The Data Distribution Service for Real-Time Systems

This is the first article of a series covering the Object Management Group (OMG) Data Distribution Service for Real-Time Systems (DDS). In this first installment I’ll be focusing on what DDS is, why it represents an important technology you should learn about, and how it is being used today. I will also make sure that those of you that can’t spend too much time away from a keyboard will get started coding their first DDS application.

What is DDS?

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). Pub/Sub is an abstraction for one-to-many communication that provides anonymous, decoupled, and asynchronous communication between a publisher and its subscribers. This is the abstraction behind many of the technologies used today to build and integrate distributed applications, such as social application, e-services, financial trading, etc., while maintaining their composing parts loosely coupled and independently evolvable.

Different implementations of the Pub/Sub abstraction have emerged for supporting the needs of different application domains. The Data Distribution Service for Real-Time Systems (DDS) is an Object Management Group (OMG) standard for Pub/Sub that addresses the needs of mission- and business critical applications, such as, financial trading, air traffic control and management, and complex supervisory and telemetry systems.

That key challenges addressed by DDS are to provide a Pub/Sub technology in which data exchange between producers and consumers are:

- **Real-time**, meaning that the right information is delivered at the right place at the right time— all the time. Failing to deliver key information within the required deadlines can lead to life-, mission- or business-threatening situations. For instance in a financial trading 1ms could make the difference between loosing or gaining $1M. Likewise, in a supervisory applications for power-grids failing to meet deadlines under an overload situation could lead to severe blackout such as the one experienced by the northeastern US and Canada in 2003 [1].

- **Dependable**, thus ensuring availability, reliability safety and integrity in spite of hardware and software failures. For instance, from the reliable functioning of an Air Traffic Control and Management System depends the life of thousands of people flying over the area it is managing. Thus these systems have to ensure 99.999% of availability and guarantee that regardless of experienced failures critical data is reliably delivered.

- **High-Performance**, hence able to distribute very high volumes of data with very low latencies. As an example, financial auto-trading applications have to deal with millions of messages per second each of which has to be delivered with minimal latency, in the order of tens of microseconds.

---

**Angelo Corsaro, Ph.D.**
Chief Technology Officer
PrismTech

Angelo is OpenSplice DDS Chief Technology Officer (CTO) at PrismTech. As CTO, Angelo directs the technology strategy, planning, evolution, and evangelism. Angelo leads the strategic standardization at the Object Management Group (OMG), where he co-chairs the Data Distribution Service (DDS) Special Interest Group. Angelo is a widely known and cited expert in the field of real-time and distributed systems, middleware, and software patterns, has authored several international standards and enjoys over 15+ years of experience in technology management and design of high performance mission- and business-critical distributed systems.

Angelo received a Ph.D. and a M.S. in Computer Science from the Washington University in St. Louis, and a Laurea Magna cum Laude in Computer Engineering from the University of Catania, Italy.

**Contacts:**
- angelo.corsaro@prismtech.com
- http://slideshare.net/angelo.corsaro
- https://twitter.com/acorsaro

---
At this point, to the question “What is DDS?” I can safely answer that it is a Pub/Sub technology for real-time, dependable and high performance data exchanges. Another way of answering as you will find in the course of this series is that DDS is Pub/Sub on Steroids.

The OMG DDS Standard

The OMG DDS standard family is today composed, as shown in Figure 1, by the DDS v1.2 API [2] and the DDS Interoperability Wire Protocol (DDSI v2.1) [3]. The DDS API standard guarantees source code portability across different vendor implementations, while the DDSI Standard ensures on the wire interoperability across DDS implementations from different vendors. The DDS API standard defines several different profiles (see Figure 1) that enhance real-time pub/sub with content filtering, persistence, automatic fail-over, and transparent integration into object oriented languages.

At this point, to the question “What is DDS?” I can safely answer that it is a Pub/Sub technology for real-time, dependable and high performance data exchanges. Another way of answering as you will find in the course of this series is that DDS is Pub/Sub on Steroids.

The OMG DDS Standard

The OMG DDS standard family is today composed, as shown in Figure 1, by the DDS v1.2 API [2] and the DDS Interoperability Wire Protocol (DDSI v2.1) [3]. The DDS API standard guarantees source code portability across different vendor implementations, while the DDSI Standard ensures on the wire interoperability across DDS implementations from different vendors. The DDS API standard defines several different profiles (see Figure 1) that enhance real-time pub/sub with content filtering, persistence, automatic fail-over, and transparent integration into object oriented languages.

![Figure 1. The DDS Standard](image)

The DDS standard was formally adopted by the OMG in 2004. In less than 6 years, it has become the established Pub/Sub technology for distributing high volumes of data, dependably and with predictable low latencies in applications such as, Radar Processors, Flying and Land Drones, Combat Management Systems, Air Traffic Control and Management, High Performance Telemetry, Large Scale Supervisory Systems, and Automated Stocks and Options Trading. Along with wide commercial adoption, the DDS standard has been recommended and mandated as the technology for real-time data distribution by several key administrations worldwide, such as the US Navy, the DoD Information-Technology Standards Registry (DISR) the UK MoD, the MILVA Vehicles Association, and EUROCONTROL--the European organization for the safety of air navigation.

Now that I’ve given you an overall idea of what DDS is and where it is being used let’s try to see how it works.
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 in this article and then use it to explain the various DDS features throughout the series. To ensure that you can experiment with what I'll be presenting in the various article all examples will be based on OpenSplice DDS [4], an Open Source DDS implementation, and will use the upcoming ISO C++ DDS API available via the SimD (Simple Dds) [5] Open Source project. The source code for the examples will also be available as part of the SimD distribution.

The example that I will use is the temperature monitoring and control system for a very large building, where 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 of the various rooms.

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.

 DDS vs Other Pub/Sub

Whenever you hear about a new technology you may say "Cool!" or "So What?!". If you are a "So What?!" kind of person, this sidebar quickly summarizes how DDS differs from other publish/subscribe technologies. We'll compare with the Java Message Service (JMS) and Advanced Messaging Queuing Protocol (AMQP).

One way of comparing DDS to JMS and AMQP is to measure what they standardize. JMS standardizes an API while AMQP standardizes a wire-protocol -- DDS standardizes both.

Another important characteristic is the architectural style promoted. DDS is fully distributed and does not require the presence of any broker mediating communication between producers and consumers. On the other hand, both JMS and AMQP have a broker-based architecture where one or more brokers mediate the distribution of information from sources to destinations.

If you look at the expressiveness of these different technologies, you will see that DDS provides the ability of associating user defined data types with topics and perform full queries as well as continuous queries on these data types. This exceeds what both JMS and AMQP can provide since filtering is limited to message headers and topics tend to hide the user data type into more or less structured blobs.

Another dimension in which DDS excels when compared to either JMS or AMQP is the support for QoS. DDS allows you to configure just about any non-functional aspect of a distributed pub/sub application while JMS and AMQP provide a subset of QoS with often a subset of possible choices.

Finally, if we look at performance, DDS implementations can deliver very low latencies (in the range of 50-60 usec on a Gbps Ethernet) along with very high throughput (millions update per second) while retaining very low and deterministic jitter.

The key abstraction at the foundation of DDS is a fully distributed Global Data Space (GDS). It is important to remark that the DDS specification requires the implementation of the Global Data Space to be fully distributed to avoid the introduction of single point of failure or single point of bottleneck. Publishers and Subscribers can join or leave the 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 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 dynamic 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.

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

Topics

Topic Types can be represented with the subset of the OMG IDL [6] 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 provides an example of how you might define the topic type for the temperature sensor:

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

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

#pragma keylist TempSensorType id
```

Listing 1. IDL Definition for the TempSensorType

As Listing 1 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 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 ID (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 programatically register a DDS topic using the DDS API by simply instantiating a `Topic` class with proper type and name.

```cpp
dds::Topic<TempSensorType> tsTopic("TempSensorTopic");
```

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 `DataReader` and `DataWriter`

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 2 and 3 show the steps required to do so.

If you look a bit closer to Listing 3, 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, which are 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 series.

```cpp
Listing 2. Writing Data

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 article DDS is equipped with dynamic discovery of both participants as well as user-defined data types. Thus it is DDS that discovers data producers and consumers and takes care of matching them. My strongest recommendation is that you try
to compile the code examples available with SimD under demo/ddj-series/01
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 3. Reading Data

Concluding Remarks
In this first article of the DDS series I’ve explained the abstraction behind DDS,
and introduced some of its foundational 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 less than 15 lines of
code in total, remarkable, isn’t it? In the upcoming articles 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 real-time
publish/subscribe applications.

References
2. OMG. Data Distribution Service for Real-Time Systems Specification. DDS
   v1.2 <http://www.omg.org/spec/DDS/1.2/>
3. OMG. Data Distribution Service Interoperability Wire-Protocol
   Specification. DDSI v2.1 <http://www.omg.org/spec/DDSI/2.1/>
4. OpenSplice DDS <http://www.opensplice.org>
   code.google.com/p/simd-cxx/>
   www.omg.org/spec/CORBA/3.1/>