can have implications on local and remote resource usage, thus it is important that you understand this aspect.

3.1.1 Topic-Instances Life-cycle

Before getting into the details of how the life-cycle is managed, let's see which are the possible states. A topic instance is `ALIVE` if there is at least one `DataWriter` that has explicitly or implicitly (through a write) registered it. An instance is in the `NOT_ALIVE_NO_WRITERS` state when there are no more `DataWriters` writing it. Finally, the instance is `NOT_ALIVE_DISPOSED` if it was disposed either implicitly, due to some default QoS settings, or explicitly, by means of a specific `DataWriter` API call. The `NOT_ALIVE_DISPOSED` state indicates that the instance is no more relevant for the system and that it won't be written anymore (or any-time soon) by any writer. As a result the resources allocated throughout the system for storing the instance can be freed. Another way of thinking about this, is that an instance defines a live data element in the DDS global data space as far as there are writers for it. This data element ceases to be alive when there are no more writers for it. Finally, once disposed this data entity can be reclaimed from the DDS global data space, thus freeing resources.

3.1.2 Automatic Life-cycle Management

Let's try to understand the instances life-cycle management with an example. If we look at the code in Listing 3.1, and assume this is the only application writing data, the result of the three write operations is to create three new topic instances in the system for the key values associated with the $id = 1, 2, 3$ (we defined the `TempSensorType` in the first installment as having a single attribute key named id). These instances will be in the `ALIVE` state as long as this application will be running, and will be automatically registered, we could say associated, with the writer. The default behavior for DDS is then to dispose the topic instances once the `DataWriter` object is destroyed, thus leading the instances to the `NOT_ALIVE_DISPOSED` state. The default settings can be overridden to simply induce instances' unregistration, causing in this case a transition from `ALIVE` to `NOT_ALIVE_NO_WRITERS`.

Listing 3.1: Automatic management of Instance life-cycle.

```
#include <thread>
#include <chrono>
#include <TempSensorType_Dcps.h>

using namespace dds::domain;
using namespace dds::sub;

int main(int, char**) {
  DomainParticipant dp(default_ip());
  Topic<TempSensorType> topic(dp, "TempSensorTopic");
  Publisher pub(dp);
  DataWriter<TempSensorType> dw(pub, topic);

  TempSensorType ts;
  //[NOTE #1]: Instances implicitly registered as part
  // of the write.
  //   {id,  temp   hum    scale};
  ts = {1,   25.0F, 65.0F, CELSIUS};
  dw.write(ts);

  ts = {2,   26.0F, 70.0F, CELSIUS};
  dw.write(ts);
  ts = {3,   27.0F, 75.0F, CELSIUS};
  dw.write(ts);

  std::this_thread::sleep_for(std::chrono::seconds(10));
  //[NOTE #2]: Instances automatically unregistered and
  // disposed as result of the destruction of the dw object
  return 0;
}
```

3.1.3 Explicit Life-cycle Management

Topic-instances life-cycle can also be managed explicitly via the API defined on the `DataWriter`. In this case the application programmer has the control on when instances are registered, unregistered and disposed. Topic-instances registration is a good practice to follow whenever an applications writes an instance very often and requires the lowest latency write. In essence the act of explicitly registering an instance allows the middleware to reserve resources as well as optimize the instance lookup. Topic-instance unregistration provides a mean for telling DDS that an application is done with writing a specific topic-instance, thus all the resources locally associated with can be safely released. Finally, disposing topic-instances gives a way of communicating DDS that the instance is no more relevant for the distributed system, thus whenever possible resources allocated with the specific instances should be released both locally and remotely. Listing 2 shows and example of how the DataWriter API can be used to register, unregister and dispose topic-instances. In order to show you the full life-cycle management I've changed the default DataWriter behavior to avoid that instances are automatically disposed when unregistered. In addition, to maintain the code compact I take advantage of the new C++11 `auto` feature which leaves it to the the compiler to infer the left hand side types from the right hand side return-type. Listing 2 shows an application that writes four samples belonging to four different topic-instances, respectively those with $id = 0, 1, 2, 3$. The instances with $id = 1, 2, 3$ are explicitly registered by calling the `DataWriter::register_instance` method, while the instance with $id = 0$ is automatically registered as result of the write on the `DataWriter`. To show the different possible state transitions, the topic-instance with $id = 1$ is explicitly unregistered thus causing it to transition to the `NOT_ALIVE_NO_WRITER` state; the topic-instance with $id = 2$ is explicitly disposed thus causing it to transition to the `NOT_ALIVE_DISPOSED` state. Finally, the topic-instance with $id = 0, 3$ will be automatically unregistered, as a result of the destruction of the objects dw and $dwi3$ respectively, thus transitioning to the state `NOT_ALIVE_NO_WRITER`. Once again, as mentioned above, in this example the writer has been configured to ensure that topic-instances are not automatically disposed upon unregistration.

3.1.4 Keyless Topics

Most of the discussion above has focused on keyed topics, but what about keyless topics? As explained in chapter 2, keyless topics are like singletons, in the sense that there is only one instance. As a result for keyless topics the the state transitions are tied to the lifecycle of the data-writer.

Listing 3.2: Explicit management of topic-instances life-cycle.

```
#include <TempSensorType_Dcps.h>

using namespace dds::core;
using namespace dds::domain;
using namespace dds::pub;

int main(int, char**) {
  DomainParticipant dp(default_ip());
  Topic<TempSensorType> topic("TempSensorTopic");

  //[NOTE #1]: Avoid topic-instance dispose on unregister
  DataWriterQos dwqos = dp.default_datawriter_qos()
    << WriterDataLifecycle::ManuallyDisposeUnregisteredInstances();

  //[NOTE #2]: Creating DataWriter with custom QoS.
  // QoS will be covered in detail in article #4.
  dds::DataWriter<TempSensorType> dw(topic, dwqos);

  TempSensorType data = {0, 24.3F, 0.5F, CELSIUS};
  dw.write(data);

  TempSensorType key;
  key.id = 1;

  //[NOTE #3] Registering topic-instance explicitly
  InstanceHandle h1 = dw.register_instance(key);

  key.id = 2;
  InstanceHandle h2 = dw.register_instance(key);

  key.id = 3;
  InstanceHandle h3 = dw.register_instance(key);

  data = {1, 24.3F, 0.5F, CELSIUS};
  dw.write(data);

  data = {2, 23.5F, 0.6F, CELSIUS};
  dw.write(data);

  data = {3, 21.7F, 0.5F, CELSIUS};
  dw.write(data);

  // [NOTE #4]: unregister topic-instance with id=1
  dw.unregister_instance(h1);
  // [NOTE #5]: dispose topic-instance with id=2
  dw.dispose(h2);

  //[NOTE #6]:topic-instance with id=3 will be unregistered as
  // result of the dw object destruction

  return 0;
}
```

3.1.5 Blocking or Non-Blocking Write?

One question that you might have at this point is whether the write is blocking or not. The short answer is that the write is non-blocking, however, as we will see later on, there are cases in which, depending on QoS settings, the write might block. In these cases, the blocking behaviour is necessary to avoid avoid data-loss.

3.2 Accessing Data

DDS provides a mechanism to select the samples based on their content and state and another to control whether samples have to be read or taken (removed from the cache).

3.2.1 Read vs. Take

The DDS provides data access through the `DataReader` class which exposes two semantics for data access: read and the take. The read semantics, implemented by the `DataReader::read` methods, gives access to the data received by the `DataReader` without removing it from its cache. This means that the data will be again readable via an appropriate read call. Likewise, the take semantics, implemented by the `DataReader::take` method, allows to access the data received by the `DataReader` by removing it from its local cache. This means that once the data is taken, it is no more available for subsequent read or take operations. The semantics provided by the read and take operations allow you to use DDS as either a distributed cache or like a queuing system, or both. This is a powerful combination that is rarely found in the same middleware platform. This is one of the reasons why DDS is used in a variety of systems sometimes as a high-performance distributed cache, other like a high-performance messaging technology, and yet other times as a combination of the two. In addition, the read semantics is useful when using topics to model distributed state, while the take semantics when modeling distributed events.

3.2.2 Data and Meta-Data

In the first part of this chapter I showed how the `DataWriter` can be used to control the life-cycle of topic-instances. The topic-instance life-cycle along with other information describing properties of received data samples are made available to `DataReader` and can be used to select the data access via either a read or take. Specifically, each data sample received by a `DataWriter` has associated a `SampleInfo` describing the property of that sample. These properties includes information on:

- **Sample State**. The sample state can be `READ` or `NOT_READ` depending on whether the sample has already been read or not.
- **Instance State.** As explained above, this indicates the status of the instance as being either `ALIVE`, `NOT_ALIVE_NO_WRITERS`, or `NOT_ALIVE_DISPOSED`.
- **View State.** The view state can be `NEW` or `NOT_NEW` depending on whether this is the first sample ever received for the given topic-instance or not.

The `SampleInfo` also contains a set of counters that allow to reconstruct the number of times that a topic-instance has performed certain status transition, such as becoming alive after being disposed. Finally, the `SampleInfo` contains a `timestamp` for the data and a flag that tells wether the associated data sample is valid or not. This latter flag is important since DDS might generate valid samples info with invalid data to inform about state transitions such as an instance being disposed.

3.2.3 Selecting Samples

Regardless of whether data are read or taken from DDS, the same mechanism is used to express the sample selection. Thus, for brevity, I am going to provide some examples using the `read` operation yet if you want to use the `take` operation you simply have to replace each occurrence of a `take` with a `read`.

DDS allows you to select data based on state and content. State-based selection predicates on the values of the view state, instance state and sample state. Content-based selection predicates on the content of the sample.

State-based Selection

For instance, if I am interested in getting all the data received, no matter what the view, instance and sample state would issue read (or a take) as follows:

```
LoanedSamples<TempSensorType> samples =
      dr.select()
              .state(DataState.any())
              .read();
```

If on the other hand I am interested in only reading (or taking) all samples that have not been read yet, I would issue a read (or take) as follows:

```
LoanedSamples<TempSensorType> samples =
      dr.select()
              .state(SampleState.any() << SampleSate.not_read())
              .read();
```

If I want to read new valid data, meaning no samples with only a valid `SampleInfo`, I would issue a read (or take) as follows:

```
LoanedSamples<TempSensorType> samples =
      dr.select()
              .state(DataState.new_data())
              .read();
```

Finally, if I wanted to only read data associated to instances that are making their appearance in the system for the first time, I would issue a read (or take) as follows:

```
LoanedSamples<TempSensorType> samples =
      dr.select()
              .state(DataState(SampleState.not_read(),
                              ViewState.new(),
                              InstanceState.alive())
              .read();
```

Notice that with this kind of read I would only and always get the first sample written for each instance, that's it. Although it might seem a strange use case, this is quite useful for all those applications that need to do something special whenever a new instance makes its appearance in the system for the first time. An example could be a new airplane entering a new region of control, in this case the system might have to do quite a few things that are unique to this specific state transition.

It is also worth mentioning that if the status is omitted, and a read (or a take) is issued as follows:

```
auto samples = dr.read();
```

This is equivalent to selecting samples with the `NOT_READ_SAMPLE_STATE`, `ALIVE_INSTANCE_STATE` and `ANY_VIEW_STATE`.

As a final remark, it is worth pointing out that statuses allow you to select data based on its meta-information.

Constructed Type	Example
=	equal
<>	not equal
>	greater than
<	less than
>=	greater than or equal
<=	less than or equal
BETWEEN	between and inclusive range
LIKE	matches a string pattern

Table 3.1: Legal operators for content query.

Content-based Selection

Content-based selection is supported by DDS through queries. Although the idea of a query could seem overlapping with that of content-filtering (see section) the underlying idea is different. Filtering is about controlling the data received by the data reader – the data that does not matches the filter is not inserted in the data reader cache. On the other hand, queries are about selecting the data that is in the data reader cache.

The syntax supported by query expression is identical to that used to define filter expression and for convenience is reported on Table ??.

The execution of the query is completely under the user control and executed in the context of a `read` or `take` operation as shown in Listing 3.3.

Listing 3.3: Content Query.

```
// Define the query predicate
std::string predicate =
   "(temp␣NOT␣BETWEEN␣(%0␣AND␣%1))␣\
␣␣␣␣␣OR␣\
␣␣␣␣(hum␣NOT␣BETWEEN␣(%2␣and␣%3))";

// Define the query parameters
std::vector<std::string> params =
   {"20.5", "21.5", "30", "50"};

dds::core::Query query(predicate, params);

auto samples = dr.select()
  .query(predicate)
  .read();
```

Instance-based Selection

In some instances you may want to only look at the data coming from a specific topic instance. As instances are identified by value of the key attributes you may be tempted to use content filtering discriminate among them. Although this would work perfectly fine, it is not the most efficient way of selecting an instance. DDS provide another mechanism, that allows you to pinpoint the instance you are interested in a more efficient manner than content filtering. In essence, each instance has associated an instance handle, this can be used to access the data from a given instance in a very efficient manner.

Listing 3.4 shows how this can be done.

Listing 3.4: Instance-based selection.

```
TempSensor key = {123, 0, 0, 0};
auto handle = dr.lookup_instance(key);

auto samples = dr.select()
  .instance(handle)
  .read();
```

3.2.4 Iterators or Containers?

The examples we have seen so far where loaning the data from DDS, in other terms you did not have to provide the storage for the samples. The advantage of this style of read is allows zero copy reads. However, if you want to store the data in a container of your choice you can use the iterator based read/take operations. as iterator-based reads and takes.

The iterator-based read/take API supports both forward iterators as well as back-inserting iterators. These API allows you to read (or take) data into whatever structure you'd like, as far as you can get a forward or a back-inserting iterator for it. If we focus on the forward-iterator based API, the back-inserting is pretty similar, then you might be able to read data as follows:

```
   Sample<TempSensorType>* samples = ...; // Get hold of some memory
   uint32_t n = dr.read(samples, max_samples);

   std::vector<Sample<TempSensorType> vsamples(max_samples);
   n = dr.read(vsamples.begin(), max_samples);
```

3.2.5 Blocking or Non-Blocking Read/Take?

The DDS read and take are always non-blocking. If no data is available to read then the call will return immediately. Likewise if there is less data than requested the call will gather what available and return right away. The non-blocking nature of read/take operations ensures that these can be safely used by applications that poll for data.

3.3 Waiting and being Notified

One way of coordinating with DDS is to have the application poll for data by performing either a read or a take every so often. Polling might be the best approach for some classes of applications, the most common example being control applications that execute a control loop or a cyclic executive. In general, however, applications might want to be notified of the availability of data or perhaps be able to wait for its availability, as opposed to poll. DDS supports both synchronous and asynchronous coordination by means of wait-sets and listeners.

3.3.1 Waitsets

DDS provides a generic mechanism for waiting on conditions. One of the supported kind of conditions are read conditions which can be used to wait for the availability data on one or more `DataReaders`. This functionality is provided by DDS via the `Waitset` class which can be seen as an object oriented version of the Unix `select`.

Listing 3.5: Using WaitSet to wait for data availability.

```
WaitSet ws;
ReadCondition rc(dr, DataState::new_data());
ws += rc;
// Wait for data to be available
ws.wait();
// read data
auto samples = dr.read();
```

DDS conditions can be associated with functor objects which are then used to execute application-specific logic when the condition is triggered. If we wanted to wait for temperature samples to be available we could create a read-condition on our `DataReader` by passing it a functor such as the one showed in Listing **??**. Then as shown in Listing 3.5, we would create a `Waitset` and attach the condition to it. At this point, we can synchronize on the availability of data, and there are two ways of doing it. One approach is to invoke the `Waitset::wait` method which returns the list of active conditions. These active conditions can then be iterated upon and their associated functors can be executed. The other approach is to invoke the `Waitset::dispatch`. In this case the infrastructure will automatically invoke the functor associated with triggered conditions before unblocking.

Notice that, in both cases, the execution of the functor happens in an application thread.

3.3.2 Listeners

Another way of finding-out when there is data to be read, is to take advantage of the events raised by DDS and notified asynchronously to registered handlers. Thus, if we wanted an handler to be notified of the availability of data, we would connect the appropriate handler with the `on_data_available` event raised by the `DataReader`. Listing 3.6 shows how this can be done.

Listing 3.6: Using a listener to receive notification of data availability.

```
LambdaDataReaderListener<TempSensorType> listener;

listener.data_available = [](DataReader<TempSensorType>& dr) {
  auto samples = dr.read();
  std::for_each(samples.begin(), samples.end(),
                [](const Sample<TempSensorType>& s) {
                  std::cout << s.data() << std::endl;
                });

};

dr.listener(&listener, StatusMask::data_available());
```

[inline, caption=Listener Code]Add Listener code example The event handling mechanism allows you to bind anything you want to a DDS event, meaning that you could bind a function, a class method, or a functor. The only contract you need to comply with is the signature that is expected by the infrastructure. For example, when dealing with the `on_data_available` event you have to register a callable entity that accepts a single parameter of type `DataReader`. Finally, something worth pointing out is that the handler code will execute in a middleware thread. As a result, when using listeners you should try to minimize the time spent in the listener itself.

3.4 Summary

In this chapter I have presented the various aspects involved in writing and reading data with DDS. I went through the topic-instance life-cycle, explained how that can be managed via the `DataWriter` and showcased all the meta-information available to `DataReader`. I also went into explaining wait-sets and listeners and how these can be used to receive indication of when data is available. At this point my usual suggestion is that you try to compile and run the examples and try to experiment a bit by yourself.

Chapter 4

Quality of Service

4.1 The DDS QoS Model

DDS provides applications policies to control a wide set of non-functional properties, such as data availability, data delivery, data timeliness and resource usage – Figure 4.1 shows the full list of available QoS. The semantics and the behaviour of DDS entities, such as a topic, data reader, and data writer, can be controlled through available QoS policies. The policies that control and end-to-end property are considered as part of the subscription matching. DDS uses a request vs. offered QoS matching approach, as shown in Figure 4.2 in which a data reader matches a data writer if and only if the QoS it is requesting for the given topic does not exceed (e.g., is no more stringent) than the QoS with which the data is produced by the data writer.

DDS subscriptions are matched against the topic type and name, as well as against the QoS being offered/requested by data writers and readers. This DDS matching mechanism ensures that (1) types are preserved end-to-end due to the topic type matching and (2) end-to-end QoS invariants are also preserved.

The reminder of this section describes the most important QoS policies in DDS.

4.1.1 Data availability

DDS provides the following QoS policies that control the availability of data to domain participants:

- The `DURABILITY` QoS policy controls the lifetime of the data written to the global data space in a DDS domain. Supported durability levels include (1) `VOLATILE`, which specifies that once data is published it is not maintained by DDS for delivery to late joining applications, (2) `TRANSIENT_LOCAL`, which specifies that publishers store data locally so that late joining subscribers get the last published item if a pub-

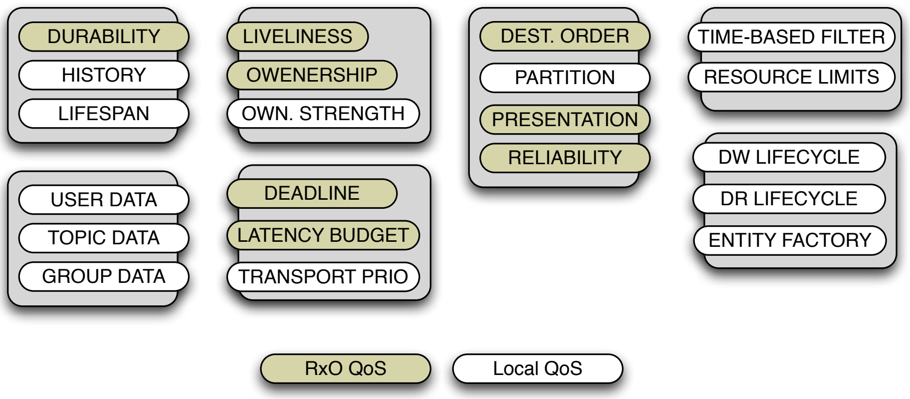

Figure 4.1: DDS QoS Policies.

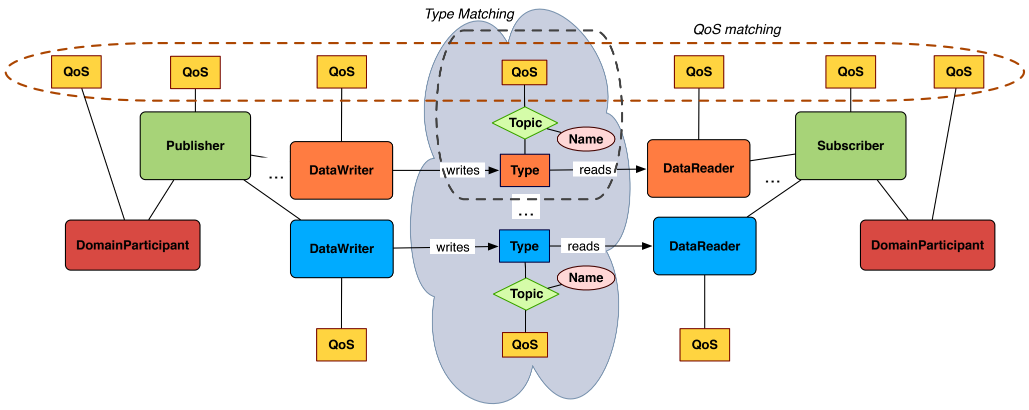

Figure 4.2: DDS Request vs. Offered QoS Model.

lisher is still alive, (3) TRANSIENT, which ensures that the GDS maintains the information outside the local scope of any publishers for use by late joining subscribers, and (4) PERSISTENT, which ensures that the GDS stores the information persistently so to make it available to late joiners even after the shutdown and restart of the whole system. Durability is achieved by relying on a durability service whose properties are configured by means of the DURABILITY_SERVICE QoS of non-volatile topics.

- The LIFESPAN QoS policy controls the interval of time during which a data sample is valid. The default value is infinite, with alternative values being the time-span for which the data can be considered valid.

- The HISTORY QoS policy controls the number of data samples (i.e., subsequent writes of the same topic) that must be stored for readers or writers. Possible values are the last sample, the last n samples, or all samples.

These DDS data availability QoS policies decouple applications in time and space. They also enable these applications to cooperate in highly dynamic environments characterized by continuous joining and leaving of publisher/subscribers. Such properties are particularly relevant in Systems-of-Systems (SoS) since they increase the decoupling of the component parts.

4.1.2 Data delivery

DDS provides the following QoS policies that control how data is delivered and how publishers can claim exclusive rights on data updates:

- The `PRESENTATION` QoS policy gives control on how changes to the information model are presented to subscribers. This QoS gives control on the ordering as well as the coherency of data updates. The scope at which it is applied is defined by the access scope, which can be one of `INSTANCE`, `TOPIC`, or `GROUP` level.
- The `RELIABILITY` QoS policy controls the level of reliability associated with data diffusion. Possible choices are `RELIABLE` and `BEST_EFFORT` distribution.
- The `PARTITION` QoS policy gives control over the association between DDS partitions (represented by a string name) and a specific instance of a publisher/subscriber. This association provides DDS implementations with an abstraction that allow to segregate traffic generated by different partitions, thereby improving overall system scalability and performance.
- The `DESTINATION_ORDER` QoS policy controls the order of changes made by publishers to some instance of a given topic. DDS allows the ordering of different changes according to source or destination timestamps.
- The `OWNERSHIP` QoS policy controls which writer "owns" the write-access to a topic when there are multiple writers and ownership is `EXCLUSIVE`. Only the writer with the highest `OWNERSHIP_STRENGTH` can publish the data. If the `OWNERSHIP` QoS policy value is shared, multiple writers can concurrently update a topic. `OWNERSHIP` thus helps to manage replicated publishers of the same data.

These DDS data delivery QoS policies control the reliability and availability of data, thereby allowing the delivery of the right data to the right place at the right time. More elaborate ways of selecting the right data are offered by the DDS content-awareness profile, which allows applications to select information of interest based upon their content. These QoS policies are particularly useful in SoS since they can be used to finely tune how—and to whom—data is delivered, thus limiting not only the amount of resources used, but also minimizing the level of interference by independent data streams.

4.1.3 Data timeliness

DDS provides the following QoS policies to control the timeliness properties of distributed data:

- The `DEADLINE` QoS policy allows applications to define the maximum inter-arrival time for data. DDS can be configured to automatically notify applications when deadlines are missed.
- The `LATENCY_BUDGET` QoS policy provides a means for applications to inform DDS of the urgency associated with transmitted data. The latency budget specifies the time period within which DDS must distribute the information. This time period starts from the moment the data is written by a publisher until it is available in the subscriber's data-cache ready for use by reader(s).
- The `TRANSPORT_PRIORITY` QoS policy allows applications to control the importance associated with a topic or with a topic instance, thus allowing a DDS implementation to prioritize more important data relative to less important data. These QoS policies help ensure that mission-critical information needed to reconstruct the shared operational picture is delivered in a timely manner.

These DDS data timeliness QoS policies provide control over the temporal properties of data. Such properties are particularly relevant in SoS since they can be used to define and control the temporal aspects of various subsystem data exchanges, while ensuring that bandwidth is exploited optimally.

4.1.4 Resources

DDS defines the following QoS policies to control the network and computing resources that are essential to meet data dissemination requirements:

- The `TIME_BASED_FILTER` QoS policy allows applications to specify the minimum inter-arrival time between data samples, thereby expressing their capability to consume information at a maximum rate. Samples that are produced at a faster pace are not delivered. This policy helps a DDS implementation optimize network bandwidth, memory, and processing power for subscribers that are connected over limited bandwidth networks or which have limited computing capabilities.

- The RESOURCE_LIMITS QoS policy allows applications to control the maximum available storage to hold topic instances and related number of historical samples DDS's QoS policies support the various elements and operating scenarios that constitute net-centric mission-critical information management. By controlling these QoS policies it is possible to scale DDS from low-end embedded systems connected with narrow and noisy radio links, to high-end servers connected to high-speed fiber-optic networks.

These DDS resource QoS policies provide control over the local and end-to-end resources, such as memory and network bandwidth. Such properties are particularly relevant in SoS since they are characterized by largely heterogeneous subsystems, devices, and network connections that often require down-sampling, as well as overall controlled limit on the amount of resources used.

4.1.5 Configuration

The QoS policies described above, provide control over the most important aspects of data delivery, availability, timeliness, and resource usage. DDS also supports the definition and distribution of user specified bootstrapping information via the following QoS policies:

- The USER_DATA QoS policy allows applications to associate a sequence of octets to domain participant, data readers and data writers. This data is then distributed by means of a built-in topic. This QoS policy is commonly used to distribute security credentials.

- The TOPIC_DATA QoS policy allows applications to associate a sequence of octet with a topic. This bootstrapping information is distributed by means of a built-in topic. A common use of this QoS policy is to extend topics with additional information, or meta-information, such as IDL type-codes or XML schemas.

- The GROUP_DATA QoS policy allows applications to associate a sequence of octets with publishers and subscribers–this bootstrapping information is distributed by means built-in topics. A typical use of this information is to allow additional application control over subscriptions matching.

These DDS configuration QoS policies provide useful a mechanism for bootstrapping and configuring applications that run in SoS. This mechanism is particularly relevant in SoS since it provides a fully distributed means of providing configuration information.

4.1.6 Setting QoS

All the code examples you have have seen so far did rely on default QoS settings, as such we did not have to be concerned with defining the desired QoS. Listing 4.1 shows how you can create and set QoS on DDS entities.

Listing 4.1: Setting QoS on DDS entities.

```
DomainParticipant dp(default_domain_id());
Topic<TempSensorType> topic("TempSensor");

PublisherQos pqos = dp.default_publisher_qos()
  << Partition("building-1:floor-2:room:3");

Publisher pub(dp,pqos);

DataWriterQos dwqos = dp.default_topic_qos()
     << Reliability::Reliable()
     << Durability::TransientLocal();

DataWriter<TempSensorType> dw(pub, drqos);
```

Appendix A

4.2 Examples Source Code

All the examples presented throughout the Tutorial are available online at `https://github.com/kydos/dds-tutorial-cpp`. The README.md provides all the information necessary to install and run the examples.

Chapter 5

Acronyms

DDS	The Data Distribution Service	3
DDSI	Data Distribution Service Interoperability Wire Protocol	5
DP	Domain Participant	8
GDS	Global Data Space	7
I2	Industrial Internet	5
IDL	Interface Definition Language	8
IoT	Internet of Things	5
JMS	the Java Message Service	7
OMG	Object Management Group	5
Pub/Sub	Publish/Subscribe	5
QoS	Quality of Service	8
SoS	Systems-of-Systems	37

Bibliography

[1] Object Management Group. Dynamic and extensible topic types, 2010.

[2] The Open Group. fnmatch API (1003.2-1992) section B.6, 1992.