The DDS Tutorial

Angelo Corsaro, Ph.D.

1.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.

1.1. What is DDS?

Wether 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.

At this point, to the question "What is DDS?" I can safely answer that it is 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.

1.1. The OMG DDS Standard

The OMG DDS standards 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.

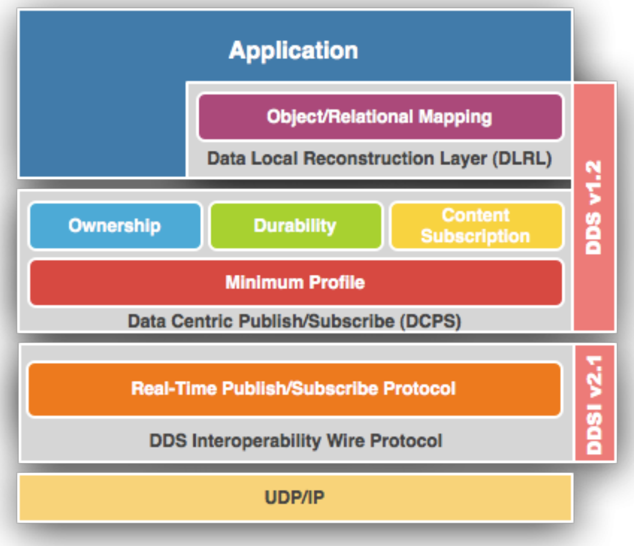

Application
Object/Relational Mapping
Data Local Reconstruction Layer (DLRL)
Ownership
Durability
Content Subscription
Minimum Profile
Data Centric Publish/Subscribe (DCPS)
DDS v1.2
Real-Time Publish/Subscribe Protocol
DDS Interoperability Wire Protocol
DDSI v2.1
UDP/IP

Figure 1. The DDS Standard

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.

1.2. 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.

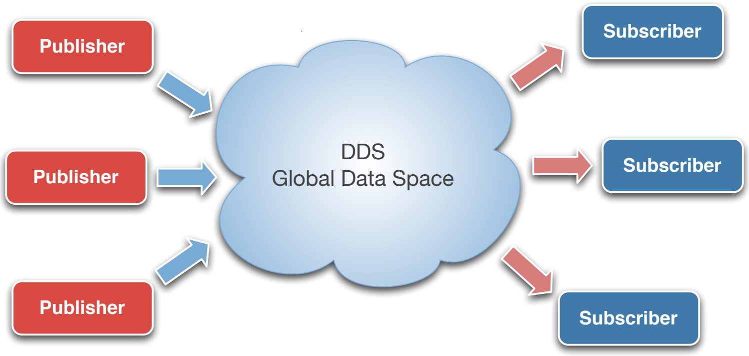


Figure 2. The DDS Global Data Space

1.2.1. Global Data Space

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.

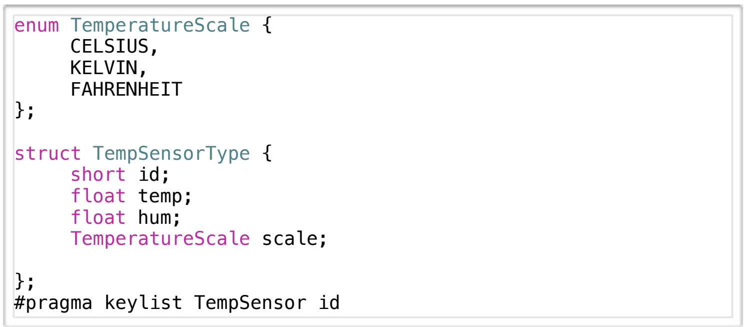
In essence, the presence of a GDS equipped with dynamic discovery means that when you deploy a system, you don't have to configure anything. Everything will be automatically discovered and data will begin to flow. Moreover, since the GDS is fully distributed you don't have to fear the crash of some server 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.

1.2.2. Topics

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.

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 {
     CELSIUS,
     KELVIN,
     FAHRENHEIT
};

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

};
#pragma keylist TempSensor 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 programmatically register a DDS topic using the DDS API by simply instantiating a `Topic` class with proper type and name.

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

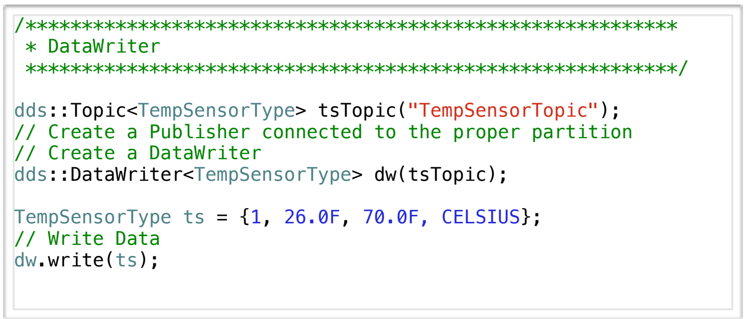
1.2.3. Reading and Writing Data

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 2 and 3 show the steps required to do so.

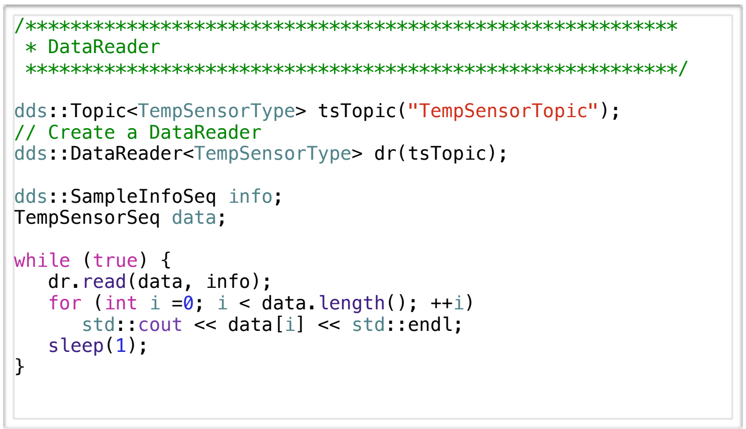
If you look a bit closer to Listing 3, you'll see that our first DDS application is using polling to read data out of DDS every second. A `sleep` is used to avoid spinning in the loop to fast since the DDS `read` is non-blocking and returns right away if there is no data available. Although polling is 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.



```
/***********************************************************
 * DataWriter
 ***********************************************************/

dds::Topic<TempSensorType> tsTopic("TempSensorTopic");
// Create a Publisher connected to the proper partition
// Create a DataWriter
dds::DataWriter<TempSensorType> dw(tsTopic);

TempSensorType ts = {1, 26.0F, 70.0F, CELSIUS};
// Write Data
dw.write(ts);
```

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 produces 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.



```
/***********************************************************
 * DataReader
 ***********************************************************/

dds::Topic<TempSensorType> tsTopic("TempSensorTopic");
// Create a DataReader
dds::DataReader<TempSensorType> dr(tsTopic);

dds::SampleInfoSeq info;
TempSensorSeq data;

while (true) {
   dr.read(data, info);
   for (int i =0; i < data.length(); ++i)
      std::cout << data[i] << std::endl;
   sleep(1);
}
```

Listing 3. Reading Data

1.3. 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

[1] http://en.wikipedia.org/wiki/Northeast_Blackout_of_2003

[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>

[5] New ISO C++ DDS API Reference Implementation. <http://code.google.com/p/simd-cxx/>

[6] Common Object Request Broker Architecture (CORBA/IIOP) v3.1. <http://www.omg.org/spec/CORBA/3.1/>

2. Topics, Domains and Partitions

This is the second installment [1] of a series covering the Object Management Group (OMG) Data Distribution Service for Real-Time Systems (DDS). In the first installment 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. DDS 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. Leaving out the QoS, for the time being, let's explore what is so special about them.

2.1.1. Topic Types

DDS is a programming language and OS independent publish/subscribe technology. To ensure independence from a specific programming language and OS, while guaranteeing portability and interoperability, it uses a subset of IDL [2, 4] as the formalism for describing topic types and CDR [2,3] for encoding them. IDL is a standard syntax for expressing various kind of types while retaining independence from a specific programming language. CDR is a binary serial encoding providing a good trade-off between space and time efficiency.

Since CDR is used under the hood by DDS and not visible to the user, I will be focusing next on the subset of IDL that can be legally 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 1), a template type (see Table 2), or a constructed type (see Table 3).

Table 1. IDL Primitive Types

Primitive Types	
`boolean`	`long`
`octet`	`unsigned long`
`char`	`long long`
`wchar`	`unsigned long long`
`short`	`float`
`unsigned short`	`double`
	`long double`

As shown in Table 1, 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 3. IDL template types.

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 shows IDL templates types. The `string` and `wstring` can be parametrized only with respect to their maximum length; the `sequence` type with respect 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 3. IDL constructed types.

Constructed Types	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;};`

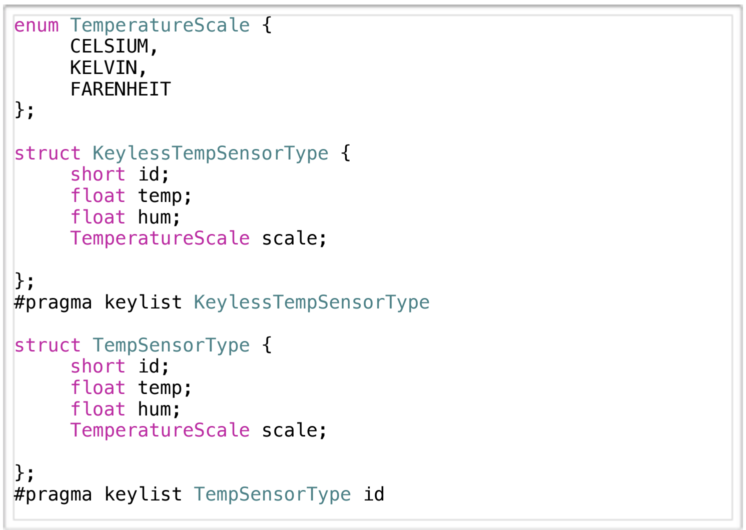
`union`	`union Coord switch (Dimension) {`
	`  case 1D:`
	`     Coord1D c1d;`
	`  case 2D:`
	`     Coord2D c2d;`
	`  case 3D:`
	`     Coord3D c3d;`
	`  case 4D:`
	`     Coord4D c4d;`
	`};`

Table 3 shows that DDS supports three different kind 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.



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

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

};
#pragma keylist KeylessTempSensorType

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

};
#pragma keylist TempSensorType id
```

Listing 1. Keyless and Keyed Topic Type

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 the first installment. Listing 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 1 what would be the exact difference between them?

```
dds::Topic<KeylessTempSensorType> kltsTopic(“KLTempSensorTopic”);
dds::Topic<TempSensorType> tsTopic(“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 wether (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 1.

```
dds::DataWriter<KeylessTempSensorType> kltsdw(kltsTopic);
TempSensorType ts = {1, 26.0F, 70.0F, CELSIUS};
secdw.write(ts);
ts = {2, 26.0F, 70.0F, CELSIUS};
secdw.write(ts);
```

These two samples will be posted in the same reader queue; the queue associated with the singleton instance, as shown in Figure 1.

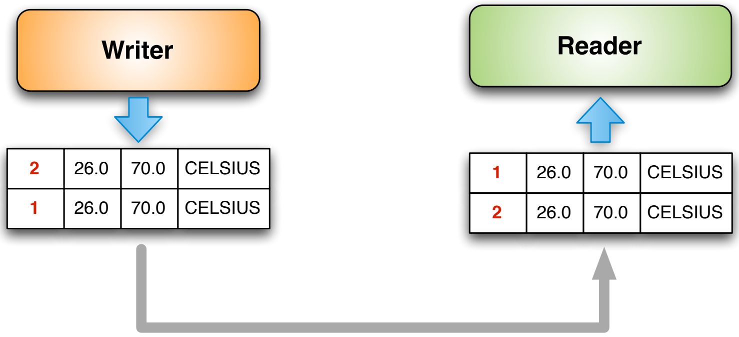

Writer
Reader
2 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
2 | 26.0 | 70.0 | CELSIUS

Figure 1. Single reader queue associated to a Keyless Topics.

If we write the 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.

```
dds::DataWriter<EventCountType>  ecdw(ecTopic);
TempSensorType ts = {1, 26.0F, 70.0F, CELSIUS};
secdw.write(ts);
ts = {2, 26.0F, 70.0F, CELSIUS};
secdw.write(ts);
```

In this case the reader will see these two samples posted into two different queues, as represented in Figure 2, one queue for each instance.

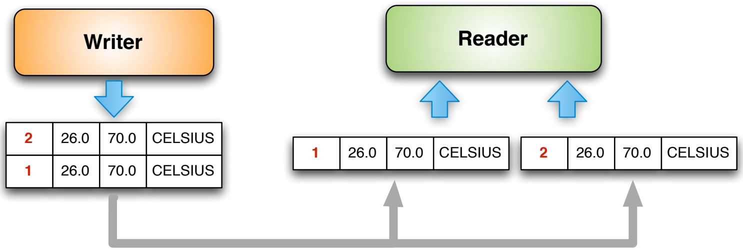

Writer
Reader
2 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
1 | 26.0 | 70.0 | CELSIUS
2 | 26.0 | 70.0 | CELSIUS

Figure 2. Multiple reader-queues associated with each instance of a keyed topic

In 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 exploit 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.

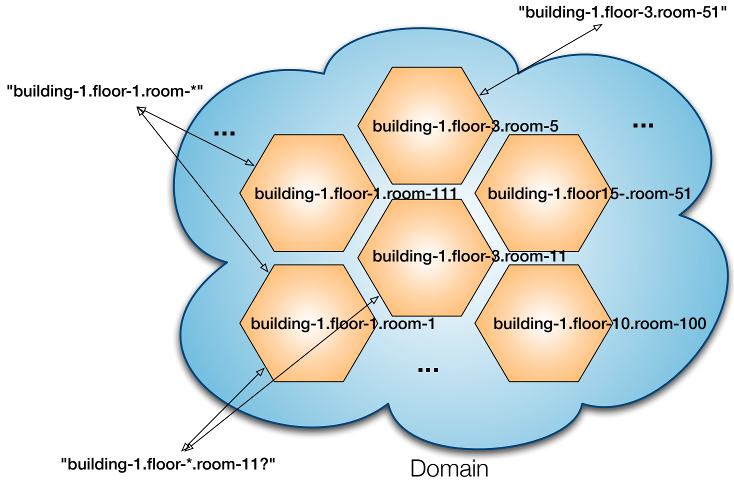

"building-1.floor-3.room-51"
"building-1.floor-1.room-*"
...
building-1.floor-3.room-5
...
building-1.floor-1.room-111
building-1.floor15-.room-51
building-1.floor-3.room-11
building-1.floor-1.room-1
building-1.floor-10.room-100
...
"building-1.floor-*.room-11?"
Domain

Figure 3. Domain, partitions, and partition matching.

2.2. Mechanism and Techniques for Scoping Information

2.2.1. Domains

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. Partitions

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 [5] function.

To recap, partitions provide a way for 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, and for which once can come up with natural segregation.

If we take as an example our temperature monitoring and control system, then we can come up with a very natural partitioning of data that mimics the physical 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 3, all the topics produced in room 51 on the 15th floor on building 1 would be 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-Filtered Topics

Domains and Partitions are a very good way of organizing data. Yet, they operate at structural level. What if you need to control the data received based on its actual content? DDS Content-Filtered Topics allow 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 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 3.

Table 4. Legal operators for DDS Filters and Query Conditions

Operator	Description
=	Equal
<>	Not Equal
>	Greater Than
<	Less Than
>=	Greater then equal
<=	Less than equal
BETWEEN	Between and inclusive range
LIKE	Search for a pattern

Content-Filtered topics are very useful from several different perspective. First of all they limit the amount of memory used by the middleware 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 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.

```
// Create the TempSensor topic
dds::Topic<TempSensorType> tsTopic("TempSensor");

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

// Define the filter parameters (taking some C++0x liberty)
std::vector<std::string> params =
   {"20.5", "21.5", "30", "50"};

// Create the ContentFilteredTopic
dds::ContentFilteredTopic<TempSensorType>
   cfTsTopic("CFTempSensor",
             tsTopic,
             filter_expression,
             params);


// Create the DataReader
dds::DataReader<TempSensorType> dr(cfTsTopic);

dds::SampleInfoSeq info;
TempSensorSeq data;

while (true) {
   // The only data received will satisfy the filter
   // condition of the ContentFilteredTopic
   dr.read(data, info);
   for (int i =0; i < data.length(); ++i)
      std::cout << data[i] << std::endl;
   sleep(1);
}
```

Listing 2. Creating a content-filtered topic.

2.4. Query Conditions

For completeness I should mention that DDS also supports Query Conditions . The main difference between a Query Condition and a Content-Filtered Topics is that the former does not filter incoming data, it simply queries data received and available on an existing reader cache. As a result the execution of the Query Condition is completely under the user control and executed in the context of a read as shown in Listing 3. The syntax supported by Query Expression is identical to that used to define filter expression and listed in Table 4.

```
// Create the TempSensor topic
dds::Topic<TempSensorType> tsTopic("TempSensor");

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

// Define the filter parameters (taking some C++0x liberty)
std::vector<std::string> params =
   {"20.5", "21.5", "30", "50"};

// Create Query Condition
dds::QueryCondition cond =
   dr.create_querycondition(filter_expression, params);

TempSensorTypeSeq data;
SampleInfoSeq info;
// Read with Condition
dr.read(data, info, cond);
```

Listing 3. Using Query Conditions to read data

2.5. Concluding Remarks

In this second installment of the DDS Series 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 available with SimD [7] under demo/ddj-series/02 and then try to experiment a bit by yourself.

References

[1] The Data Distribution Service for Real-Time Systems: Part-1 <http://www.drdobbs.com/architect/222900238>

[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] Common Object Request Broker Architecture (CORBA/IIOP) v3.1. <http://www.omg.org/spec/CORBA/3.1/

[5] POSIX fnmatch API (1003.2-1992 section B.6)

[6] OpenSplice DDS <http://www.opensplice.org>

[7] SimD. <http://code.google.com/p/simd-cxx/>

3. Reading and Writing Data

Last month I covered in great details 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 installment 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 installment of the series, 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 writes 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 in the ALIVE state if there is at least one `DataWriter` writing it. An instance is in the NOT_ALIVE_NO_WRITERS state when there are no more `DataWriter`s writing it. Finally, the instance is in the NOT_ALIVE_DISPOSED state 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 I, 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, or 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.

```
int main(int, char**) {
  dds::Topic<TempSensorType> tsTopic("TempSensorTopic");
  dds::DataWriter<TempSensorType> dw(tsTopic);

  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);

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

Listing 1 - Automatic management of Instance life-cycle.

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 is 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++0x **`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 the last installment of the series, 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.

```
int main(int, char**) {
   dds::Runtime runtime;

   dds::Topic<TempSensorType> topic("TempSensorTopic");

   //[NOTE #1]: Avoid topic-instance dispose on unregister
   dds::DataWriterQos dwqos;
   dwqos.set_auto_dispose(false);

   //[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]: Using C++0x “auto” to keep code more compact
   //[NOTE #4] Registering topic-instance explicitly
   auto diw1 = dw.register_instance(key);

   key.id = 2;
   auto diw2 = dw.register_instance(key);

   key.id = 3;
   auto diw3 = dw.register_instance(key);

   data = {1, 24.3F, 0.5F, CELSIUS};
   diw1.write(data);

   data = {2, 23.5F, 0.6F, CELSIUS};
   diw2.write(data);

   data = {3, 21.7F, 0.5F, CELSIUS};
   diw3.write(data);

   // [NOTE #5]: unregister topic-instance with id=1
   diw1.unregister();

   // [NOTE #6]: dispose topic-instance with id=2
   diw2.dispose();


   //[NOTE #7]:topic-instance with id=3 will be unregistered
   // as result of the diw3 object destruction

   //[NOTE #8]: topic instance with id=0 will be unregistered as
   // result of the dw object destruction
   return 0;
}
```

Listing 2 - Explicit management of topic-instances life-cycle

3.1.5. Blocking or Non-Blocking Write?

One question that you might have at this point is wether the `write` is blocking or not. The short answer is that the `write` is non-blocking, however, as we will see in the next installment there are some cases in which, depending on QoS settings, that the `write` might block to avoid data-loss.

3.2. Accessing Data

DDS provides two different mechanisms to access data, each of which allowing to control what data is accessed and to be informed about data availability. Let's see in details what does this boils down to.

3.2.1. Read vs. Take

The DDS provides data access via 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 the data received by the `DataReader` without removing it from its local cache. This means that this data will be again readable via an appropriate `read` call. Likewise, the DDS provides a take semantics, implemented by the `DataReader::take` methods, that 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.

3.2.2. Data and Meta-Data

In the first part of this article 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, to each data sample received by a `DataWriter` is associated a structure, called `SampleInfo` describing the property of that sample. These properties includes information on:

- **Sample State.** The sample state can be READ or NOT_REAT 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 wether data are read or taken away from DDS, the same mechanism is used to specify the sample properties thus, for brevity sake, I am going to provide some examples using the `read` yet if you want to use the `take` semantics you simply have to replace each occurrence of a `read` with a `take`. Specifically, DDS allows you to select data based on values of the view state, instance state and sample state. 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:

```
dr.read(data, info, ANY_SAMLE_STATE, ANY_VIEW_STATE,
                    ANY_INSTANCE_STATE);
```

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:

```
dr.read(data, info, NOT_READ_SAMLE_STATE, ANY_VIEW_STATE,
                    ANY_INSTANCE_STATE);
```

If I wanted to ensure that I would always and only read new valid data (meaning no samples with only a valid SampleInfo) I would issue a read (or take) as follows:

```
dr.read(data, info, NOT_READ_SAMLE_STATE, ANY_VIEW_STATE,
                    ALIVE_INSTANCE_STATE);
```

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:

```
   dr.read(data, info, NOT_READ_SAMLE_STATE, NEW_VIEW_STATE,
                       ALIVE_INSTANCE_STATE);
```

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:

```
 dr.read(data, info);
```

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

As a final remark, I wanted to make sure that you don't confuse these statues with the read condition that we explored in the last article. Read condition are used to query data with respect to its content. The statuses on the other hand are used to select data with respect to its meta-information.

3.2.4. Iterators or Containers?

The examples I showed above intentionally avoided to explicit the kind of structures used for the data and info parameters. If you were attentive, when reading the earlier articles, you might recalled that I used containers, yet, that's not the only option. SimD as well as the new ISO C++ API for DDS provides you with container-based as well as iterator-based reads and takes. For high performance applications it also provides you with a zero-copy API.

The container-based read/take API relies on `std::vector` for both data and sample information. The size of the vector passed to the read (or take) is used to decide the maximum number of samples that should be read. If the size is non-zero, at most a number of samples equal to the size of the container will be read. If the size is zero, then the container will be resized to hold all available samples. Thus, the following code will read all available samples:

```
   std::vector<TempSensorType> data;
   std::vector<SamplInfo> info;
   read(data, info);
```

While this, would only read at most 10 samples:

```
   std::vector<TempSensorType> data(10);
   std::vector<SamplInfo> info(10);
   read(data, info);
```

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:

```
   TempSensorType* data = ...; // Get hold of some memory
   SampleInfo* info = ...; // Get hold of some memory
   uint32_t samples = dr.read(data, info, 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 **DataReader**s. This functionality is provided by DDS via the **Waitset** class which can be seen as an object oriented version of the Unix **select**.

```
class TempSensorDataHandler {
public:
   void operator() (dds::DataReader<TempSensorType>& reader) {
      std::vector<TempSensorType> data;
      std::vector<dds::SampleInfo>     info;
      reader.read(data, info);
      //[NOTE #1] Do whatever is needed with the data
   }
};
```

Listing 3 - A sample DDS condition handler.

```
dds::DataReader<TempSensorType> dr(topic);

//[NOTE #1]: Create an instance of the handler
TempSensorDataHandler handler;

//[NOTE #2]: Create a read condition for the given reader
auto rcond = dr.create_readcondition(handler);

//[NOTE #3]: Create a Waitset and attach the condition
dds::WaitSet ws;
ws.attach(rcond);

dds::Duration timeout = {1, 0};


//[NOTE #4] Wait for some condition to become active,
// and retrieve all active conditions
auto conds = ws.wait(timeout);

//[NOTE #5] Execute the condition handlers
for (auto i = conds.begin(); i < conds.end(); ++i)
i->execute();


//[NOTE #5] Automatically dispatch the condition handlers when
//conditions are notified
ws.dispatch(timeout);
```

Listing 4 - Waitset in actions.

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 3. Then as shown in Listing 4, 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 5 and 6 shows how this can be done.

```
class TempSensorDataHandler {
public:
   void handle_data(dds::DataReader<TempSensorType>&
reader) {
      std::cout << "Reading..." << std::endl;
      std::vector<TempSensorType> data;
      std::vector<dds::SampleInfo>     info;
      reader.read(data, info);
      //[NOTE #1] Do whatever is needed with the data
   }
};
```

Listing 5 - on_data_available Event Handler

```
dds::DataReader<TempSensorType> dr(topic);

//[NOTE #1]: Create an instance of the handler
TempSensorDataHandler handler;


auto func =
   boost::bind(TempSensorDataHandler::handle_data,
               &handler, _1);

//[NOTE #2]: Register the handler for the relevant event
auto connection =
   dr.connect<on_data_available>(func);
```

Listing 6 - Registering an handler with the on_data_available event

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. Concluding Remarks

In this article I have presented in good details 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 available with SimD [7] under demo/ddj-series/03 and then try to experiment a bit by yourself.

Finally, I'd like to dedicate this installment to Jean-Claude MAHIEUX, who along with being PrismTech Chief Operating Officer, and an excellent computer scientist, was kind soul and a very close friends for me and all of us who worked very closely with him. Jean-Claude left us with surprise while still young and full of energy. His smile will live forever in our hearts and through what we do every day.

References

[1] The Data Distribution Service for Real-Time Systems <http://www.drdobbs.com/architect/222900238>

[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>

[5] SimD. <http://code.google.com/p/simd-cxx/>

The Data Distribution Service for Real-Time Systems

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

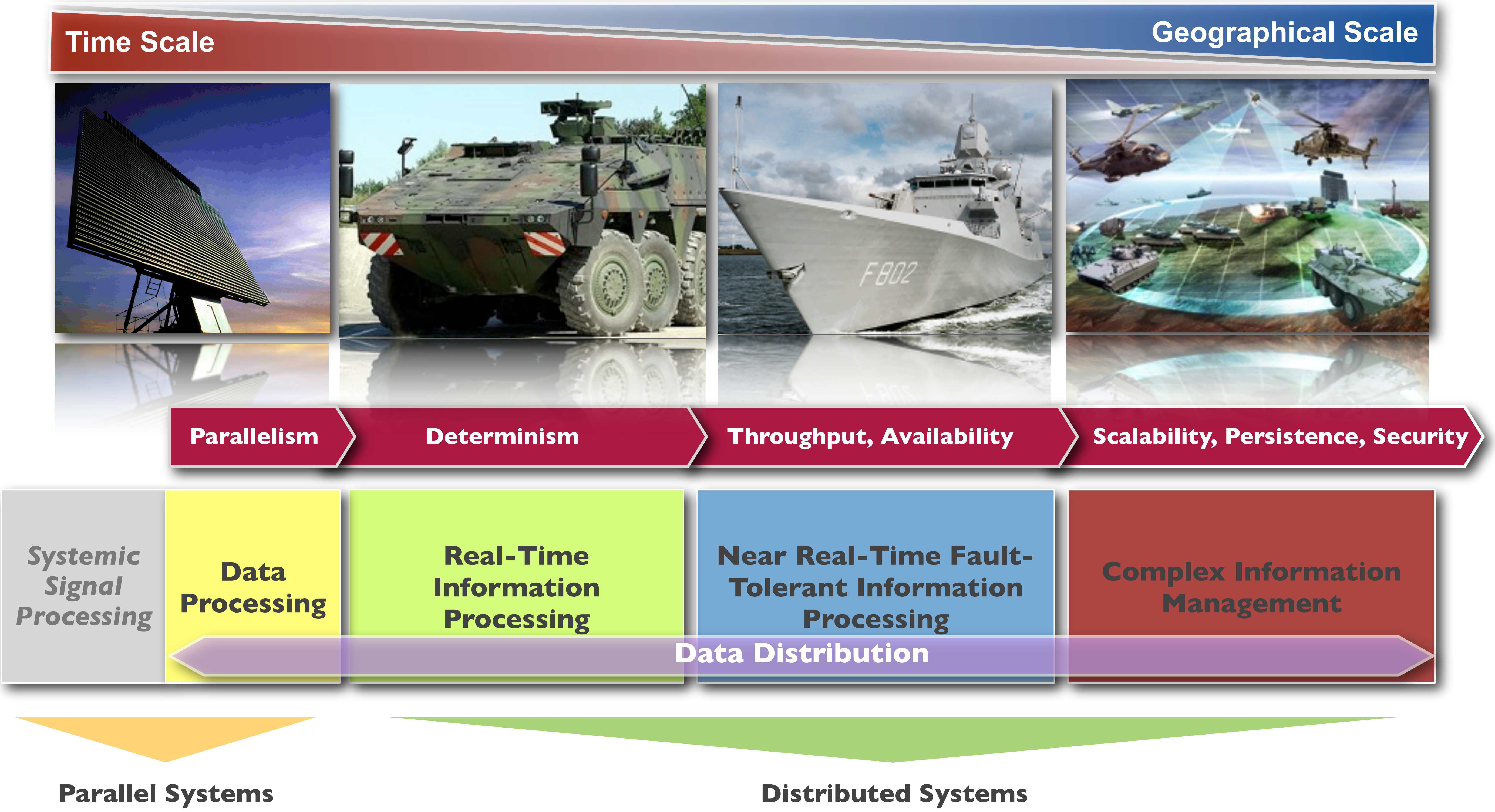
Angelo Corsaro, Ph.D.
Chief Technology Officer
PrismTech
OMG DDS SIG Co-Chair
angelo.corsaro@prismtech.com

OpenSplice|DDS

Delivering Performance, Openness, and Freedom

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

...Historical Perspectives



Time Scale

Geographical Scale

Parallelism

Determinism

Throughput, Availability

Scalability, Persistence, Security

Systemic Signal Processing

Data Processing

Real-Time Information Processing

Near Real-Time Fault-Tolerant Information Processing

Complex Information Management

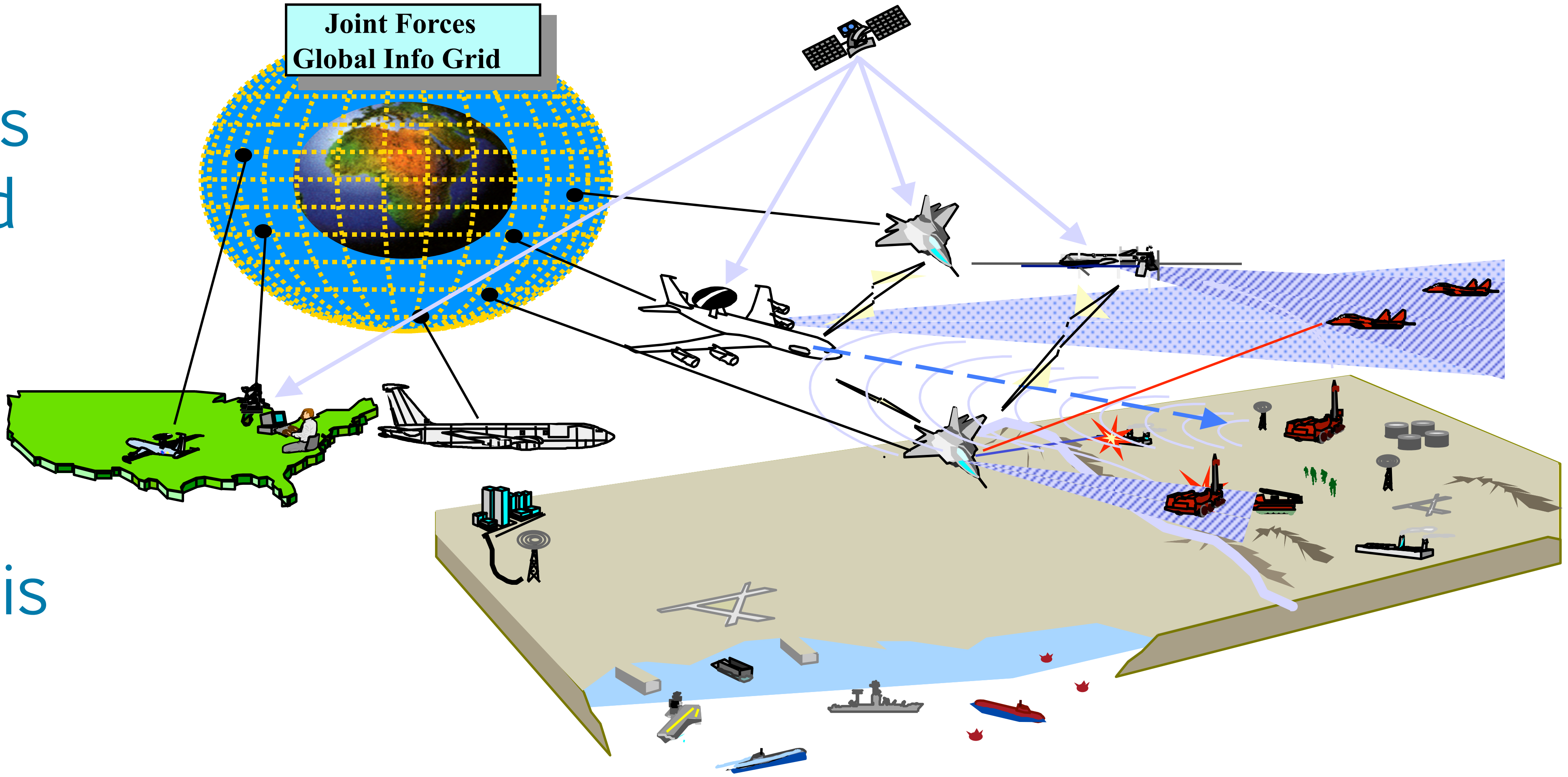
Data Distribution

Parallel Systems

Distributed Systems

The Need for Data Distribution

- Network Centric Architectures are emerging as a key trend for next generation military and civil system of systems
- Efficient, scalable and QoS-enabled data dissemination is an enabling technology for Network Centric Systems



*The Right **Information** => To the Right **People** => At the Right **Time***

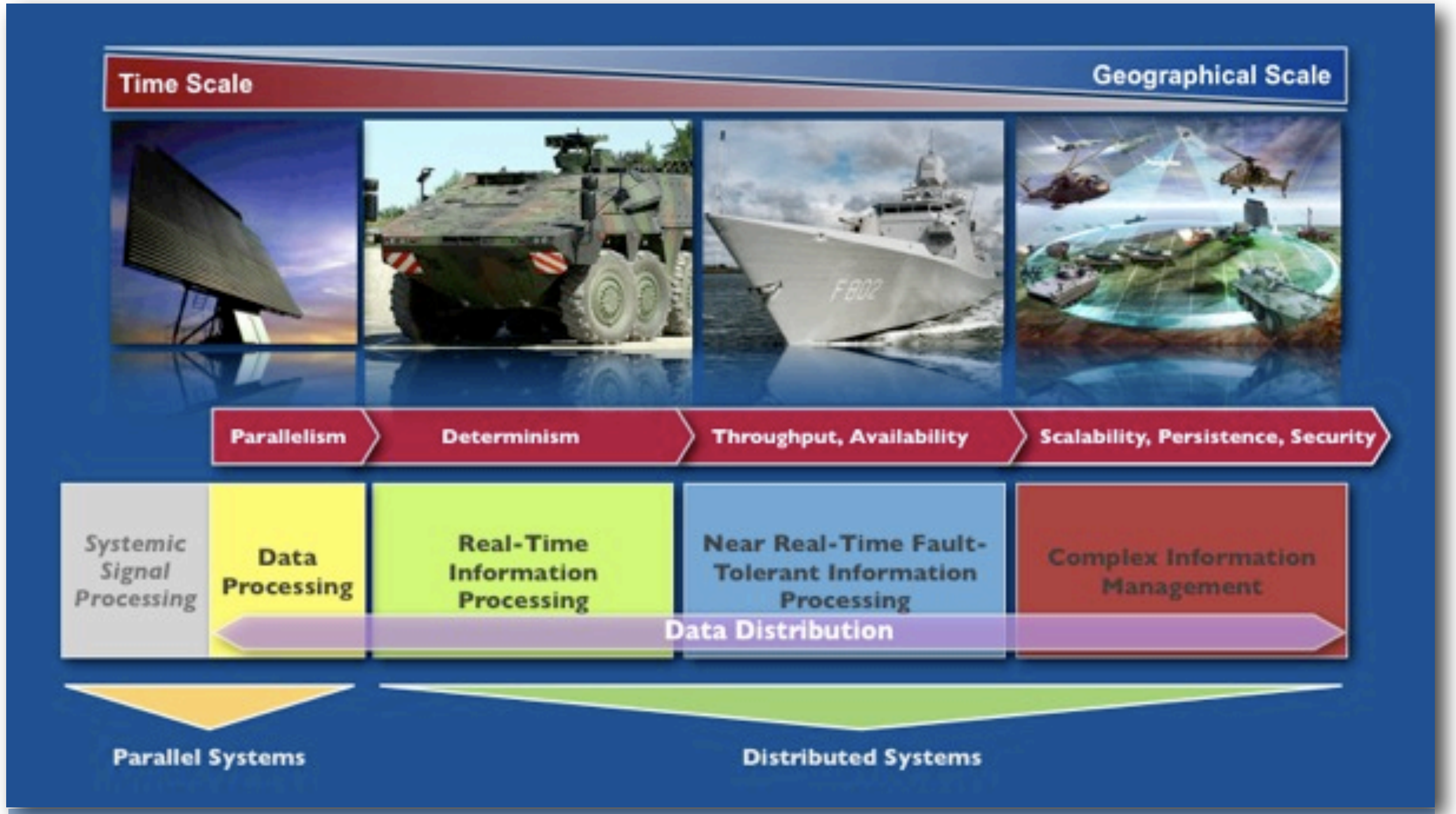
The OMG DDS

The OMG DDS Standard

- Introduced in 2004 to **address the Data distribution challenges** typical of **Defense and Aerospace Applications**

- Key requirement for the standard were **high performance** and **scalability** from **embedded to ultra-large-scale deployments**

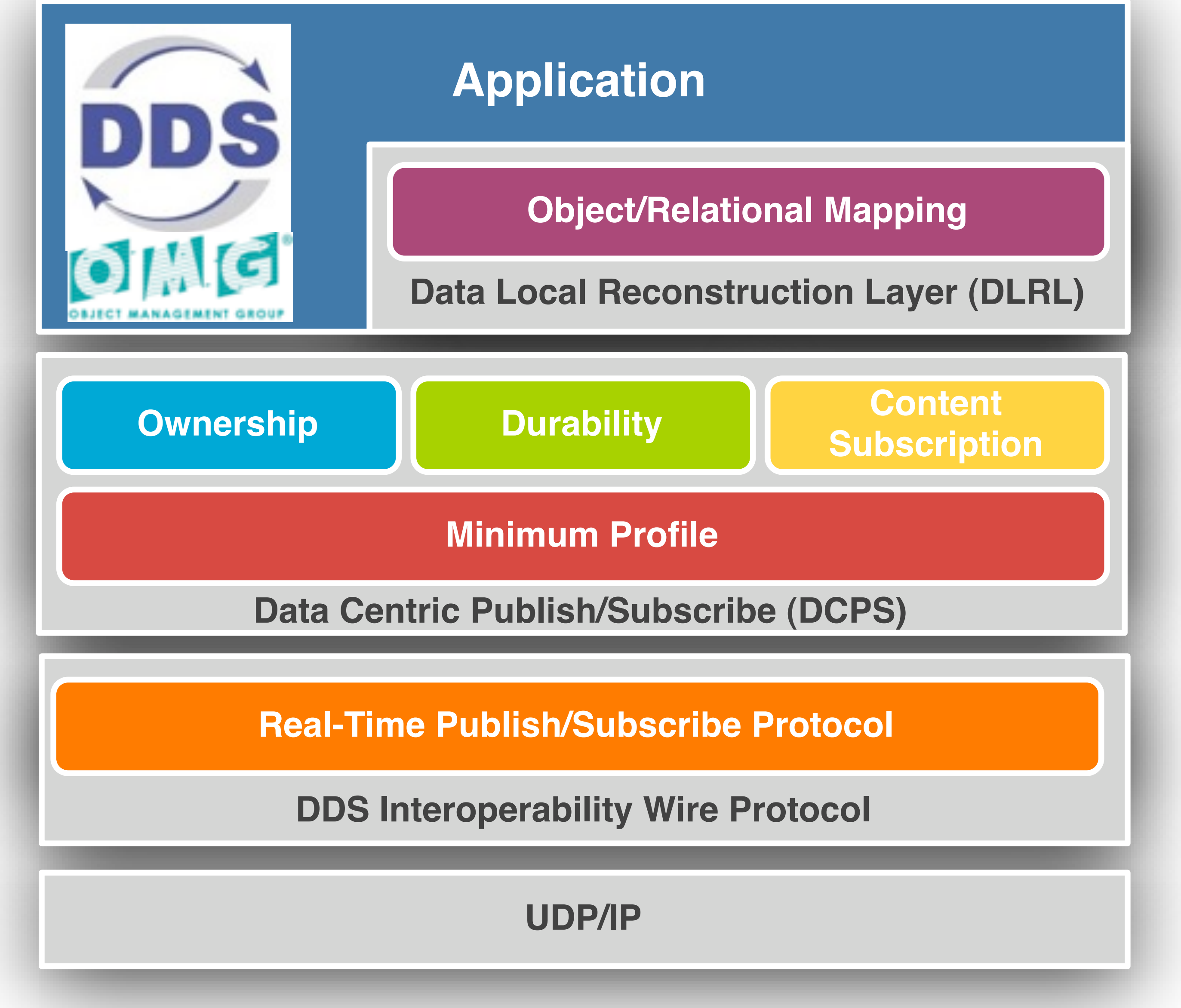
- Today **recommended** by **key administration worldwide** and **widely adopted** well beyond Aerospace and Defense in domains, such as, Automated Trading, Simulations, SCADA, Telemetry, etc.


Time Scale
Geographical Scale
Parallelism
Determinism
Throughput, Availability
Scalability, Persistence, Security
Systemic Signal Processing
Data Processing
Real-Time Information Processing
Near Real-Time Fault-Tolerant Information Processing
Complex Information Management
Data Distribution
Parallel Systems
Distributed Systems

The OMG Data Distribution Service

DDS v1.2 API Standard

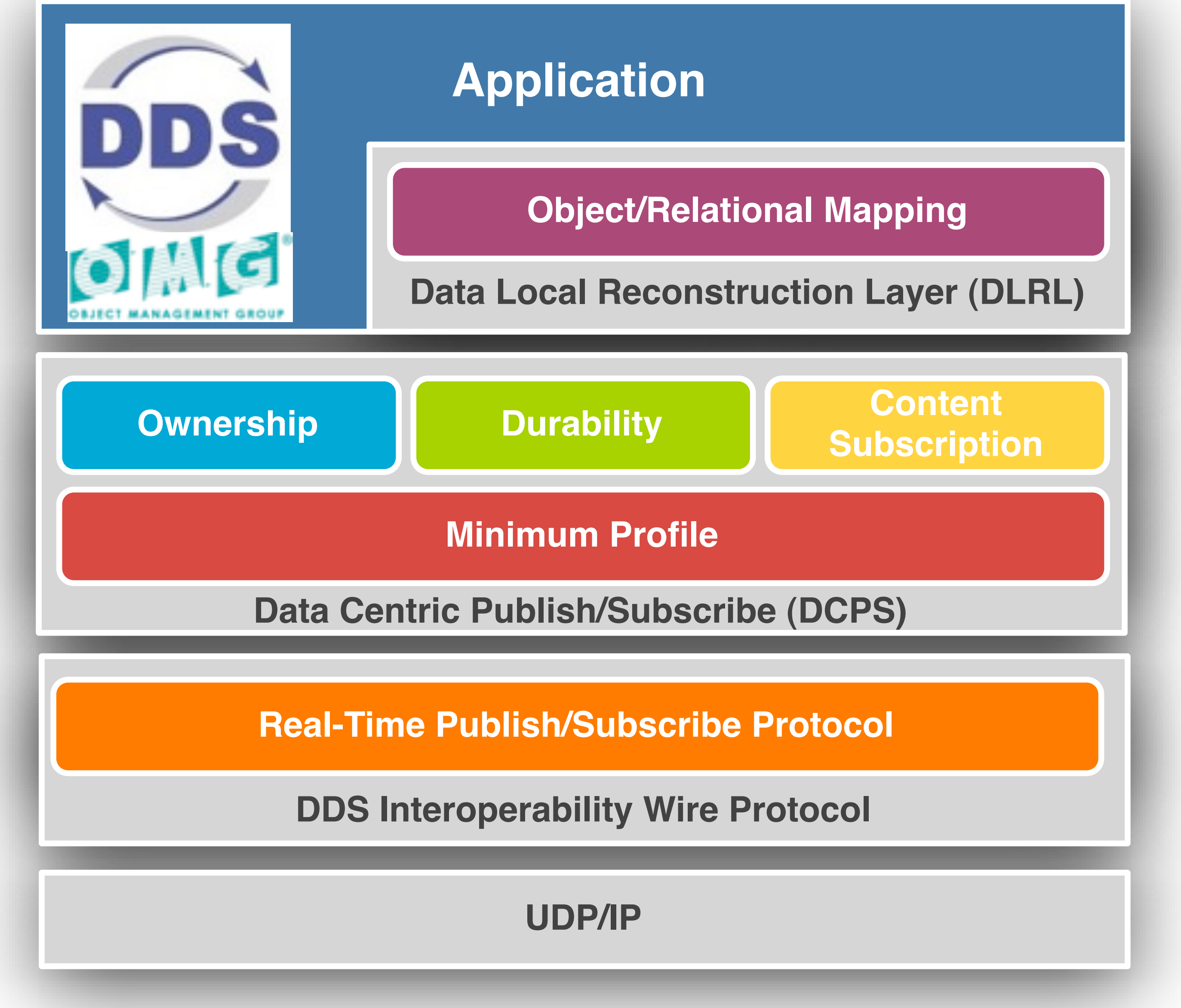
- Language Independent, OS and HW architecture independent
- **DCPS**. Standard API for Data-Centric, Topic-Based, Real-Time Publish/Subscribe
- **DLRL**. Standard API for creating Object Views out of collection of Topics


Application
Object/Relational Mapping
Data Local Reconstruction Layer (DLRL)
Ownership
Durability
Content Subscription
Minimum Profile
Data Centric Publish/Subscribe (DCPS)
Real-Time Publish/Subscribe Protocol
DDS Interoperability Wire Protocol
UDP/IP

The OMG Data Distribution Service

DDSI/RTPS v2.1 Wire Protocol Standard

- Standard wire protocol allowing interoperability between different implementations of the DDS standard
- Interoperability demonstrated among key DDS vendors in March 2009


Application
Object/Relational Mapping
Data Local Reconstruction Layer (DLRL)
Ownership
Durability
Content Subscription
Minimum Profile
Data Centric Publish/Subscribe (DCPS)
Real-Time Publish/Subscribe Protocol
DDS Interoperability Wire Protocol
UDP/IP

DDS Recommendations

Increasingly Mandated/Recommended by Administrations

- **US Navy**: Open Architecture
- **DISR/DISA**: Net-centric Systems
- **EuroControl**: Air Traffic Control Center Operational Interoperability
- **QinetiQ & MILVA**: Recommending DDS for Vehicles Electronic Architecture

OpenSplice | DDS

Delivering Performance, Openness, and Freedom

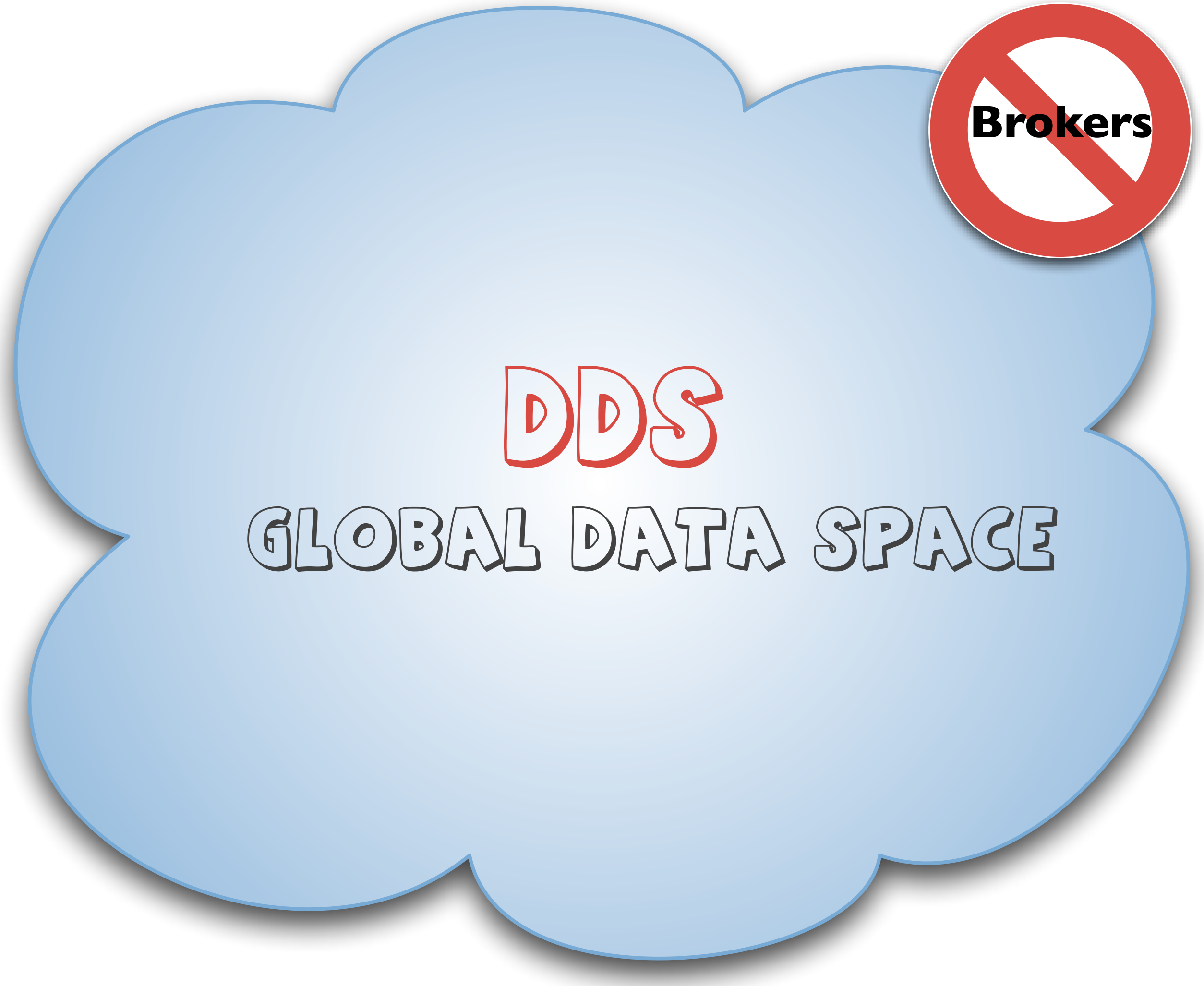
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

HOW DOES iT WORKS?!?

How Does it Work?

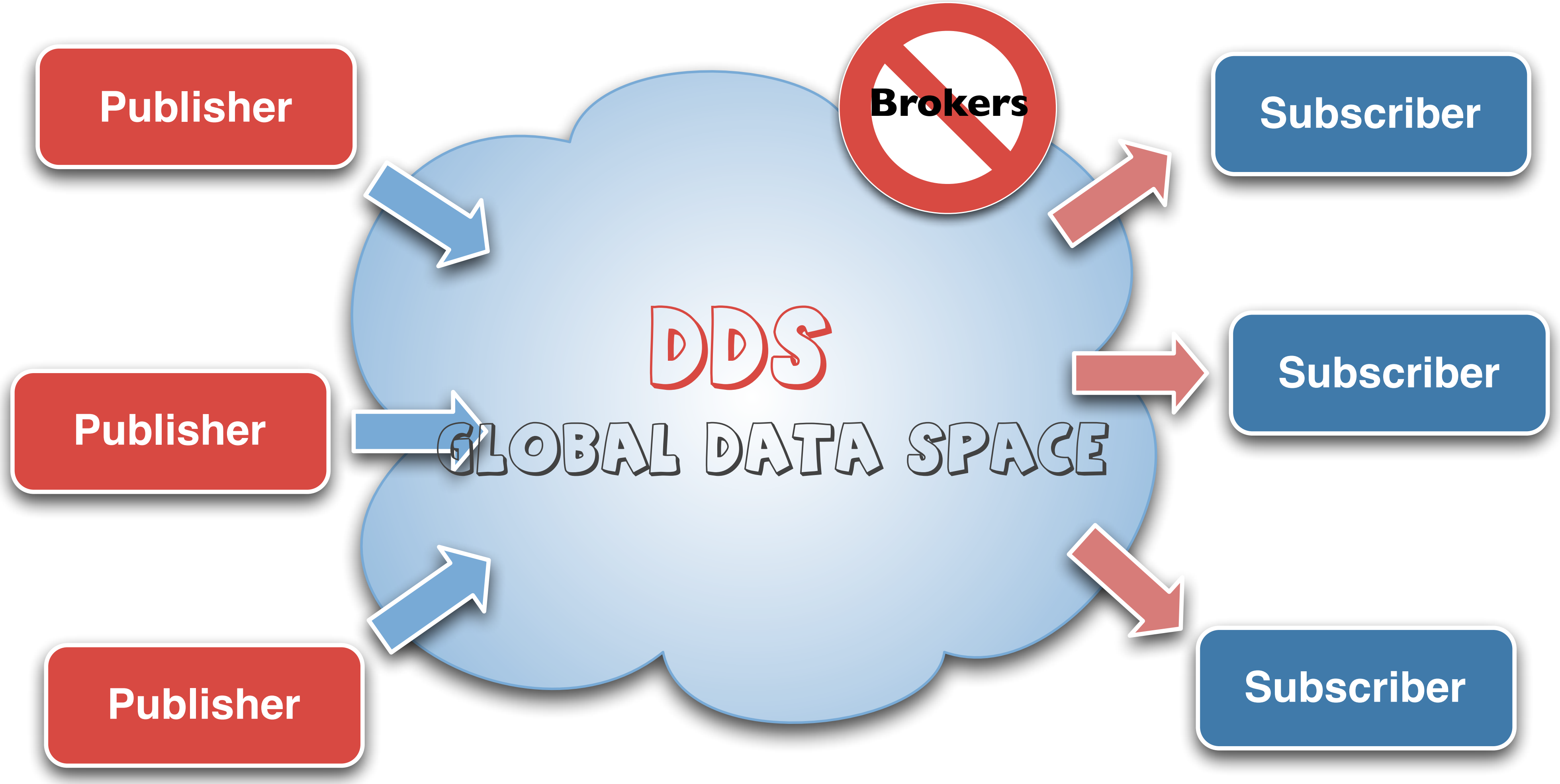
Note: DDS can be seen as a relaxation of the LINDA coordination model

- DDS is based around the concept of a **fully distributed Global Data Space (GDS)**
- Applications can autonomously and asynchronously read/written data in the GDS



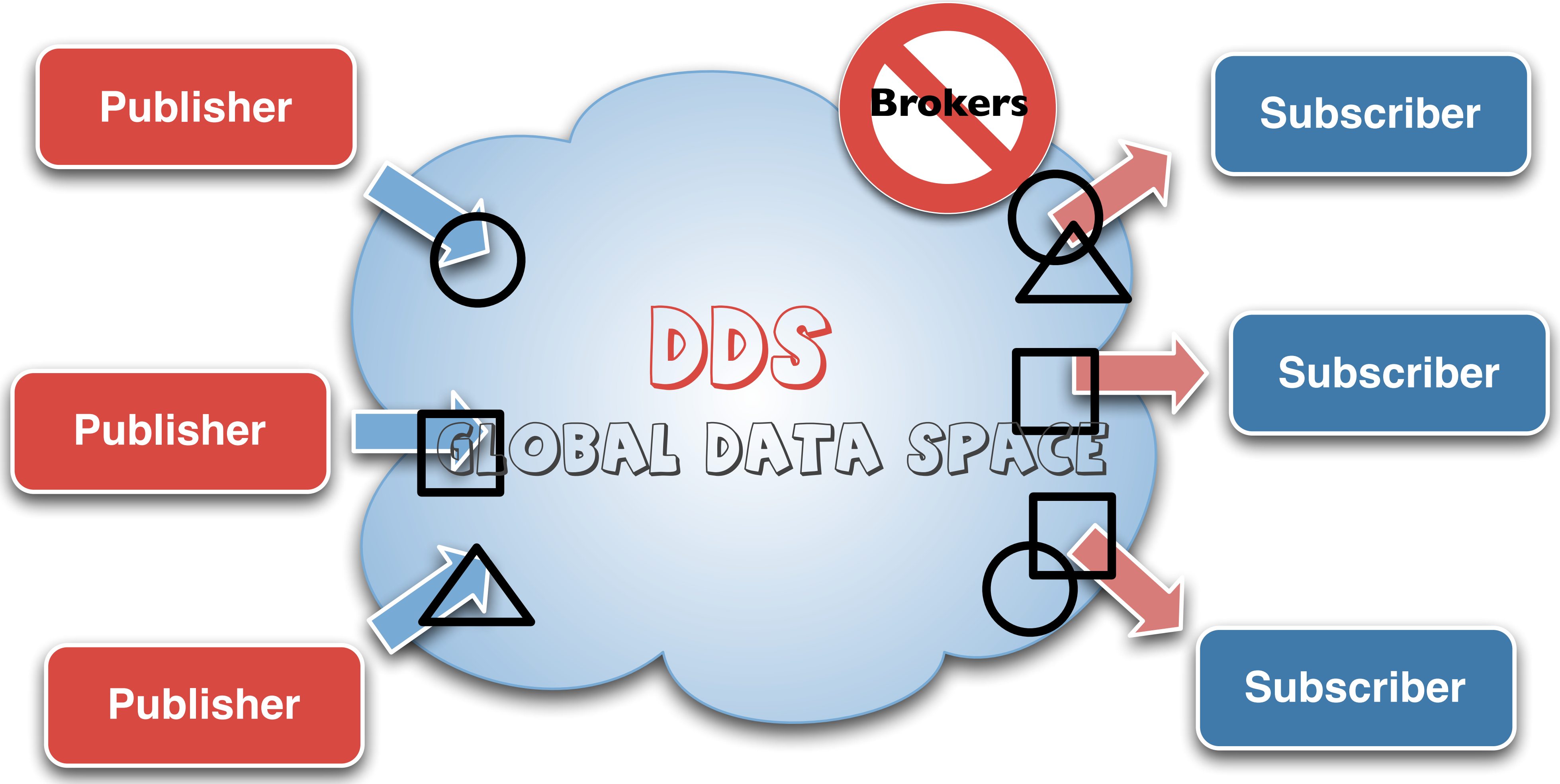
How Does it Work?

- **Publishers** and **Subscribers** can join and leave the GDS at any time



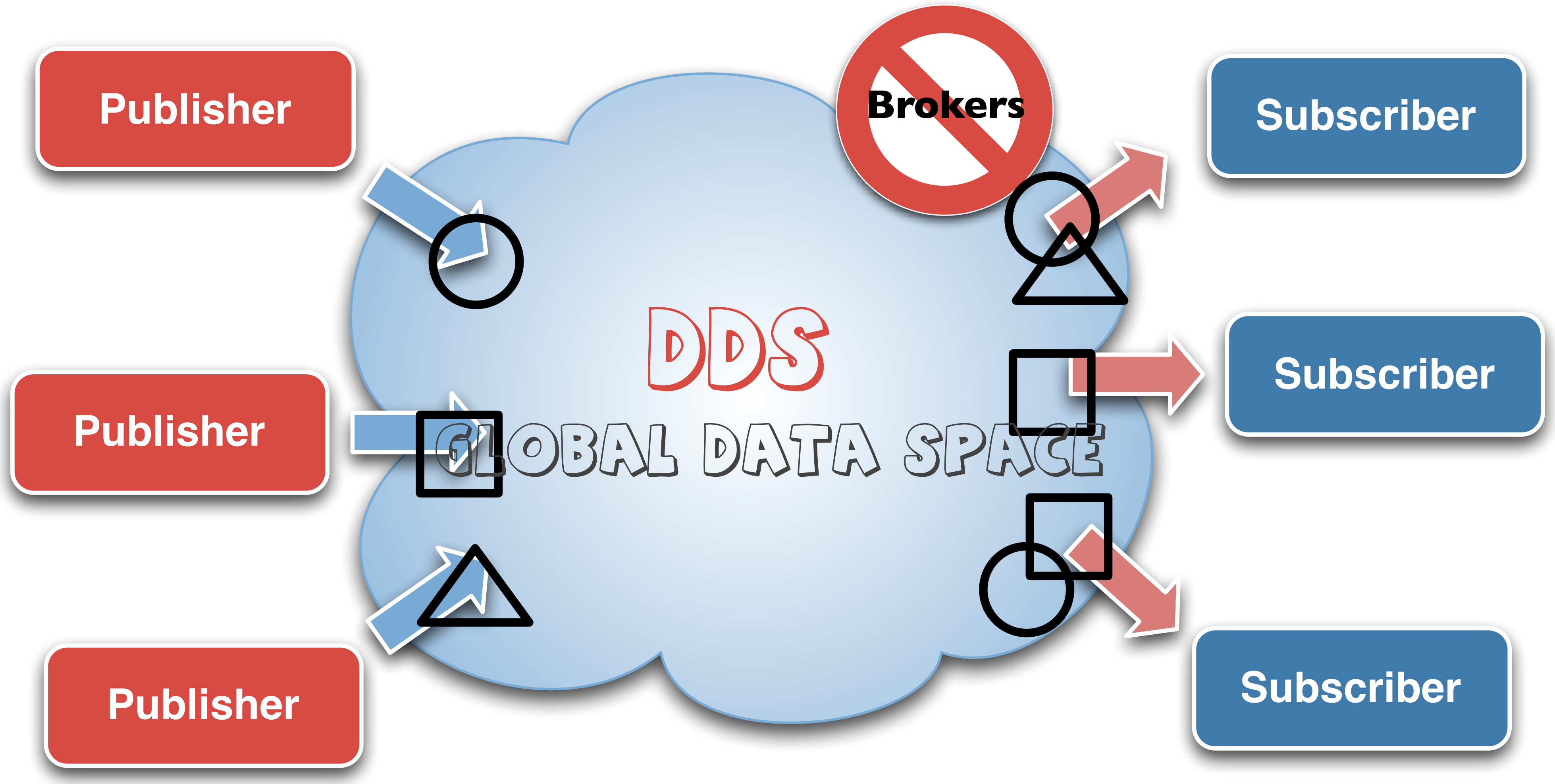
How Does it Work?

- **Publishers** and **Subscribers** express their intent to **produce/consume specific type of data**, e.g., **Topics**



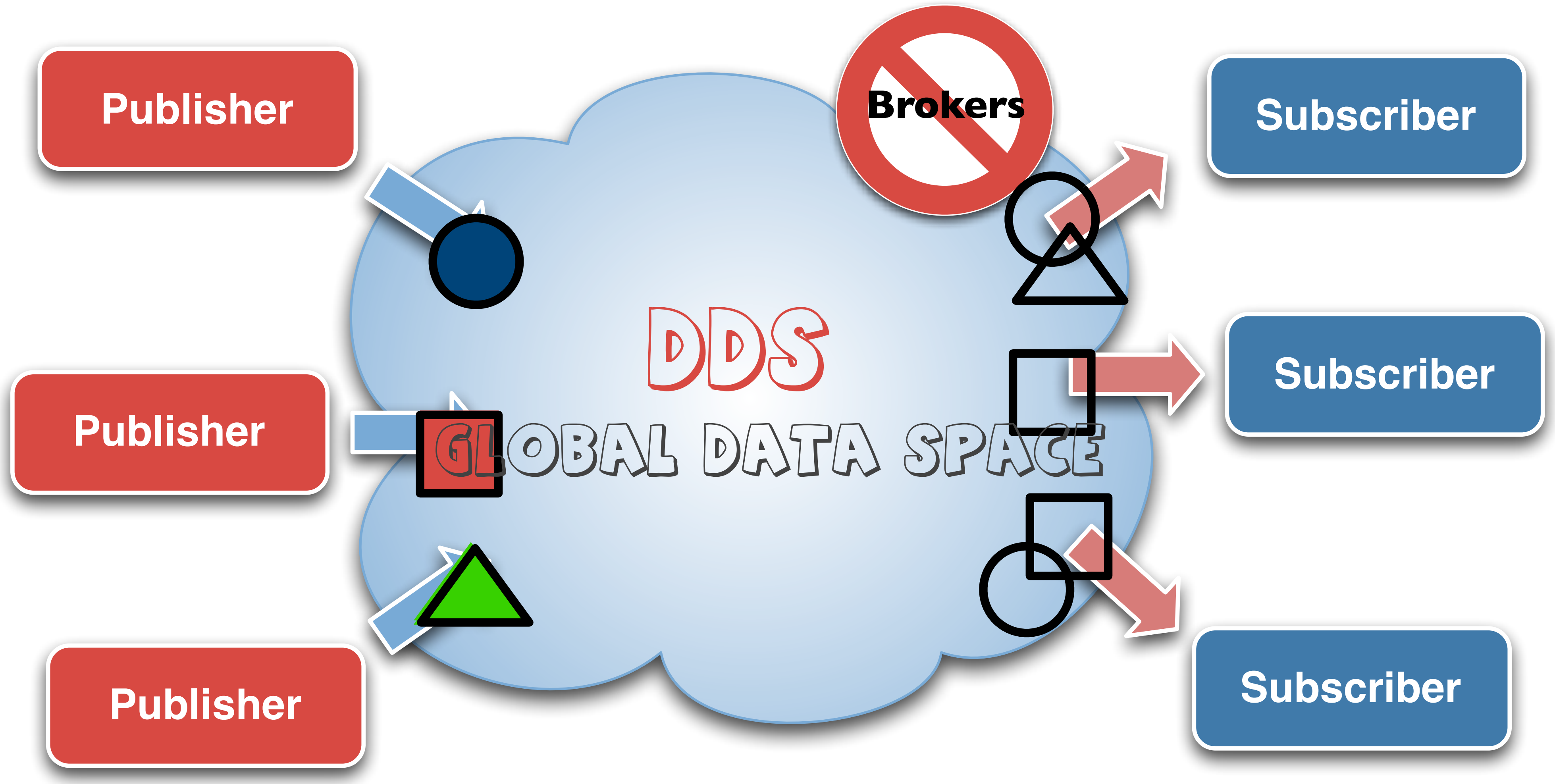
How Does it Work?

- Subscriptions are **matched** by taking into account topics (name, data type and QoS)



How Does it Work?

- Subscriptions are dynamically matched and Data flows from Publisher to Subscribers



OpenSplice|DDS

Delivering Performance, Openness, and Freedom

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

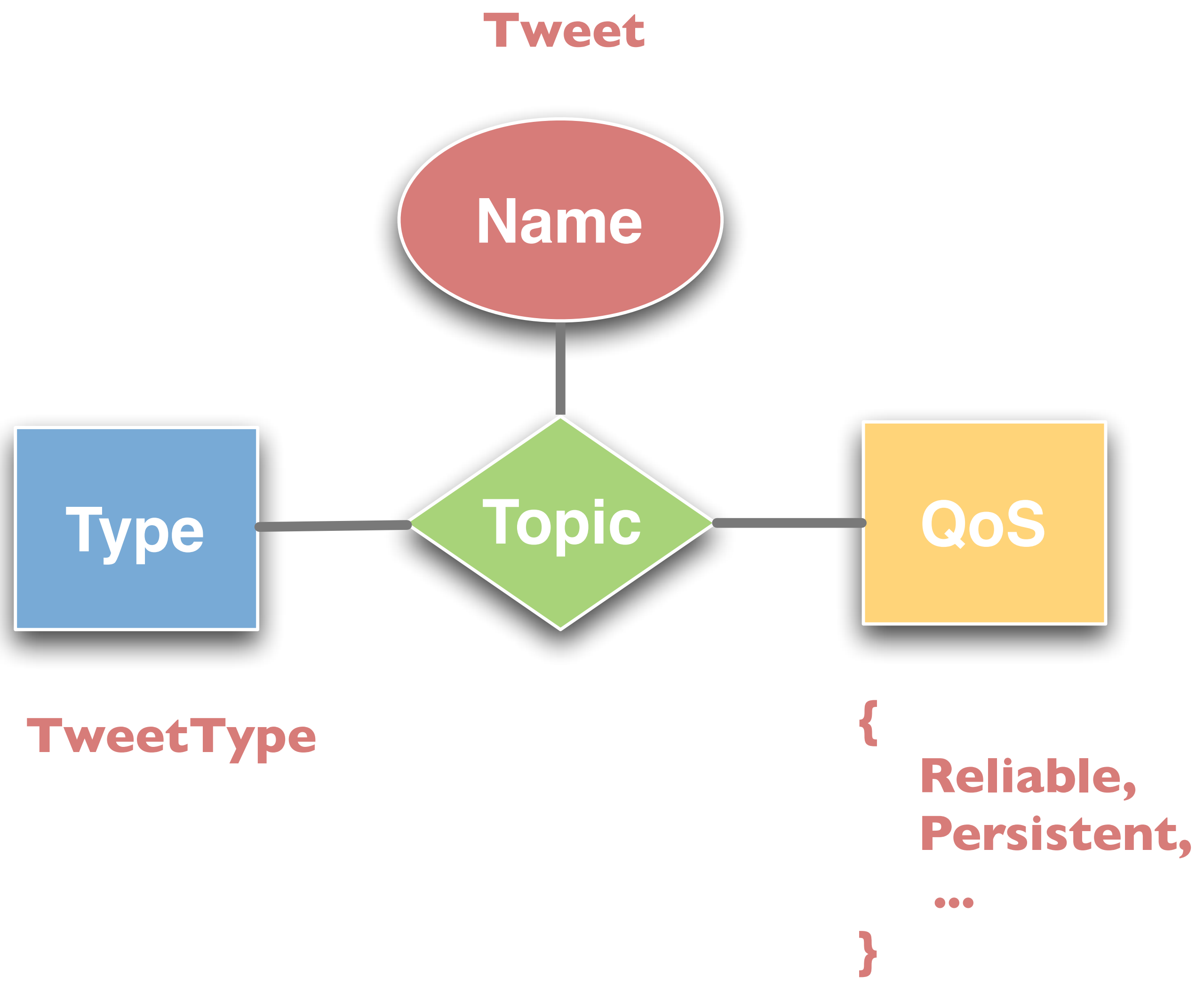
DEFINING DATA

A “Tweet” with DDS

TOPIC:

- Unit of information exchanged between Publisher and Subscribers.

- An association between a unique name, a type and a QoS setting



A “Tweet” with DDS

TOPIC TYPE:

- Type describing the data associated with one or more Topics
- A Topic type can have a key represented by an arbitrary number of attributes
- Expressed in IDL (or XML)

```
struct TweetType {
   string   userId;
   string   tweet;
};
#pragma keylist Tweet userId
```

```
struct ShapeType {
   long   x;
   long   y;
   long   shapesize;
   string color;
};
#pragma keylist ShapeType color
```

DDS Topic Instances and Samples

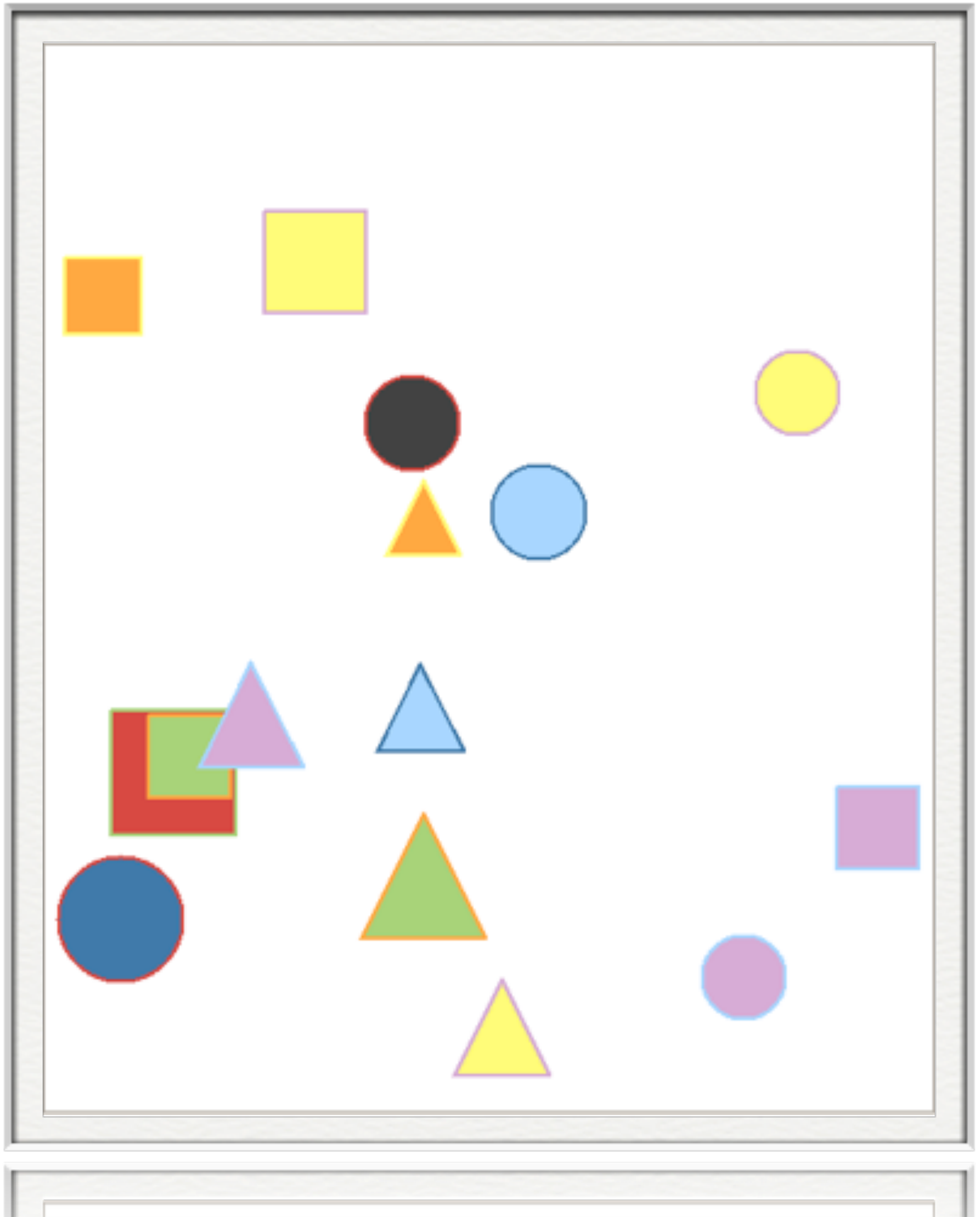
Topic Instances

- Each key value identifies a unique Topic Instance
- Topic's instance lifetime can be explicitly managed in DDS

```
struct ShapeType {
   long   x;
   long   y;
   long   shapesize;
   string color;
};
#pragma keylist ShapeType color
```

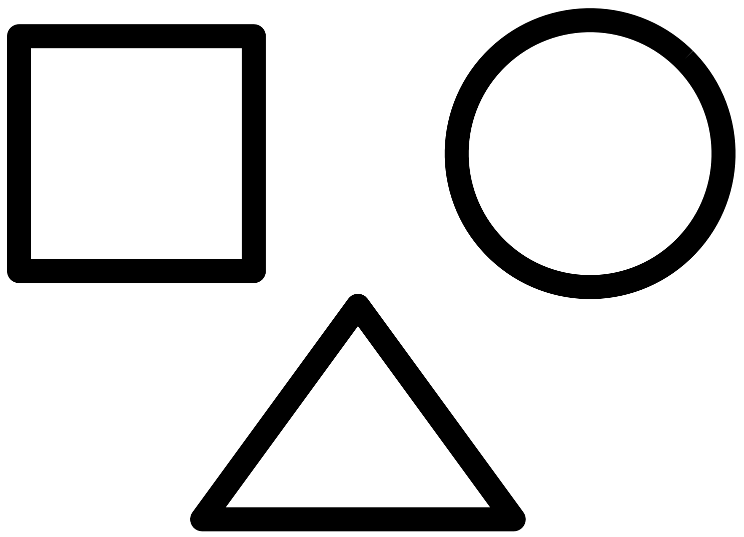
Topic Samples

- The values assumed by a Topic Instance over time are referred as Instance Sample

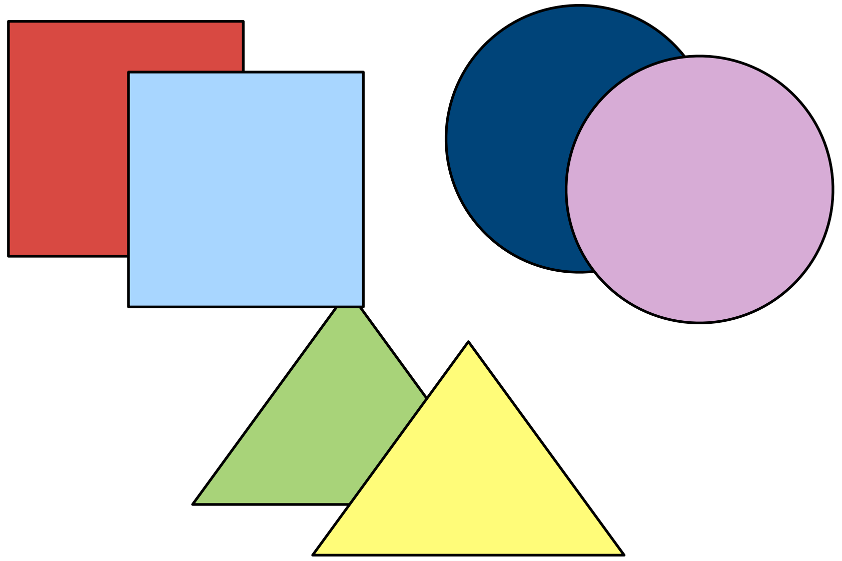


Topic/Instances/Samples Recap.

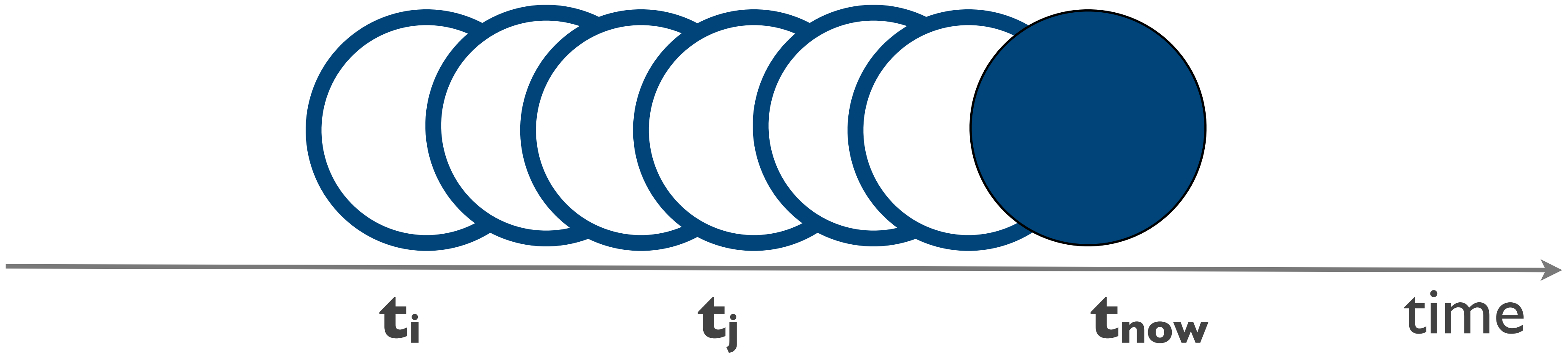
Topics



Instances



Samples



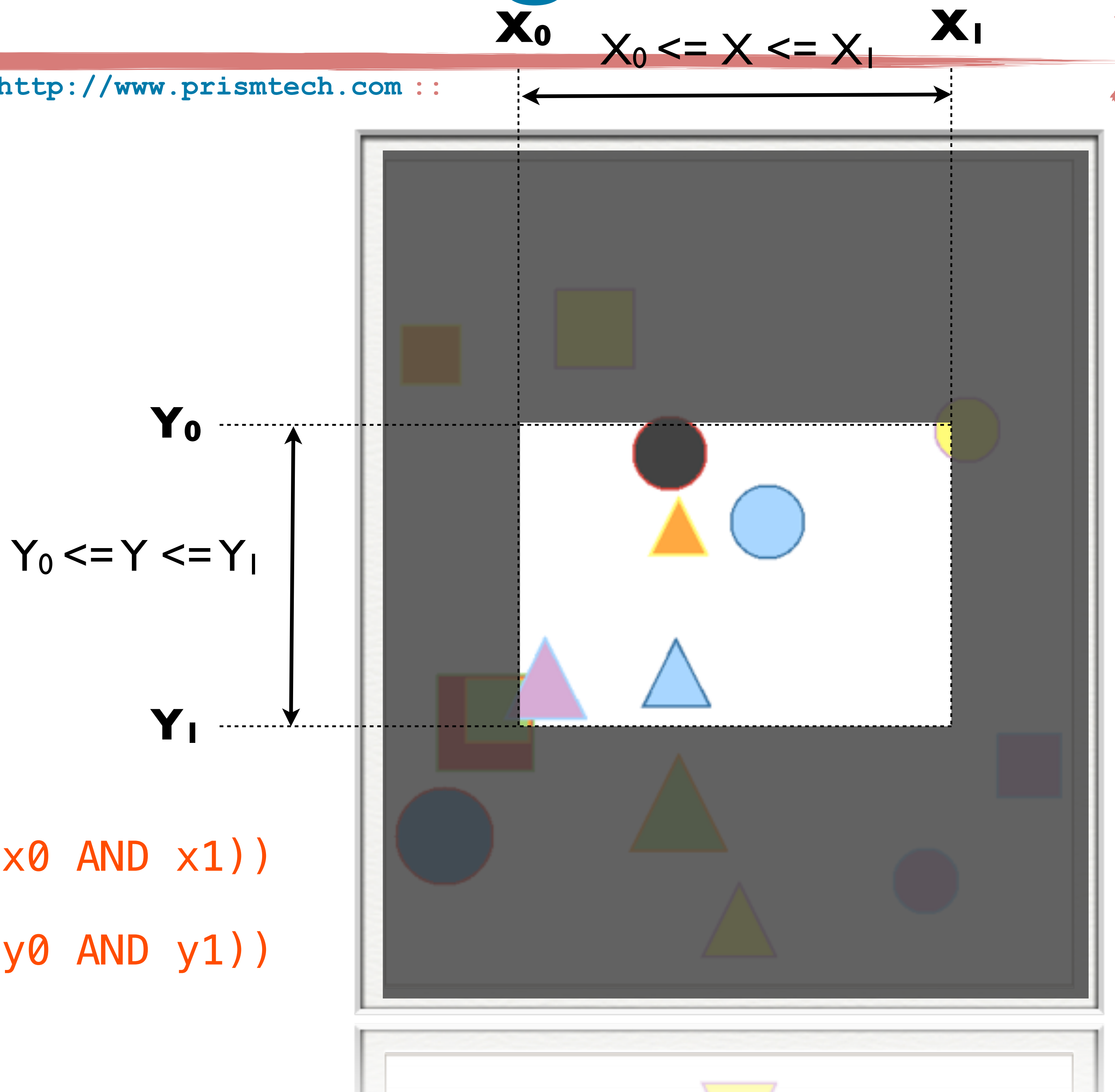
Content Filtering

- DDS allows the use of a subset of SQL92 to specify **content-filtered Topics**

- Content filters can be applied on the entire content of the Topic Type

- Content filters are applied by DDS each time a new sample is produced/delivered

```
(x BETWEEN (RANGE x0 AND x1))
 AND
(y BETWEEN (RANGE y0 AND y1))
```



Local Queries

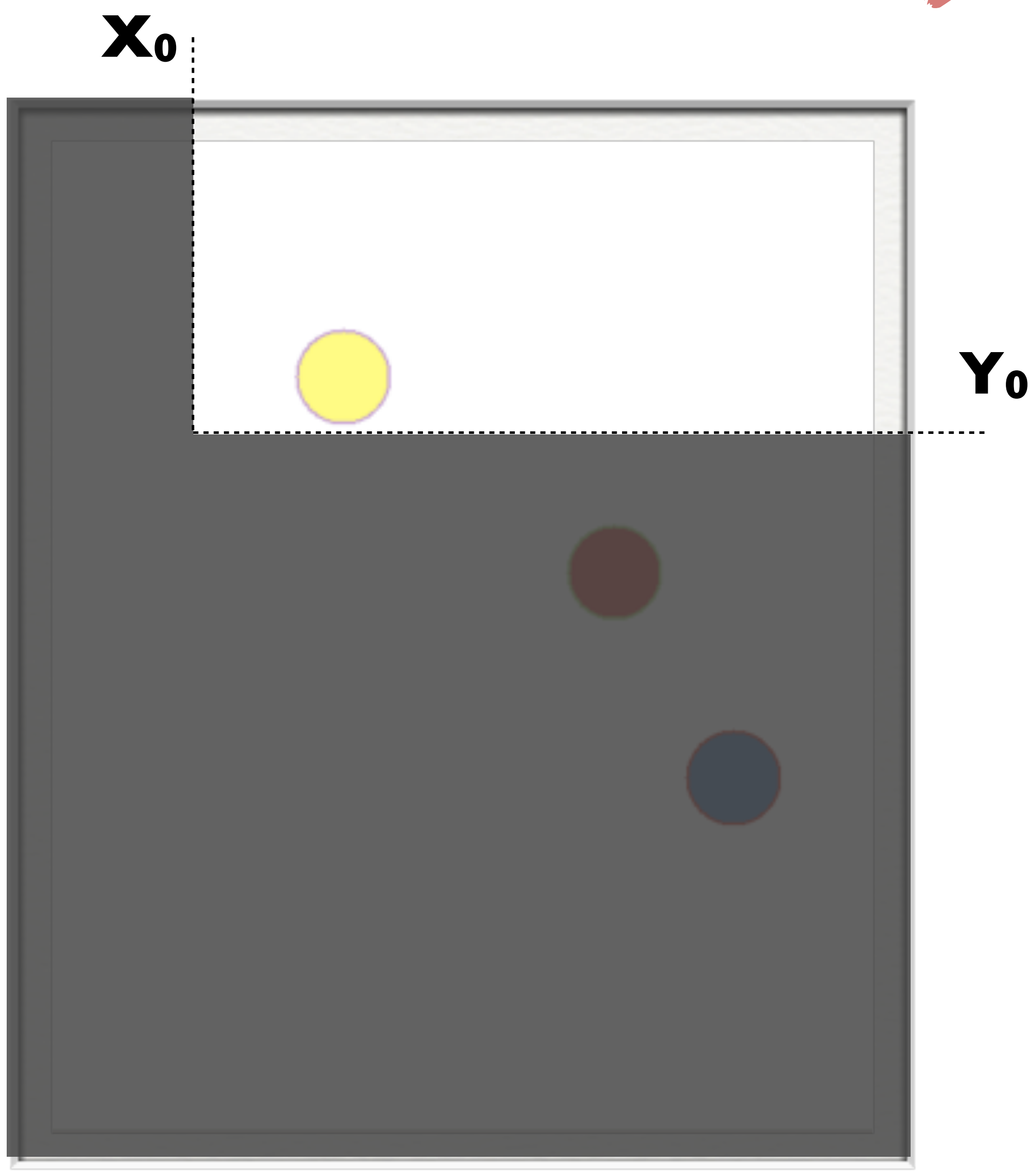
- Subscribed Topics can be seen locally as “Tables”
- A subset of SQL92 can be used for performing queries
- Queries are performed under user control and provide a result that depends on the current snapshot of the system, e.g., samples currently available

Circle Topic

color	x	y	shapesize
red	57	62	50
blue	90	85	50
yellow	30	25	50

x > 25 AND y < 55

color	x	y	shapesize
yellow	30	25	50



OpenSplice|DDS

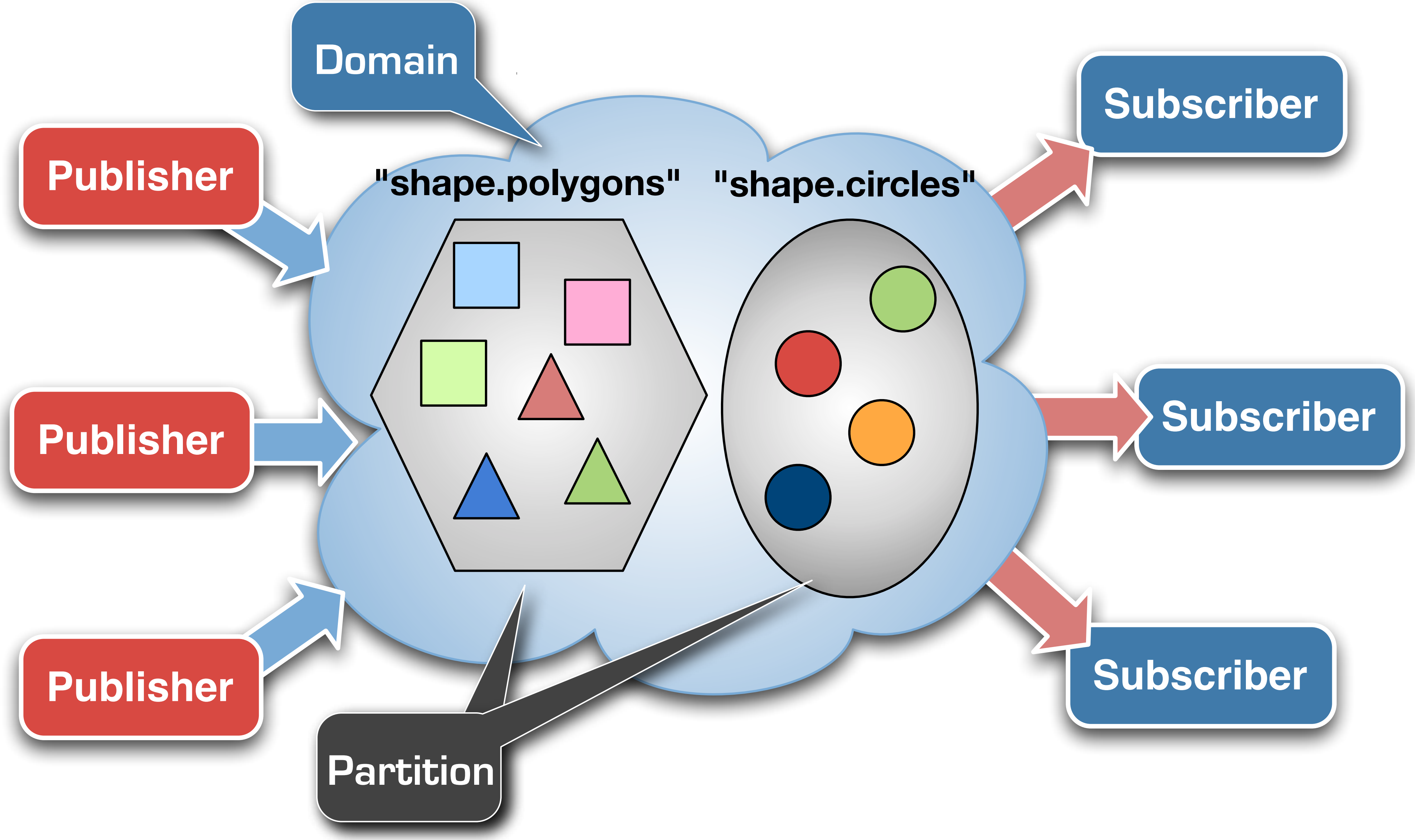
Delivering Performance, Openness, and Freedom

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

ORGANIZING DATA

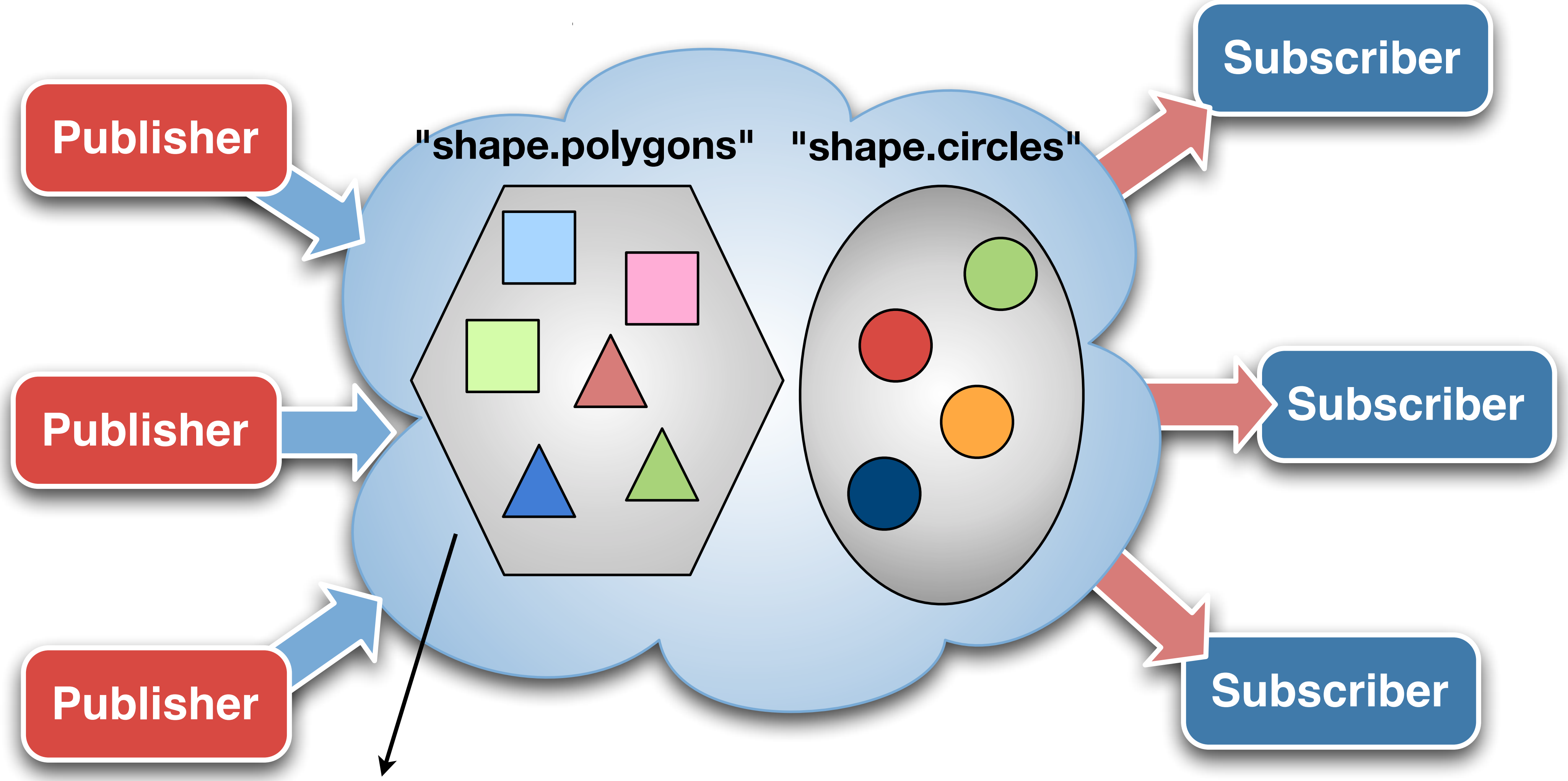
DDS Partitions

- All DDS communication confined within a **Domain**
- A domain can organized into **Partitions**
- Partitions can be used as **subjects** organizing the flow of data
- Publishers/Subscribers can connect to a Partitions' List which might also contain wildcards, e.g. shape.*
- **Topics** are published and subscribed across on or more Partitions



OpenSplice Network Partitions

- OpenSplice DDS allows to define **network partitions** along with **DDS partitions**
- Network partitions are bound to a list of unicast/multicast network addresses
- Partition.Topic combination can be mapped into OpenSplice DDS Network Partitions
- Wildcards can be used when defining the mapping, and in case of multiple matches OpenSplice DDS will always consider the best match



shape.polygons.* => 239.1.1.1

shape.polygons.Triangle => 239.1.1.2

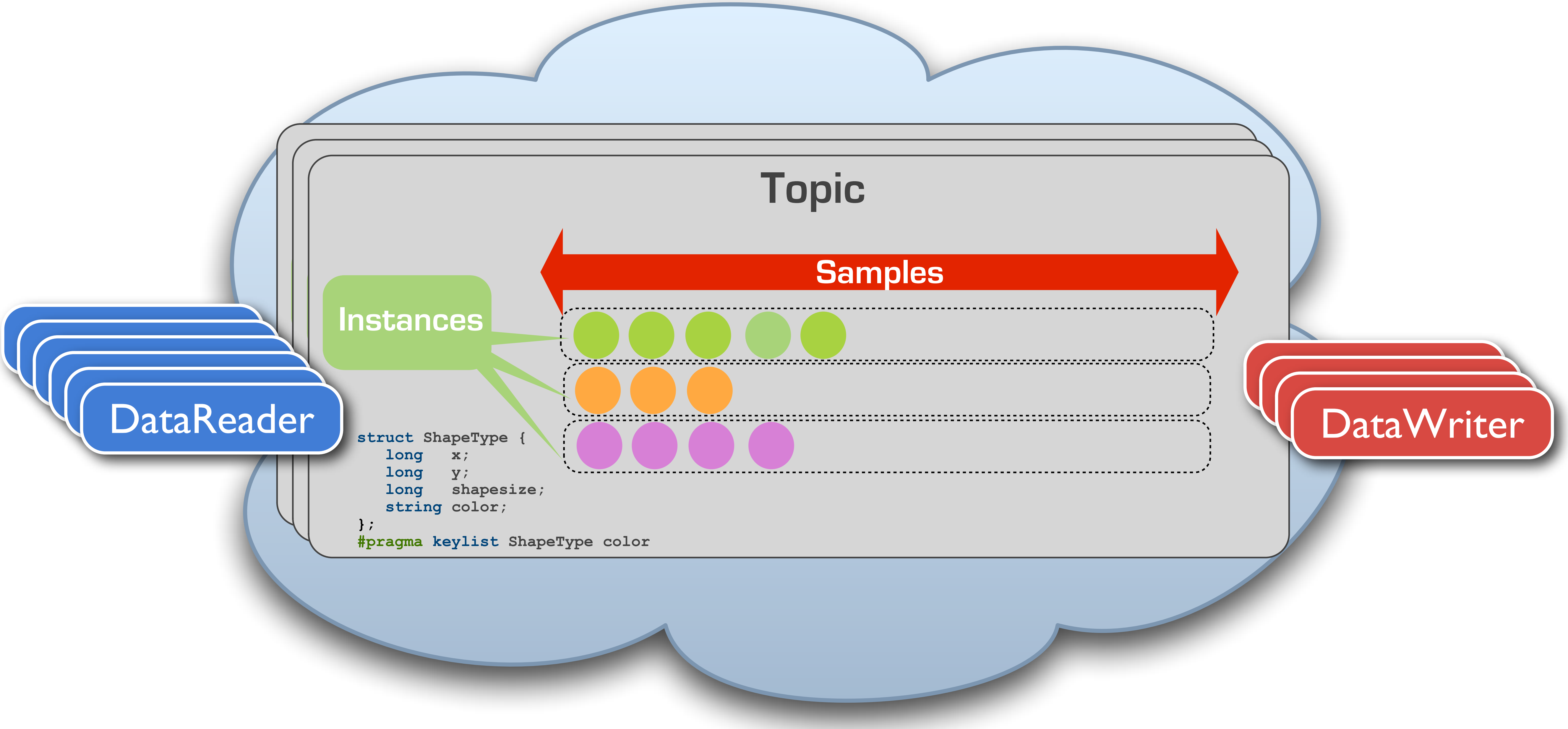
OpenSplice|DDS

Delivering Performance, Openness, and Freedom

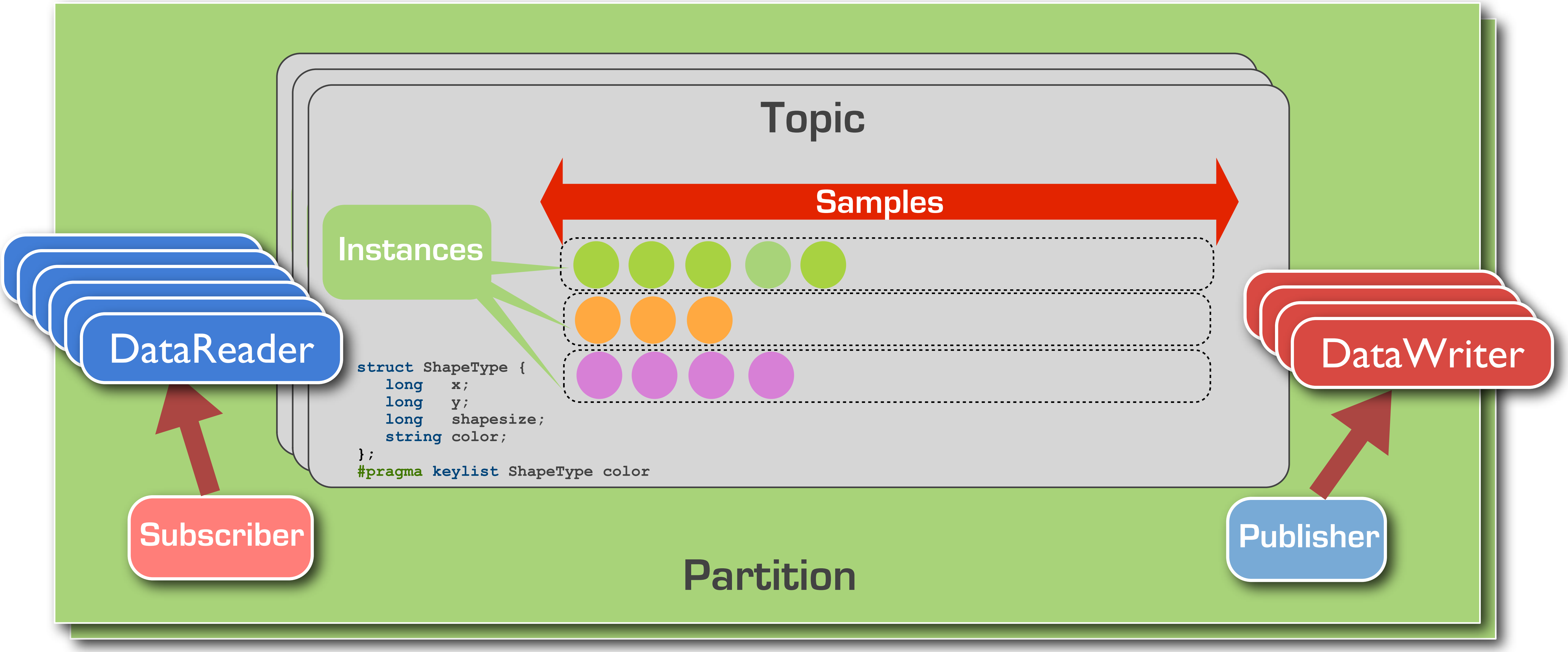
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

Quality of Service

Anatomy of a DDS Application

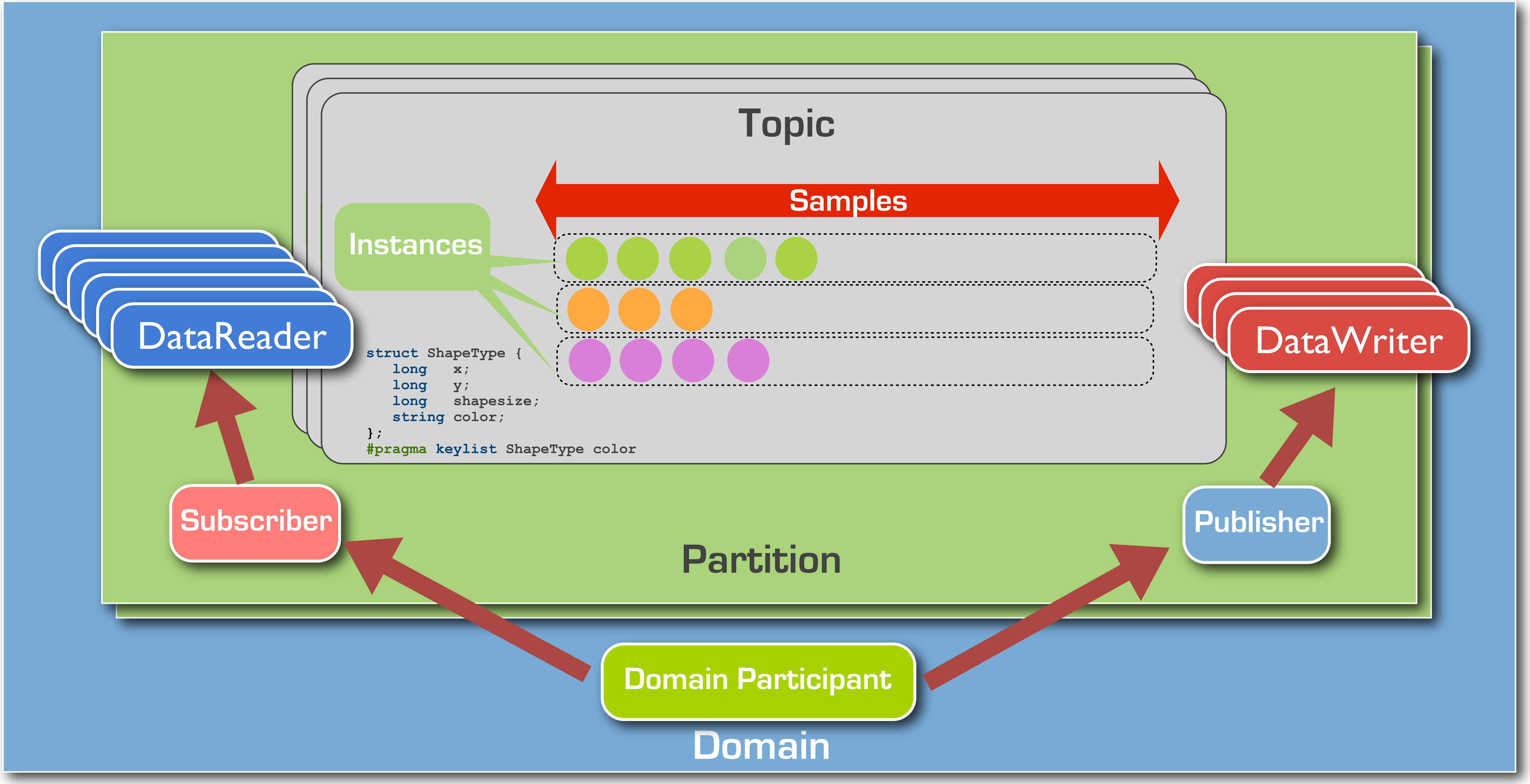


```
struct ShapeType {
   long    x;
   long    y;
   long    shapesize;
   string color;
};
#pragma keylist ShapeType color
```



```
struct ShapeType {
   long   x;
   long   y;
   long   shapesize;
   string color;
};
#pragma keylist ShapeType color
```

Arrows show structural relationships, not data-flows

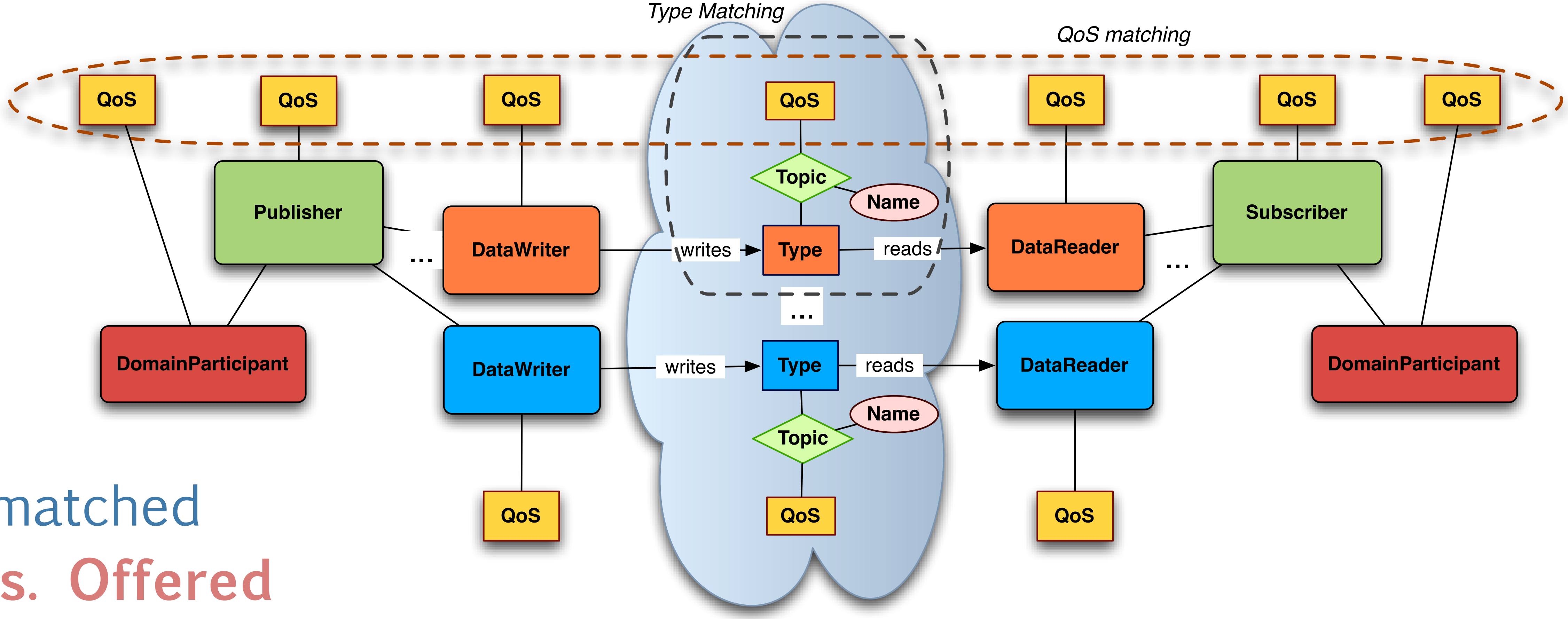


```
struct ShapeType {
    long   x;
    long   y;
    long   shapesize;
    string color;
};
#pragma keylist ShapeType color
```

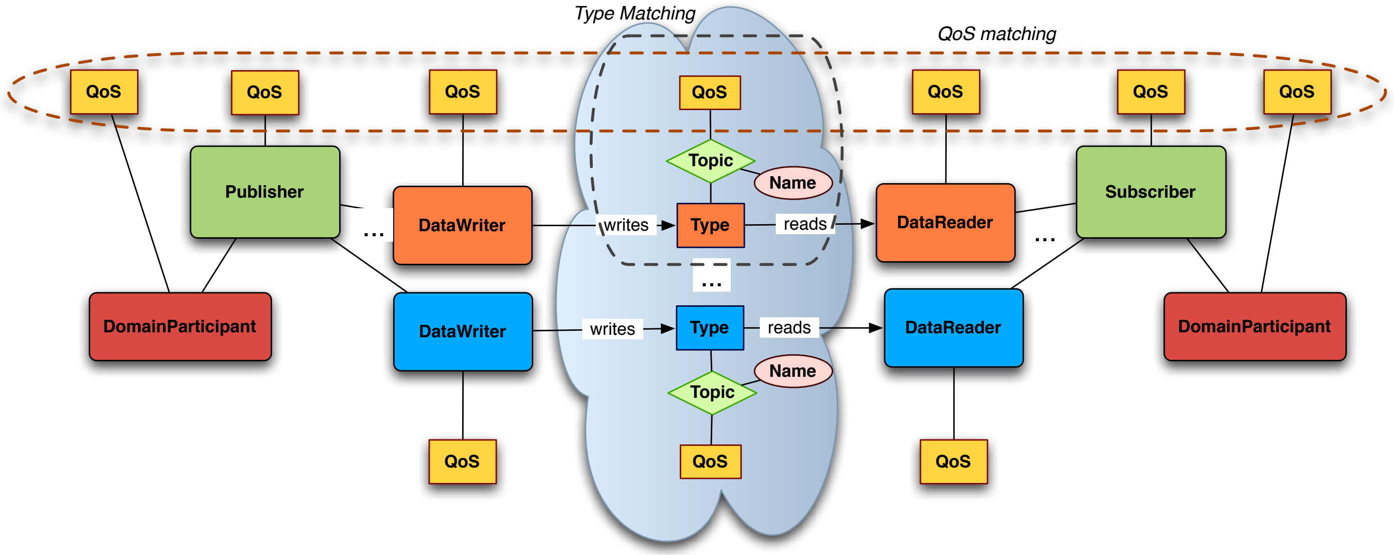
Arrows show structural relationships, not data-flows

QoS Model

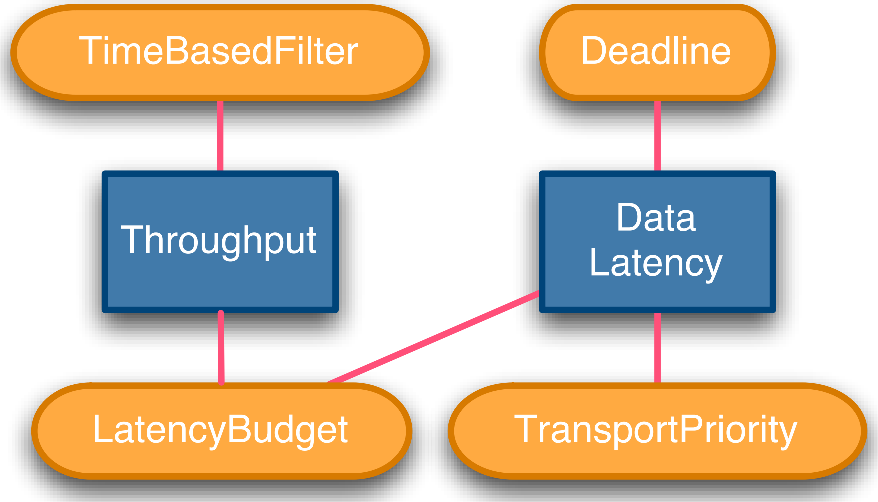
- QoS-Policies are used to control relevant properties of OpenSplice DDS entities, such as:
  - Temporal Properties
  - Priority
  - Durability
  - Availability
  - ...
- Some QoS-Policies are matched based on a **Request vs. Offered Model** thus QoS-enforcement



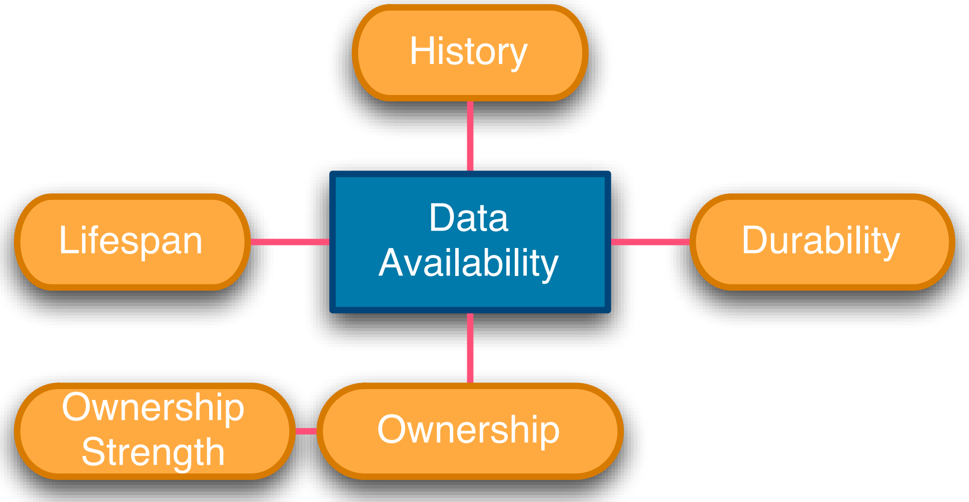
QoS Policy	Applicability	RxO	Modifiable	
DURABILITY	T, DR, DW	Y	N	Data Availability
DURABILITY SERVICE	T, DW	N	N	
LIFESPAN	T, DW	N/A	Y	
HISTORY	T, DR, DW	N	N	
PRESENTATION	P, S	Y	N	Data Delivery
RELIABILITY	T, DR, DW	Y	N	
PARTITION	P, S	N	Y	
DESTINATION ORDER	T, DR, DW	Y	N	
OWNERSHIP	T, DR, DW	Y	N	
OWNERSHIP STRENGTH	DW	N/A	Y	
DEADLINE	T, DR, DW	Y	Y	Data Timeliness
LATENCY BUDGET	T, DR, DW	Y	Y	
TRANSPORT PRIORITY	T, DW	N/A	Y	
TIME BASED FILTER	DR	N/A	Y	Resources
RESOURCE LIMITS	T, DR, DW	N	N	
ENTITY FACTORY				Configuration
USER DATA	DP, DR, DW	N	Y	
TOPIC DATA	T	N	Y	
GROUP DATA	P, S	N	Y	
LIVELINESS	T, DR, DW	Y	N	
WRITER DATA LIFECYCLE	DW	N/A	Y	Lifecycle
READER DATA LIFECYCLE	DR	N/A	Y	



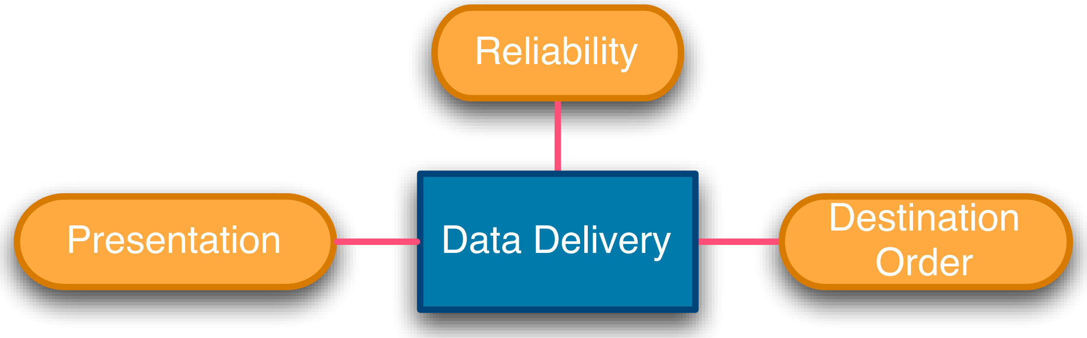
Mapping QoS



☑ Control over Latency/Throughput tradeoff
☑ Control over data latency
☑ Control over data priority



☑ Control over data queueing
☑ Control over data persistency
☑ Control over data sources hot-swap



☑ Control over data distribution reliability
☑ Control over data ordering
☑ Control over presentation

OpenSplice DDS provides programmatic QoS-driven support for configuring the most important properties of data distribution!

OpenSplice|DDS

Delivering Performance, Openness, and Freedom

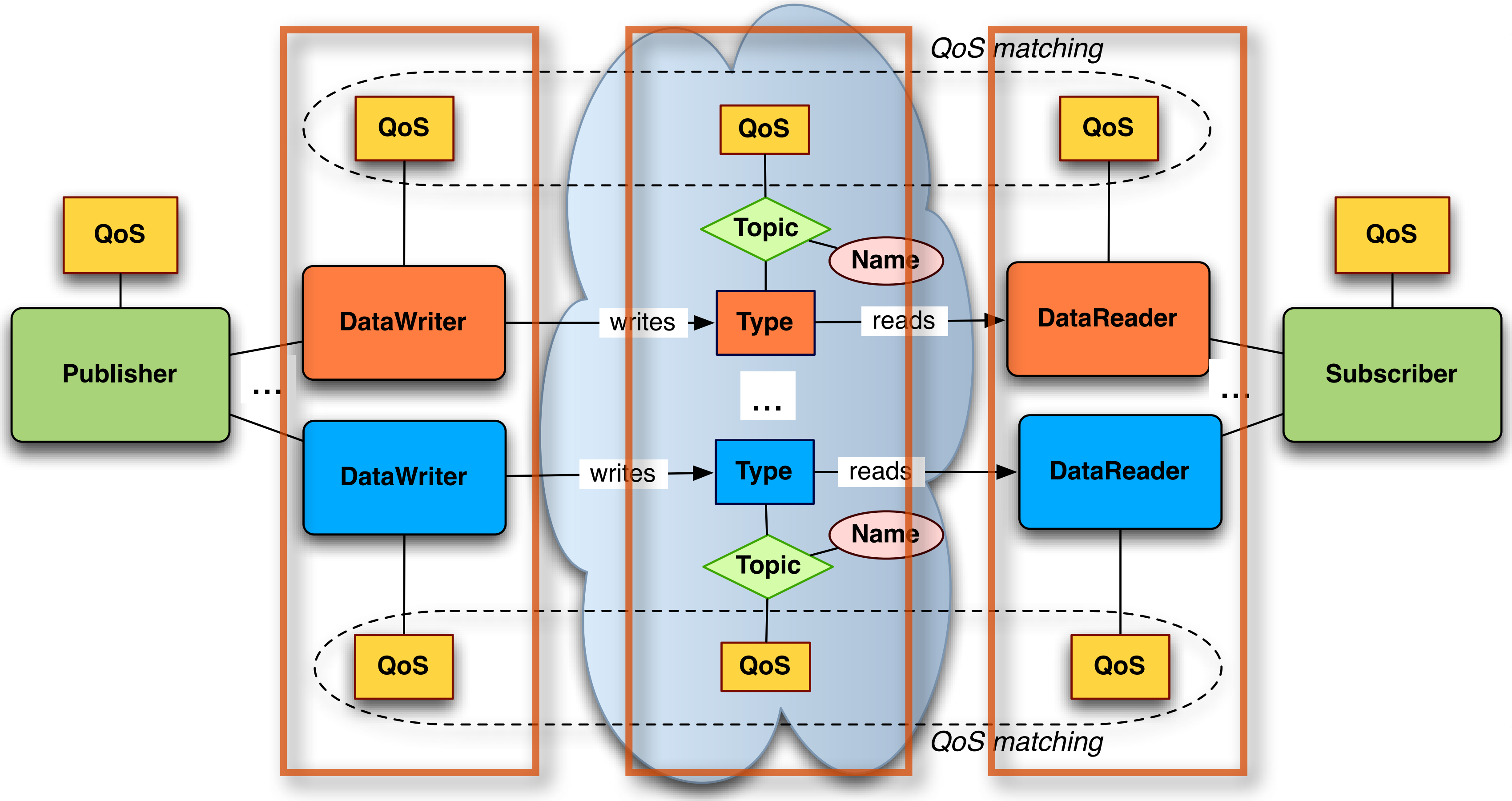
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

CONTROLLING RELIABILITY

Reliability

QoS Policy	Applicability	RxO	Modifiable
RELIABILITY	T, DR, DW	Y	N

The **RELIABILITY** QoS indicate the level of guarantee offered by the DDS in delivering data to subscribers.

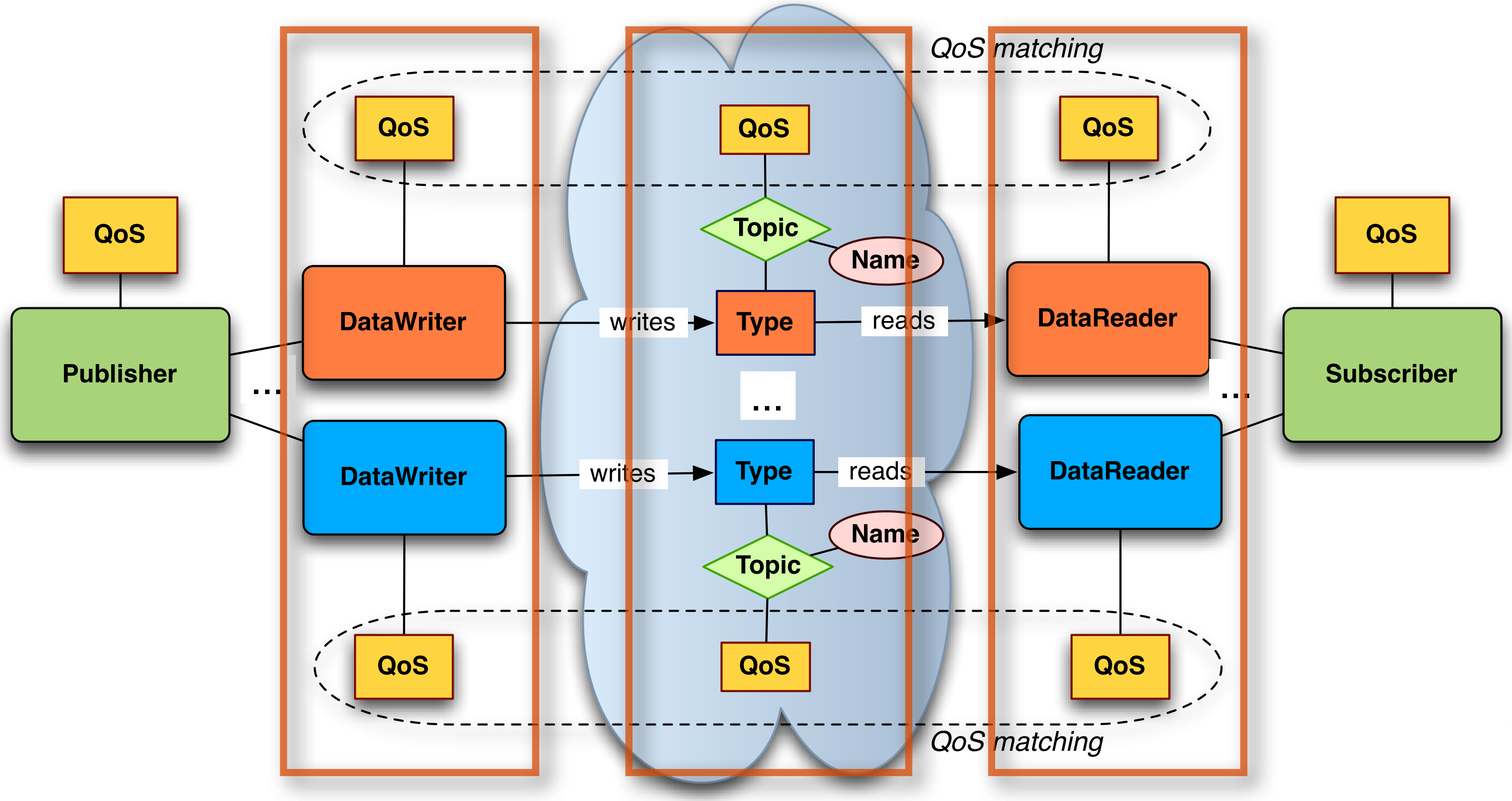


Reliability

Possible variants are:

- **Reliable.** In steady-state the middleware guarantees that all samples in the DataWriter history will eventually be delivered to all the DataReader
- **Best Effort.** Indicates that it is acceptable to not retry propagation of any samples

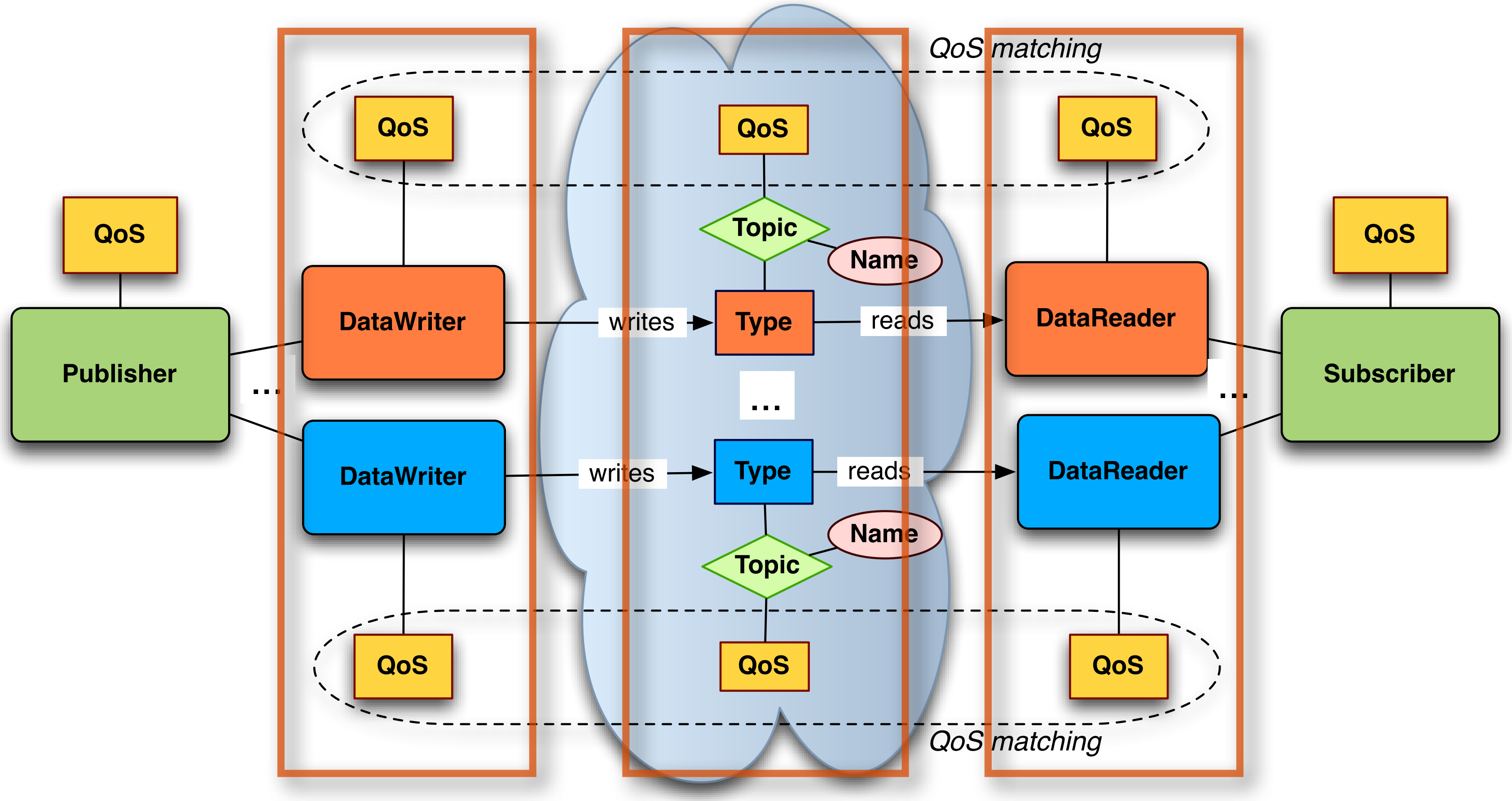
QoS Policy	Applicability	RxO	Modifiable
RELIABILITY	T, DR, DW	Y	N



History

QoS Policy	Applicability	RxO	Modifiable
HISTORY	T, DR, DW	N	N

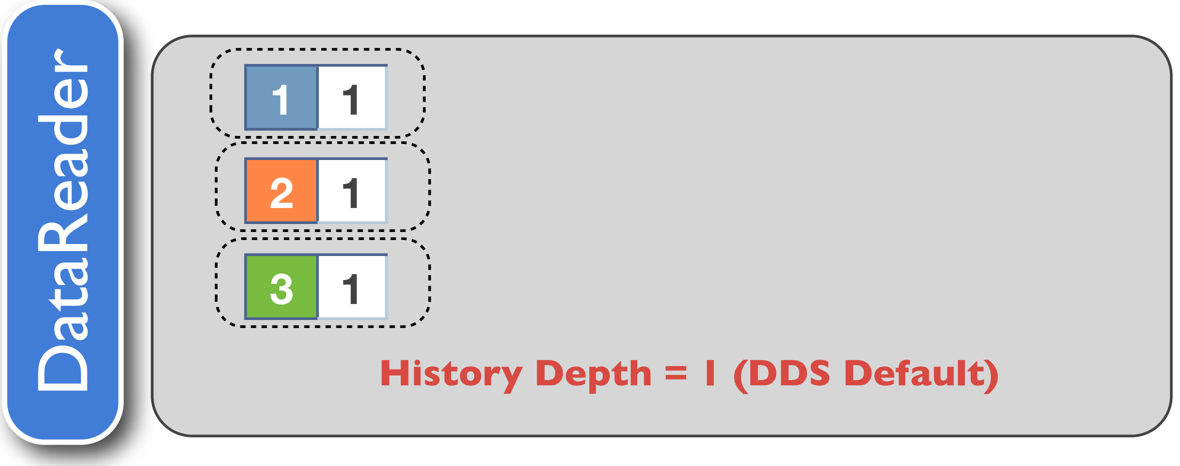
The **HISTORY** QoS policy controls whether the DDS should deliver only the most recent value, attempt to deliver all intermediate values, or do something in between.



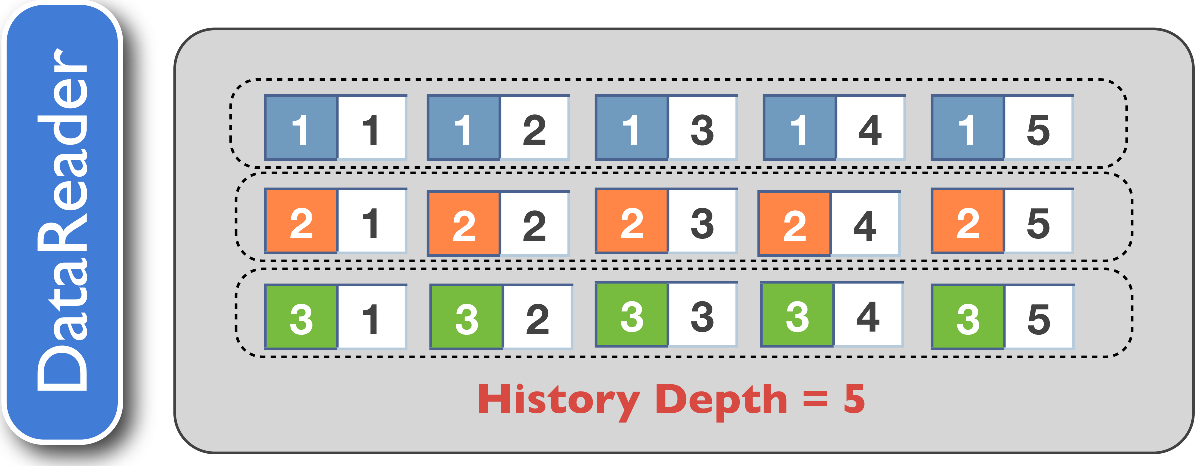
History

The policy can be configured to provide the following semantics:

- **Keep Last.** The DDS will only attempt to keep the most recent "depth" samples of each instance of data identified by its key
- **Keep All.** The DDS will attempt to keep all the samples of each instance of data identified by its key.



History Depth = 1 (DDS Default)



History Depth = 5

OpenSplice|DDS

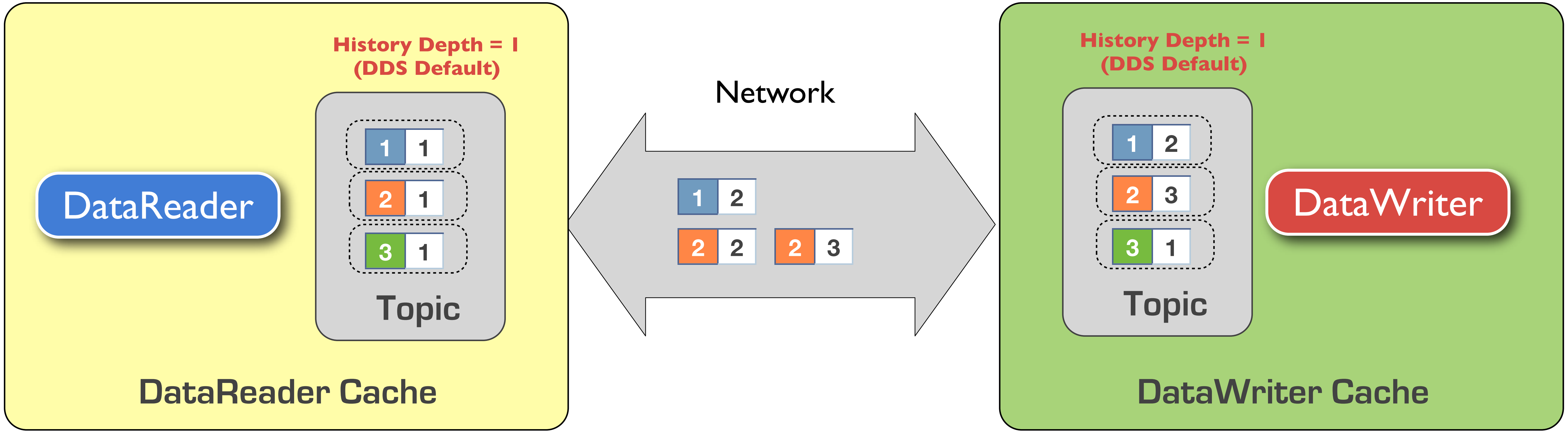
Delivering Performance, Openness, and Freedom

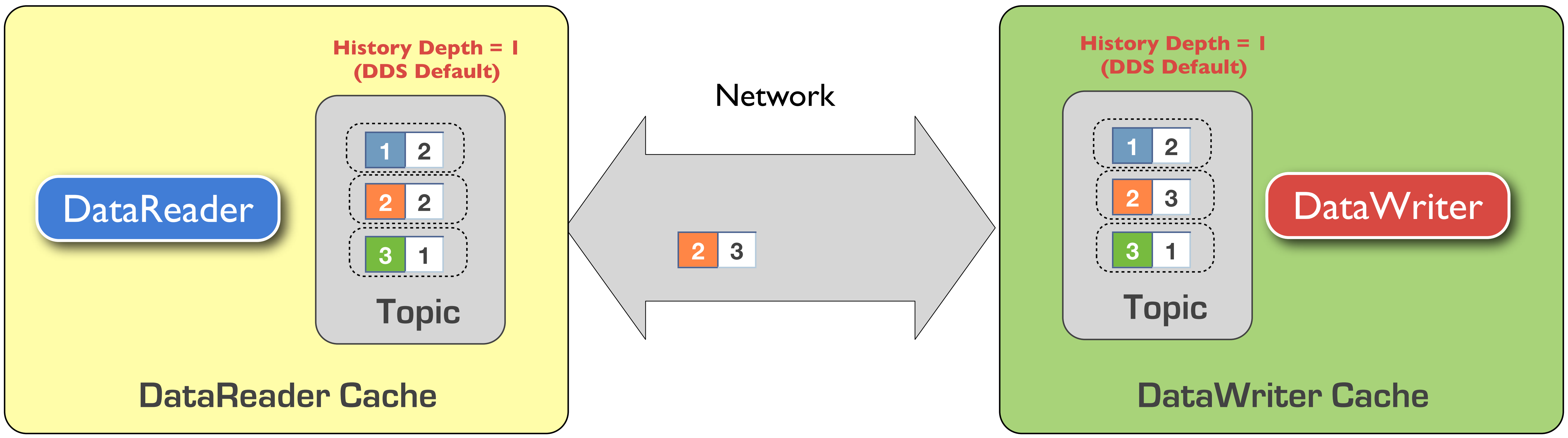
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

Topic Used in next section.

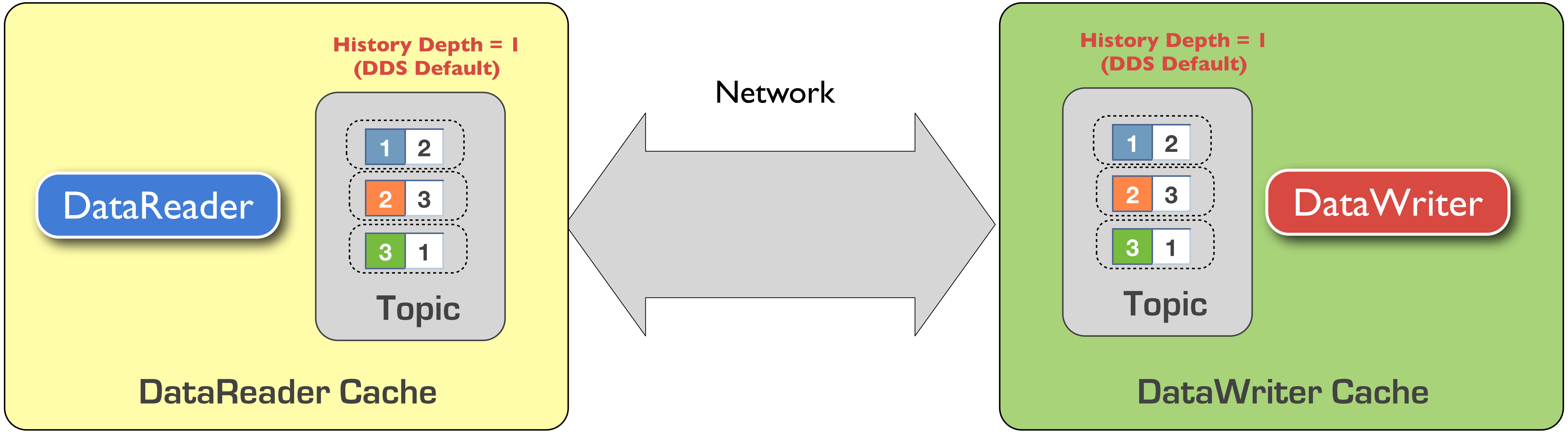
```
struct Counter {
   int cID;
   int count;
};
#pragma keylist Counter cID
```

HISTORY IN ACTION

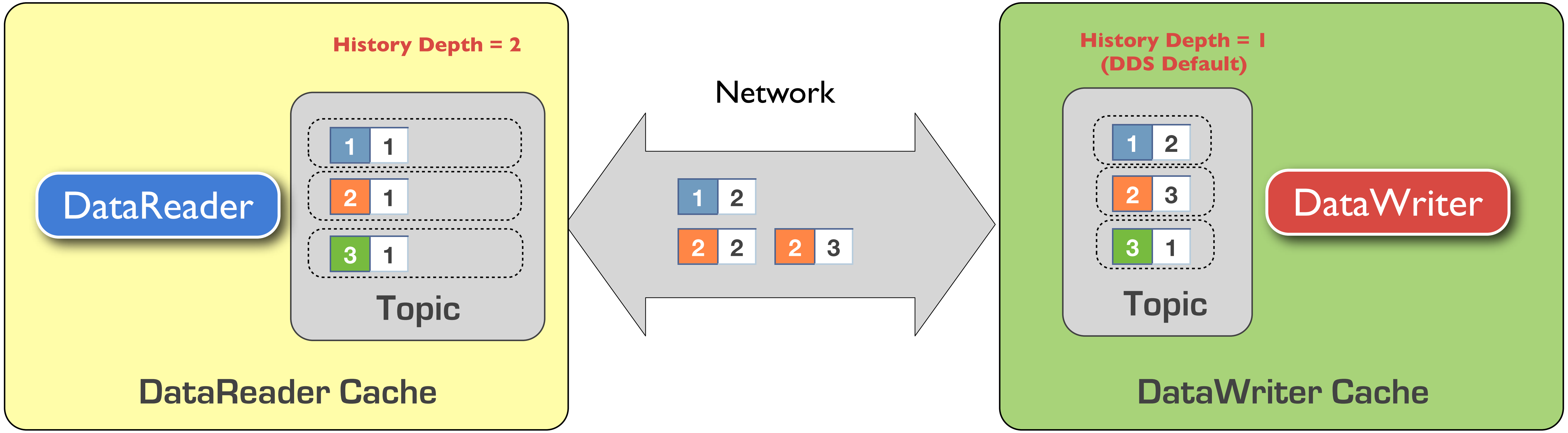

History Depth = 1
(DDS Default)
1 1
2 1
3 1
DataReader
Topic
DataReader Cache
Network
1 2
2 2
2 3
History Depth = 1
(DDS Default)
1 2
2 3
3 1
DataWriter
Topic
DataWriter Cache



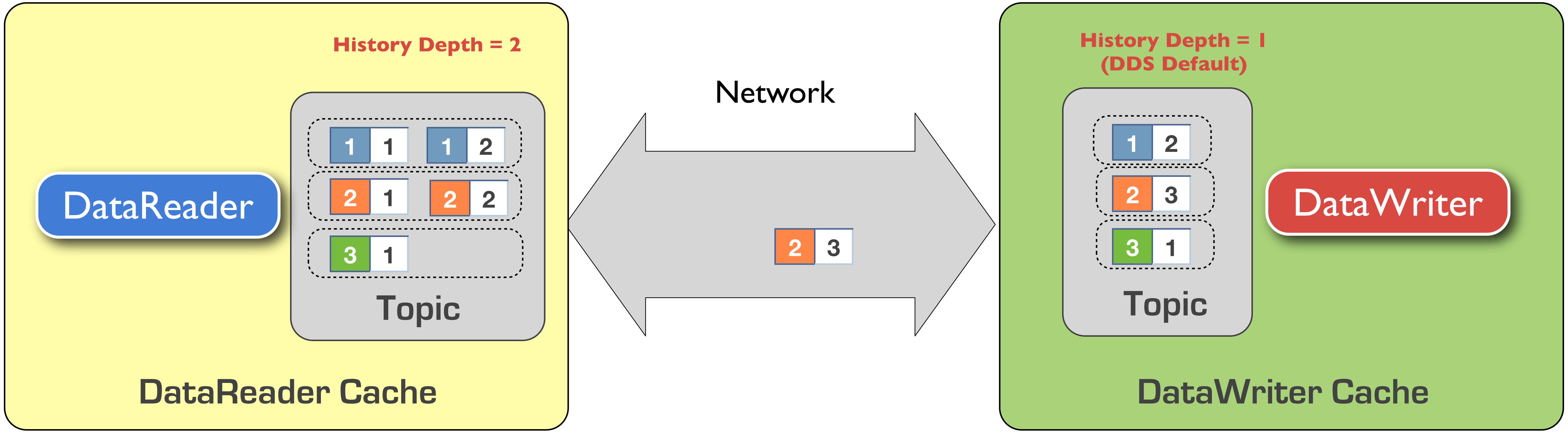
Note: The Reliability QoS controls wether data is sent reliably, or best-effort, from the DataWriter to matched DataReaders



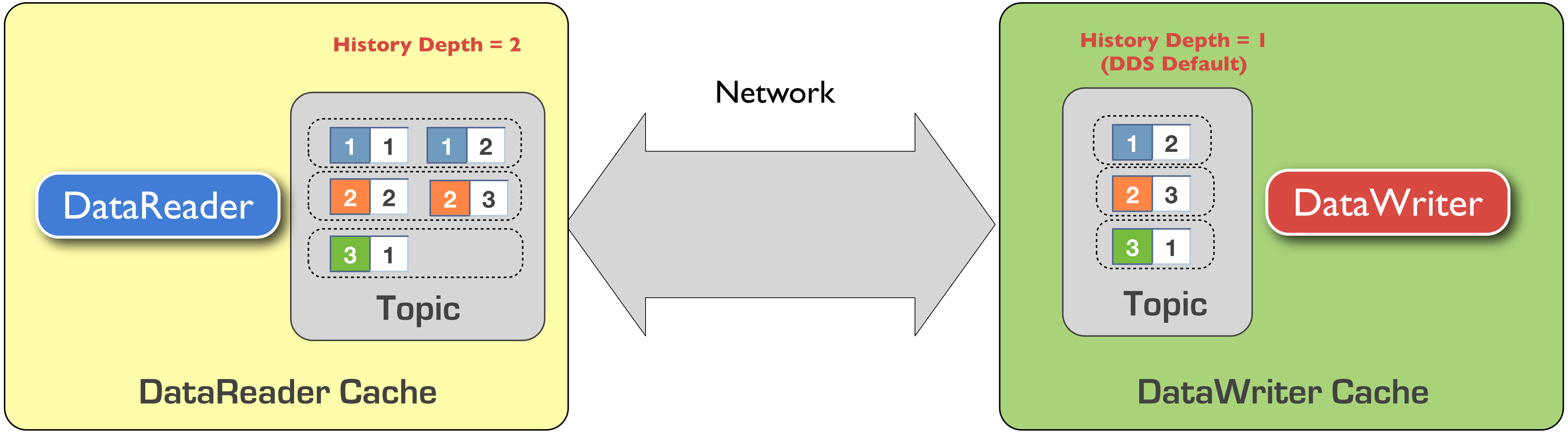
Note: The Reliability QoS controls wether data is sent reliably, or best-effort, from the DataWriter to matched DataReaders



Note: The Reliability QoS controls wether data is sent reliably, or best-effort, from the DataWriter to matched DataReaders



Note: The Reliability QoS controls wether data is sent reliably, or best-effort, from the DataWriter to matched DataReaders

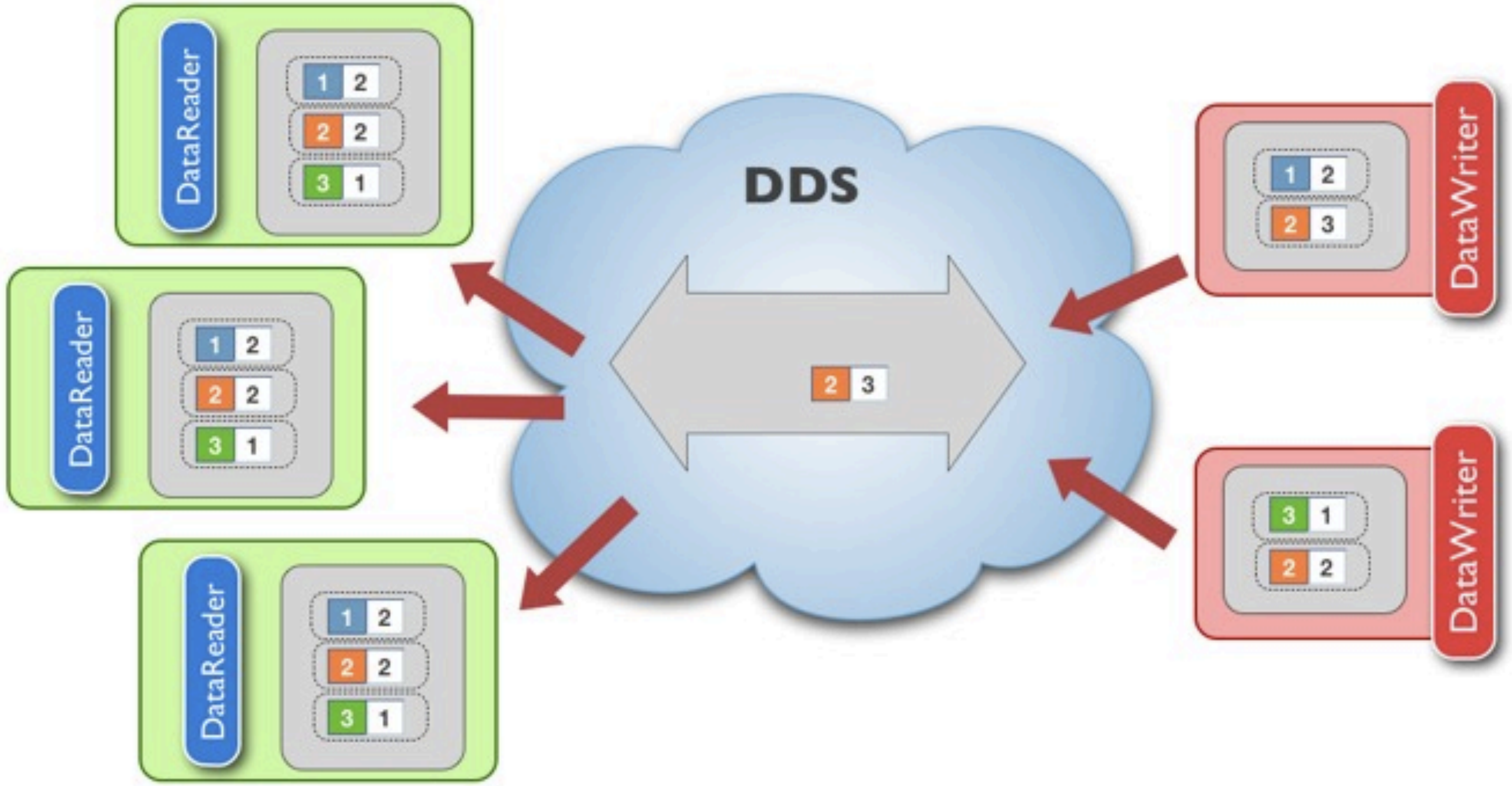


Note: The Reliability QoS controls wether data is sent reliably, or best-effort, from the DataWriter to matched DataReaders

Putting it All Together

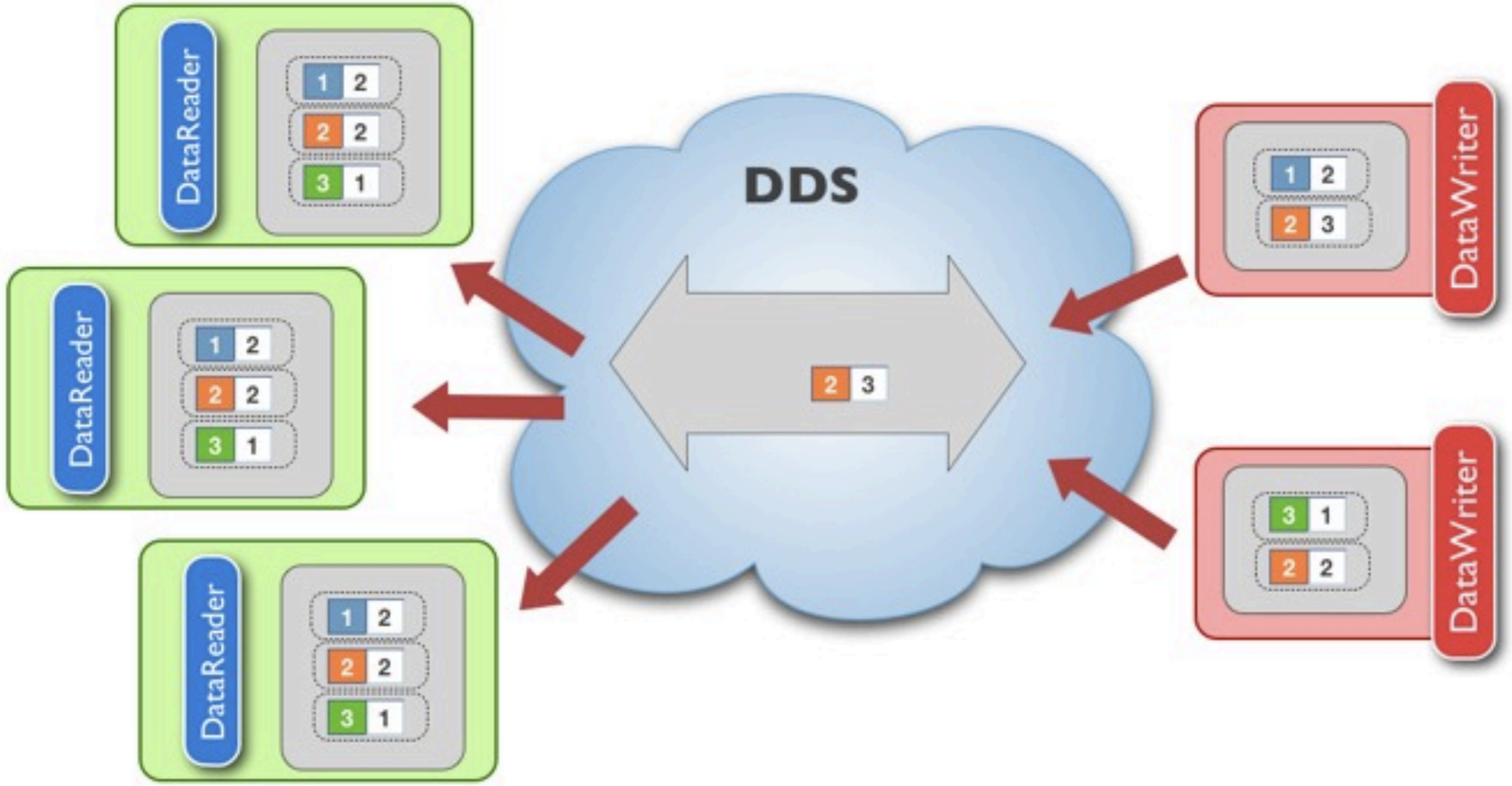
The reliability with which data is delivered to applications is impacted in DDS by the following qualities of service

- RELIABILITY
  - BEST_EFORT
  - RELIABLE
- HISTORY
  - KEEP_LAST (K)
  - KEEP_ALL



Putting it All Together

- **Theoretically**, the only way to assure that **an application will see all the samples** produced by a writer is to use **RELIABLE +KEEP_ALL**. Any other combination could induce to samples being discarded on the receiving side because of the HISTORY depth



OpenSplice | DDS

Delivering Performance, Openness, and Freedom

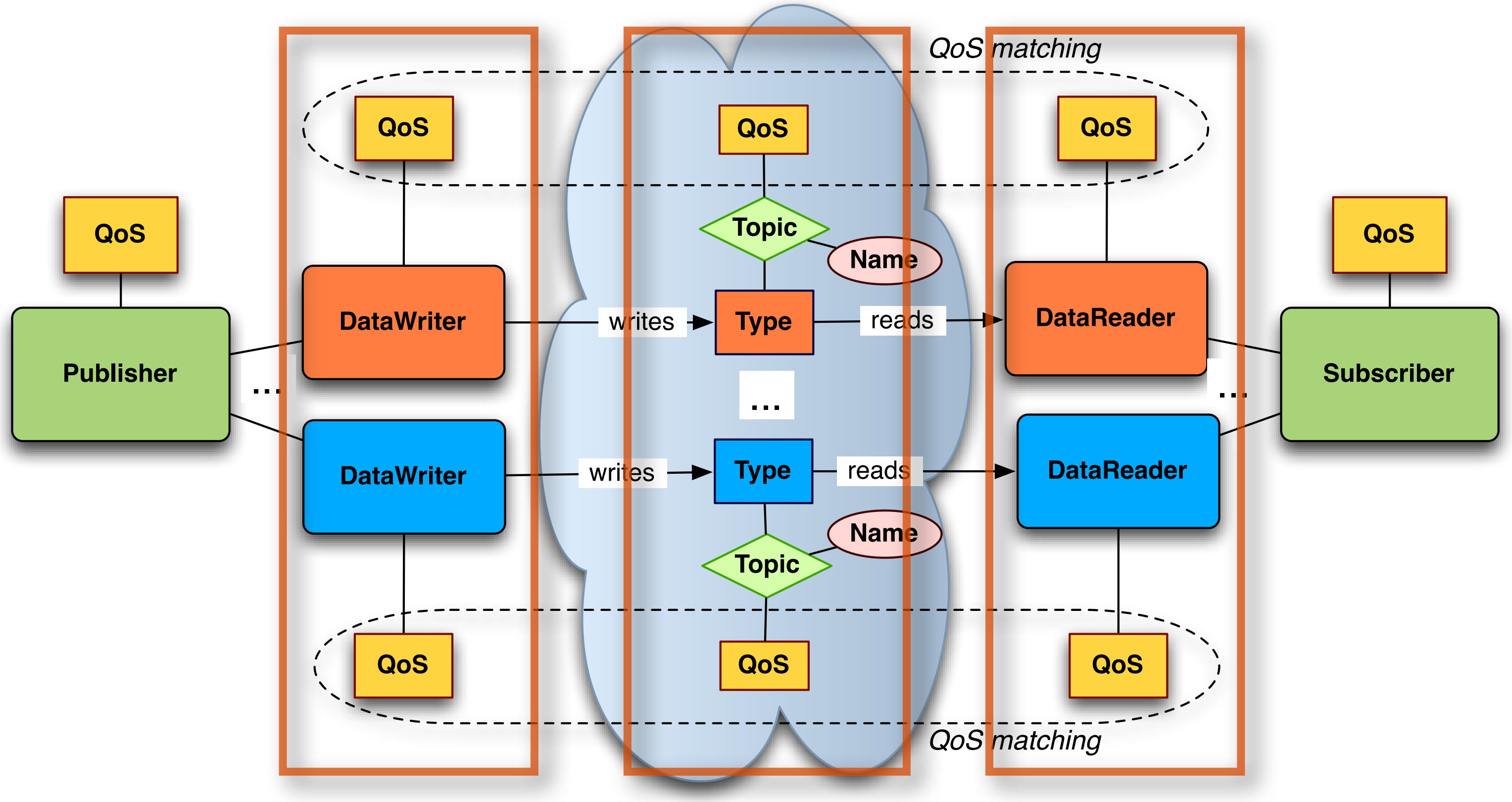
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

CONTROLLING REAL-TIME PROPERTIES

Deadline

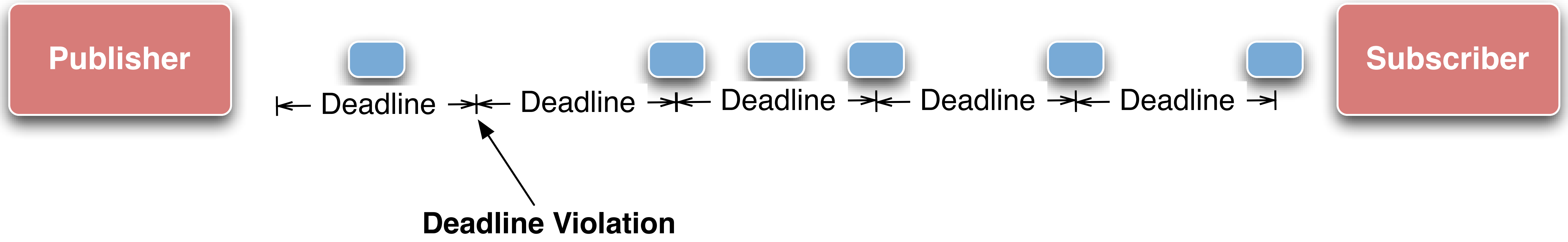
QoS Policy	Applicability	RxO	Modifiable
DEADLINE	T, DR, DW	Y	Y

The **DEADLINE** QoS policy allows to define the **maximum inter-arrival time** between data samples



Deadline QoS

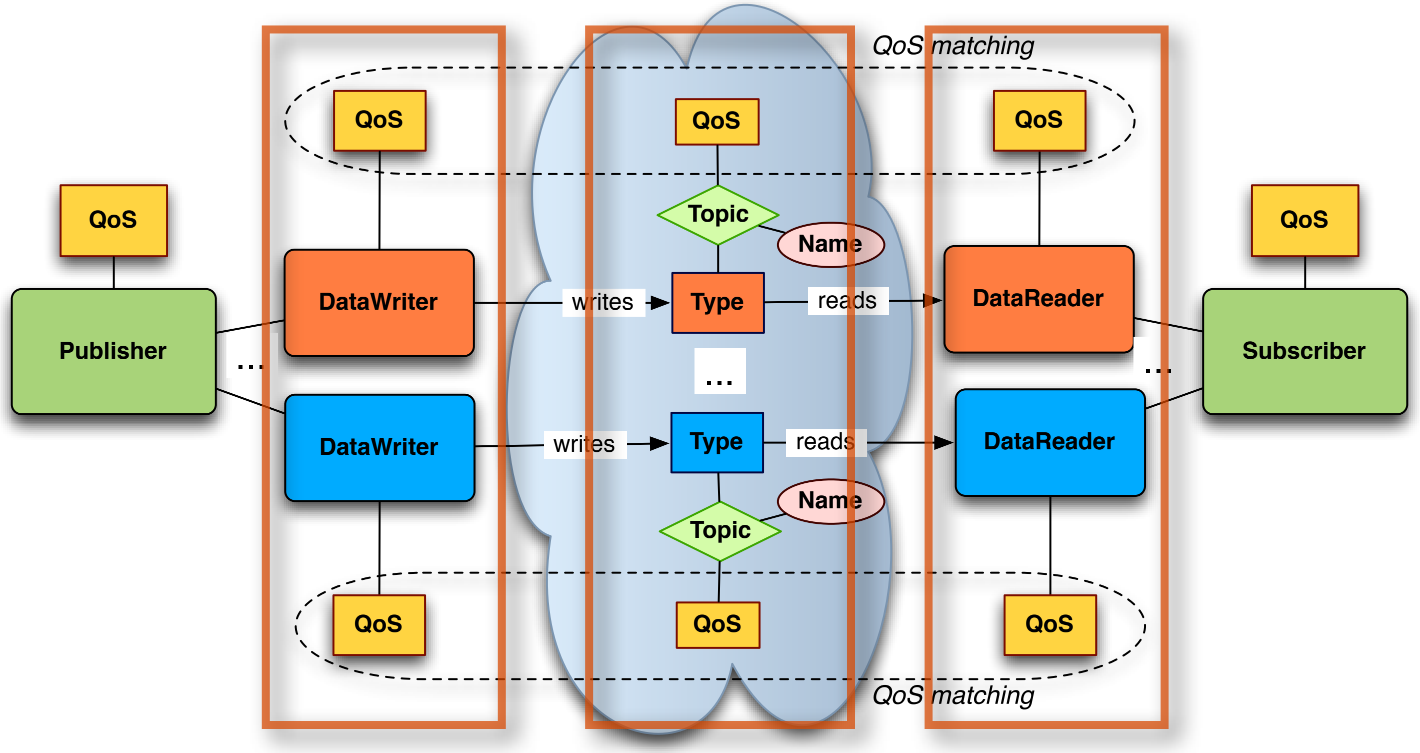
- DataWriter indicates that the application commits to write a new value at least once every deadline period
- DataReaders are notified by the DDS when the DEADLINE QoS contract is violated



Latency Budget

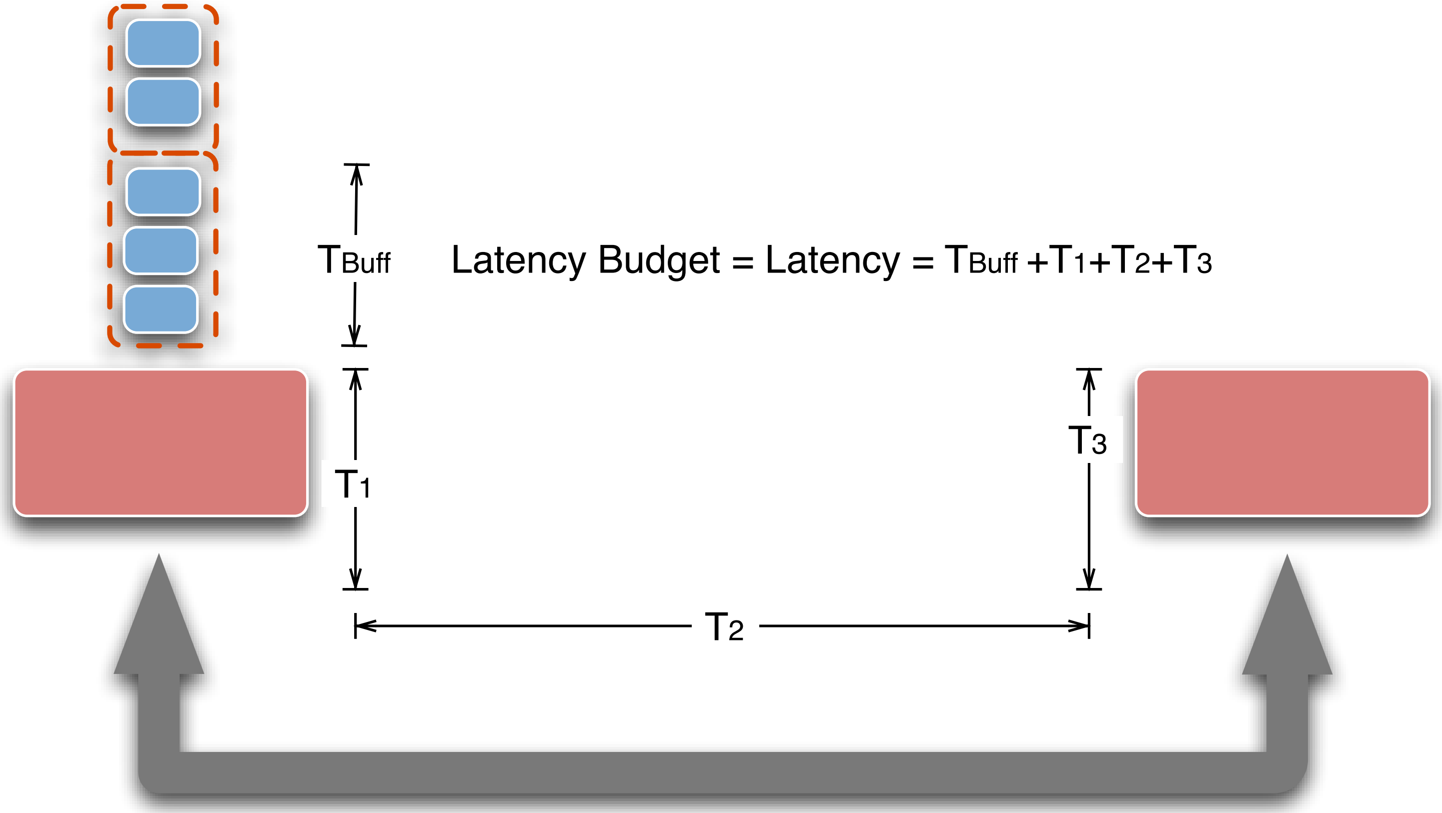
QoS Policy	Applicability	RxO	Modifiable
LATENCY BUDGET	T, DR, DW	Y	Y

The **LATENCY_BUDGET QoS** policy specifies the maximum acceptable delay from the time the data is written until the data is inserted in the receiver's application-cache



Latency Budget

- The default value of the duration is zero indicating that the delay should be minimized
- This policy is a hint to the DDS, not something that must be monitored or enforced.

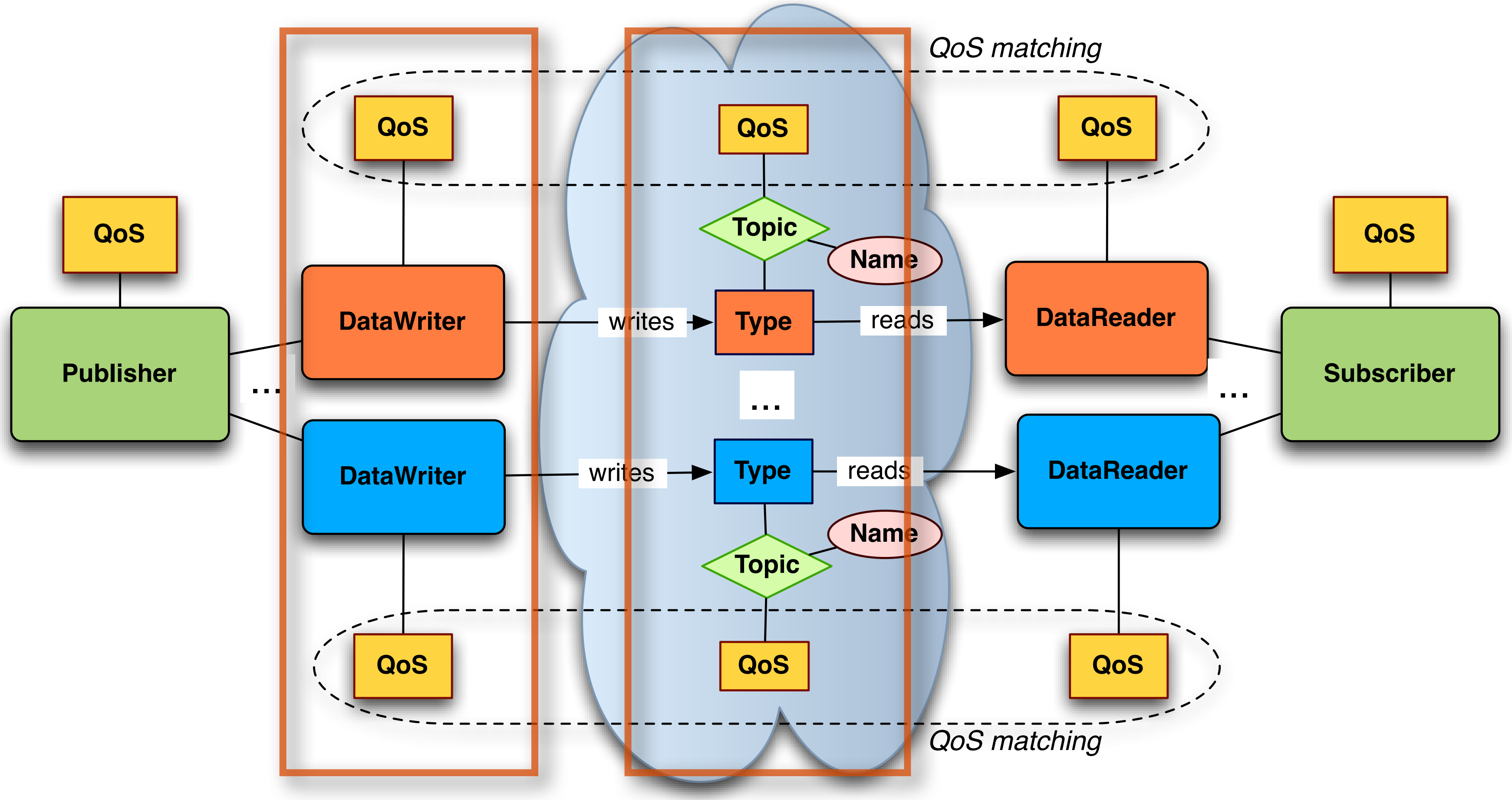


Transport Priority

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

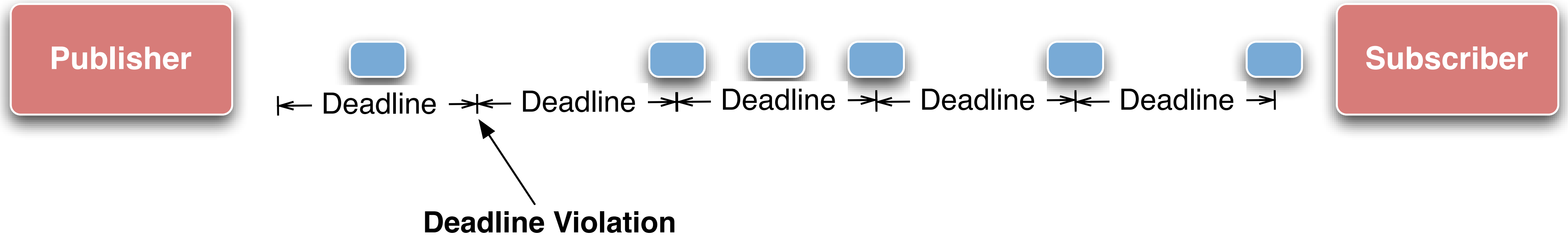
QoS Policy	Applicability	RxO	Modifiable
TRANSPORT PRIORITY	T, DW	-	Y

The **TRANSPORT_PRIORITY** QoS policy is a hint to the infrastructure as to how to set the priority of the underlying transport used to send the data.



Putting it all Together

The real-time properties with which data is delivered to applications is impacted in DDS by the following qualities of service:



- **TRANSPORT_PRIORITY**
- **LATENCY_BUDGET**
- In addition, DDS provides means for detecting performance failure, e.g., Deadline miss, by means of the **DEADLINE** QoS
- Given a periodic task-set {T} with periods Di (with $D_i < D_{i+1}$) and deadline equal to the period, than QoS should be set as follows:
  - Assign to each task T_i a TRANSPORT_PRIORITY Pi such that $P_i > P_{i+1}$
  - Set for each task T_i a DEADLINE QoS of D_i
  - For maximizing throughput and minimizing resource usage set for each Ti a LATENCY_BUDGET QoS between $D_i /2$ and $D_i/3$ (this is a rule of thumb, the upper bound is Di-(RTT/2))

OpenSplice|DDS

Delivering Performance, Openness, and Freedom

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

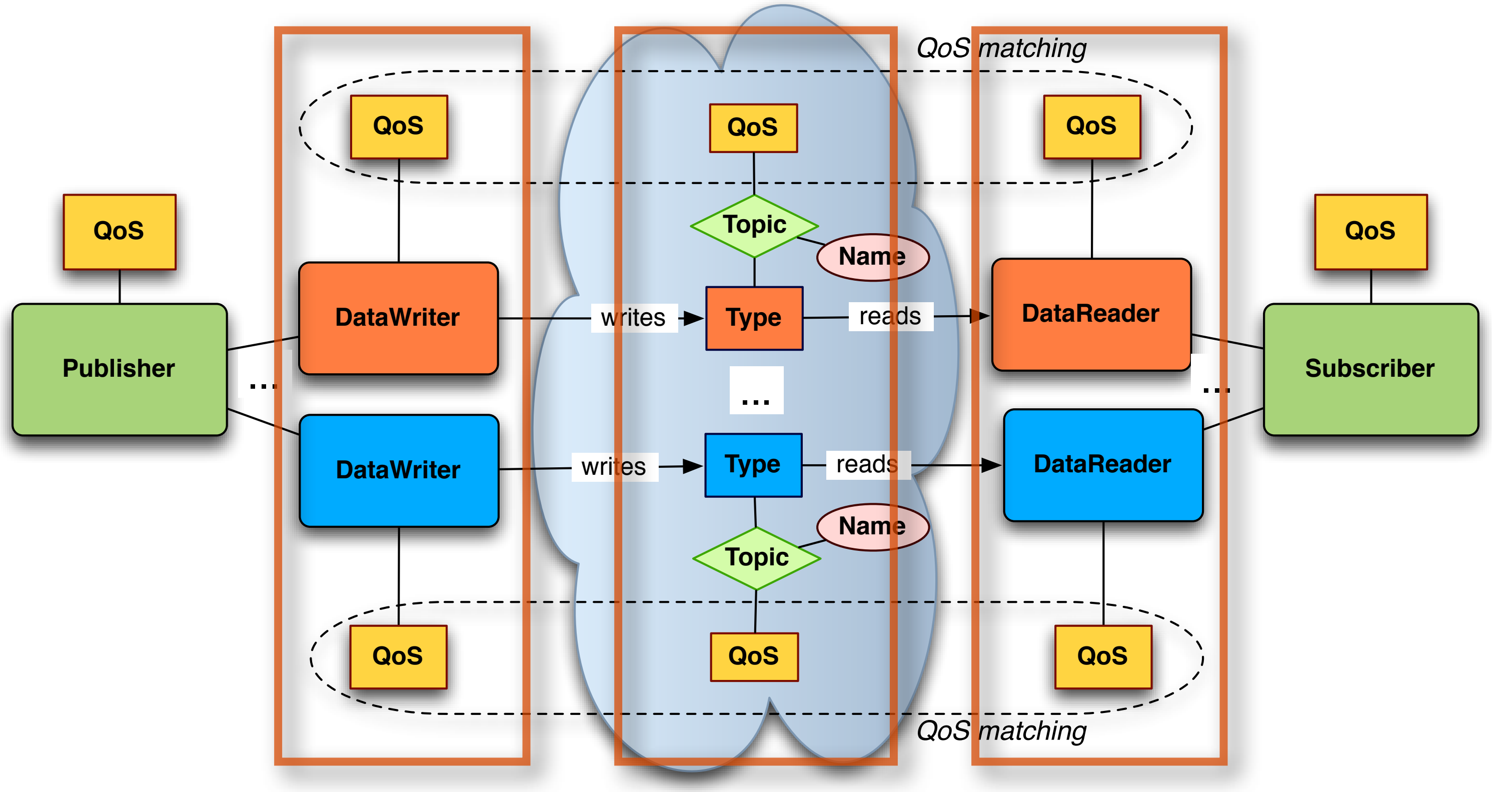
Controlling the Consistency Model

Durability

The DURABILITY QoS controls the data availability w.r.t. late joiners, specifically the DDS provides the following variants:

- **Volatile.** No need to keep data instances for late joining data readers
- **Transient Local.** Data instance availability for late joining data reader is tied to the data writer availability
- **Transient.** Data instance availability outlives the data writer
- **Persistent.** Data instance availability outlives system restarts

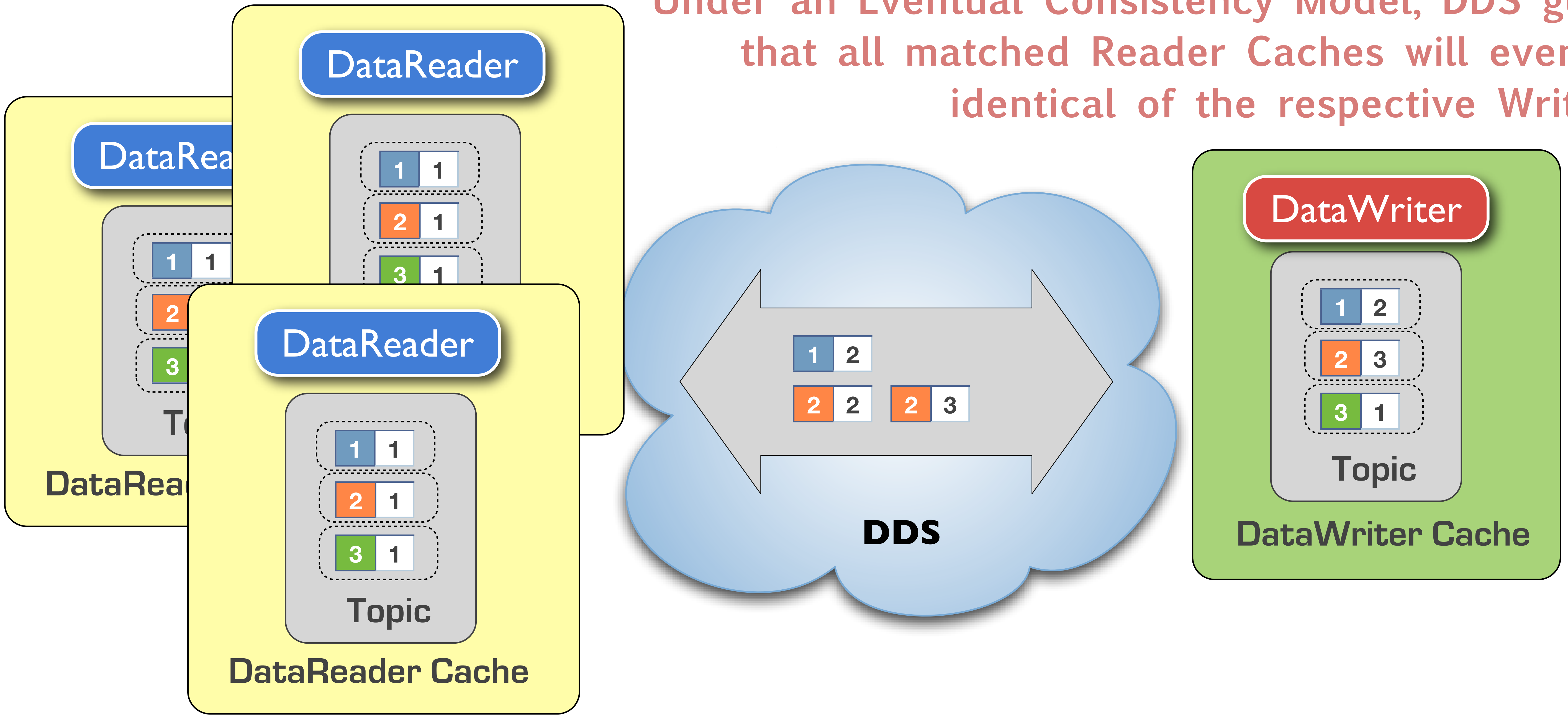
QoS Policy	Applicability	RxO	Modifiable
DURABILITY	T, DR, DW	Y	N
DURABILITY SERVICE	T, DW	N	N



Eventual Consistency & R/W Caches

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

Under an Eventual Consistency Model, DDS guarantees that all matched Reader Caches will eventually be identical of the respective Writer Cache



QoS & Consistency Model

The DDS Consistency Model is a property that can be associated to Topics or further refined by Reader/Writers. The property is controlled by the following QoS Policies:

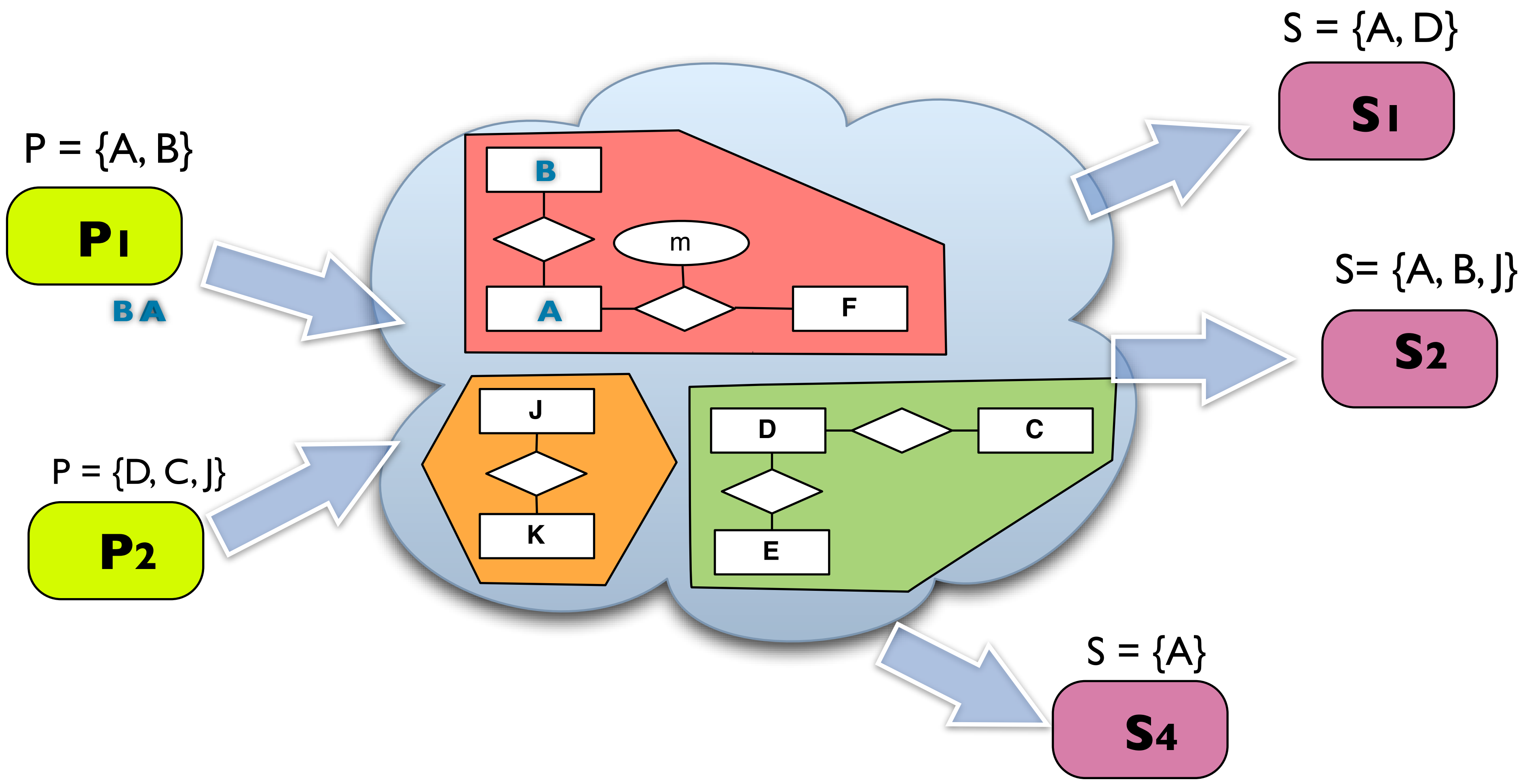
- **DURABILITY**
  - VOLATILE | TRANSIENT_LOCAL | TRANSIENT | PERSISTENT
- **LIFESPAN**
- **RELIABILITY**
  - RELIABLE | BEST_EFFORT
- **DESTINATION ORDER**
  - SOURCE_TIMESTAMP | DESTINATION_TIMESTAMP

QoS Policy	Applicability	RxO	Modifiable
DURABILITY	T, DR, DW	Y	N
LIFESPAN	T, DW	-	Y
RELIABILITY	T, DR, DW	Y	N
DESTINATION ORDER	T, DR, DW	Y	N

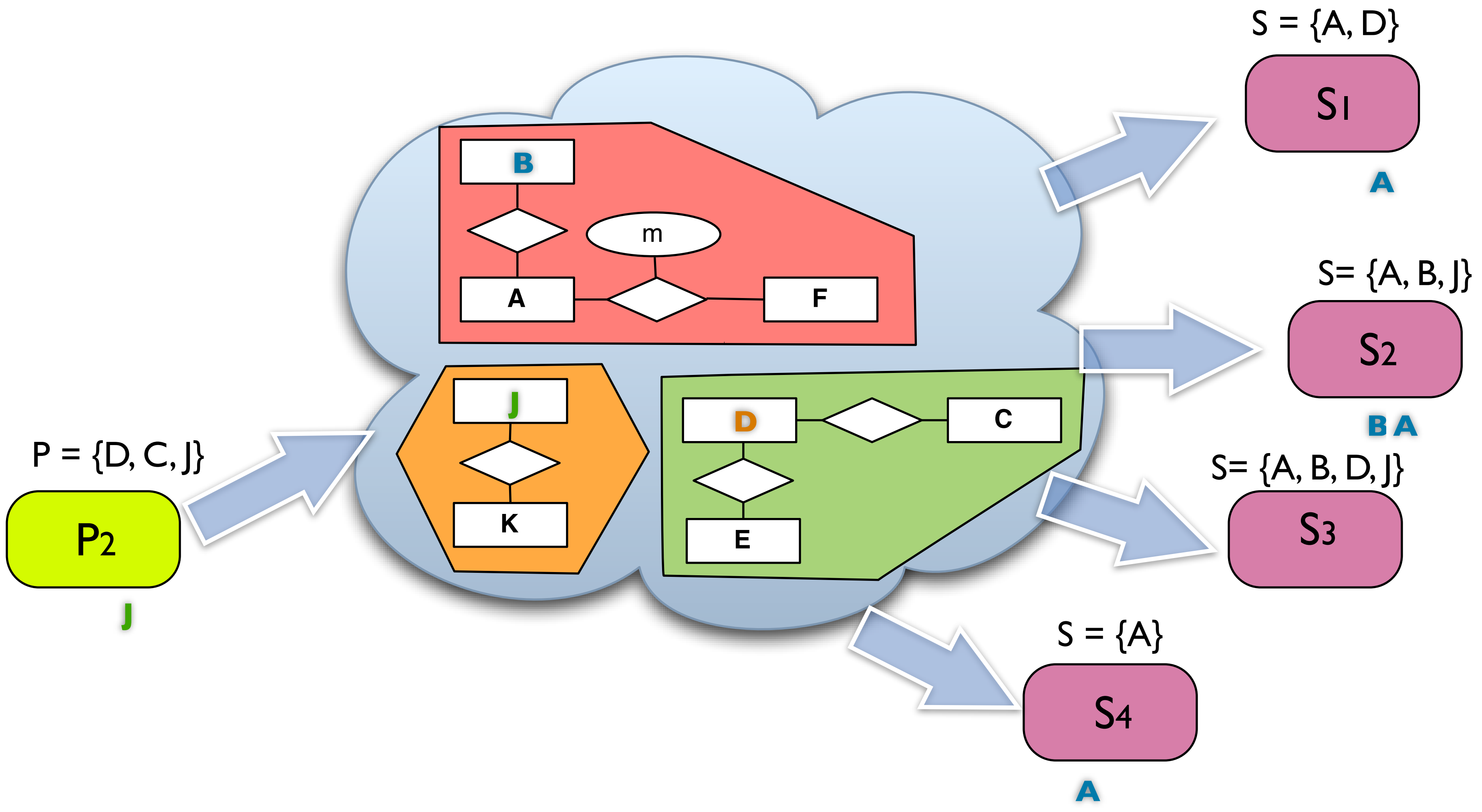
QoS & Consistency Model

	DURABILITY	RELIABILITY	DESTINATION_ORDER	LIFESPAN
Eventual Consistency (No Crash / Recovery)	VOLATILE	RELIABLE	SOURCE_TIMESTAMP	INF.
Eventual Consistency (Reader Crash / Recovery)	TRANSIENT_LOCAL	RELIABLE	SOURCE_TIMESTAMP	INF.
Eventual Consistency (Crash/Recovery)	TRANSIENT	RELIABLE	SOURCE_TIMESTAMP	INF.
Eventual Consistency (Crash/Recovery)	PERSISTENT	RELIABLE	SOURCE_TIMESTAMP	INF.
Weak Consistency	ANY	ANY	DESTINATION_TIMESTAMP	ANY
Weak Consistency	ANY	BEST_EFFORT	ANY	ANY
Weak Consistency	ANY	ANY	ANY	N

	DURABILITY	RELIABILITY	DESTINATION_ORDER	LIFESPAN	
Eventual Consistency (Reader Crash / Recovery)	TRANSIENT_LOCAL	RELIABLE	SOURCE_TIMESTAMP	INF.	{A}
Eventual Consistency (Crash/Recovery)	TRANSIENT	RELIABLE	SOURCE_TIMESTAMP	INF.	{B}
Weak Consistency	ANY	ANY	ANY	N	{J}



	DURABILITY	RELIABILITY	DESTINATION_ORDER	LIFESPAN	
Eventual Consistency (Reader Crash / Recovery)	TRANSIENT_LOCAL	RELIABLE	SOURCE_TIMESTAMP	INF.	{A}
Eventual Consistency (Crash/Recovery)	TRANSIENT	RELIABLE	SOURCE_TIMESTAMP	INF.	{B}
Weak Consistency	ANY	ANY	ANY	N	{J}



OpenSplice|DDS

Delivering Performance, Openness, and Freedom

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

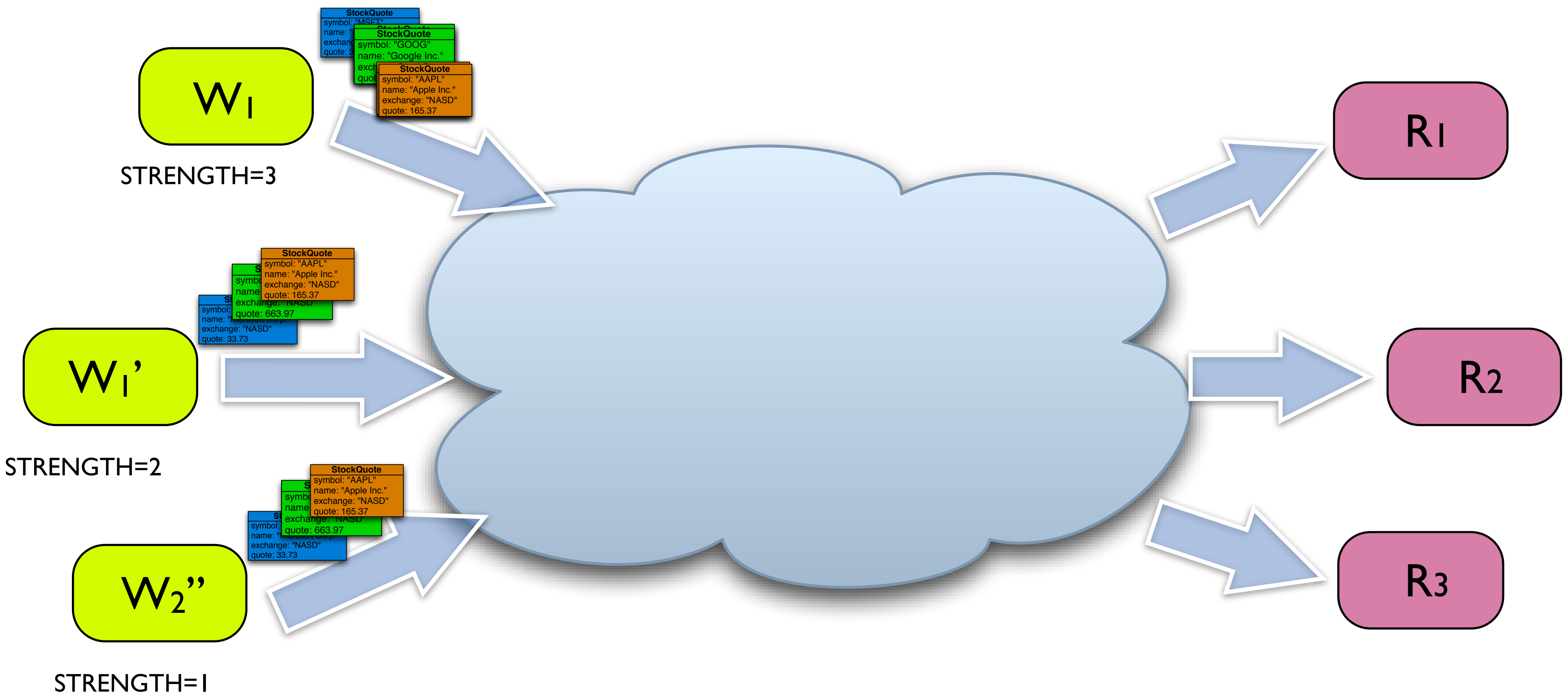
CONTROLLING REPLICATION

Availability

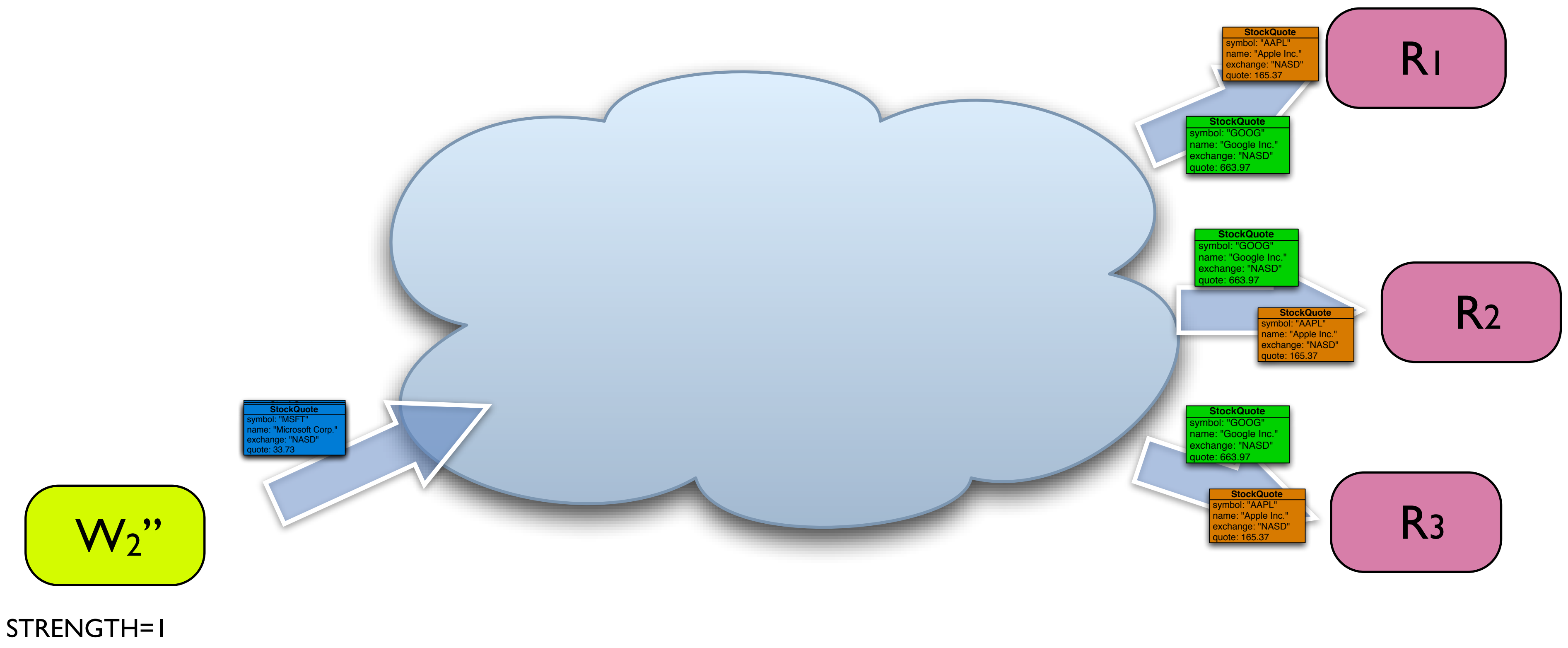
Availability of data producers can be controlled via two QoS Policies

- **OWNERSHIP**
- **OWNERSHIP STRENGTH**
- Instances of exclusively owned Topics can be modified (are owned) by the higher strength writer
- Writer strength is used to coordinate replicated writers

Availability



Availability



OpenSplice|DDS

Delivering Performance, Openness, and Freedom

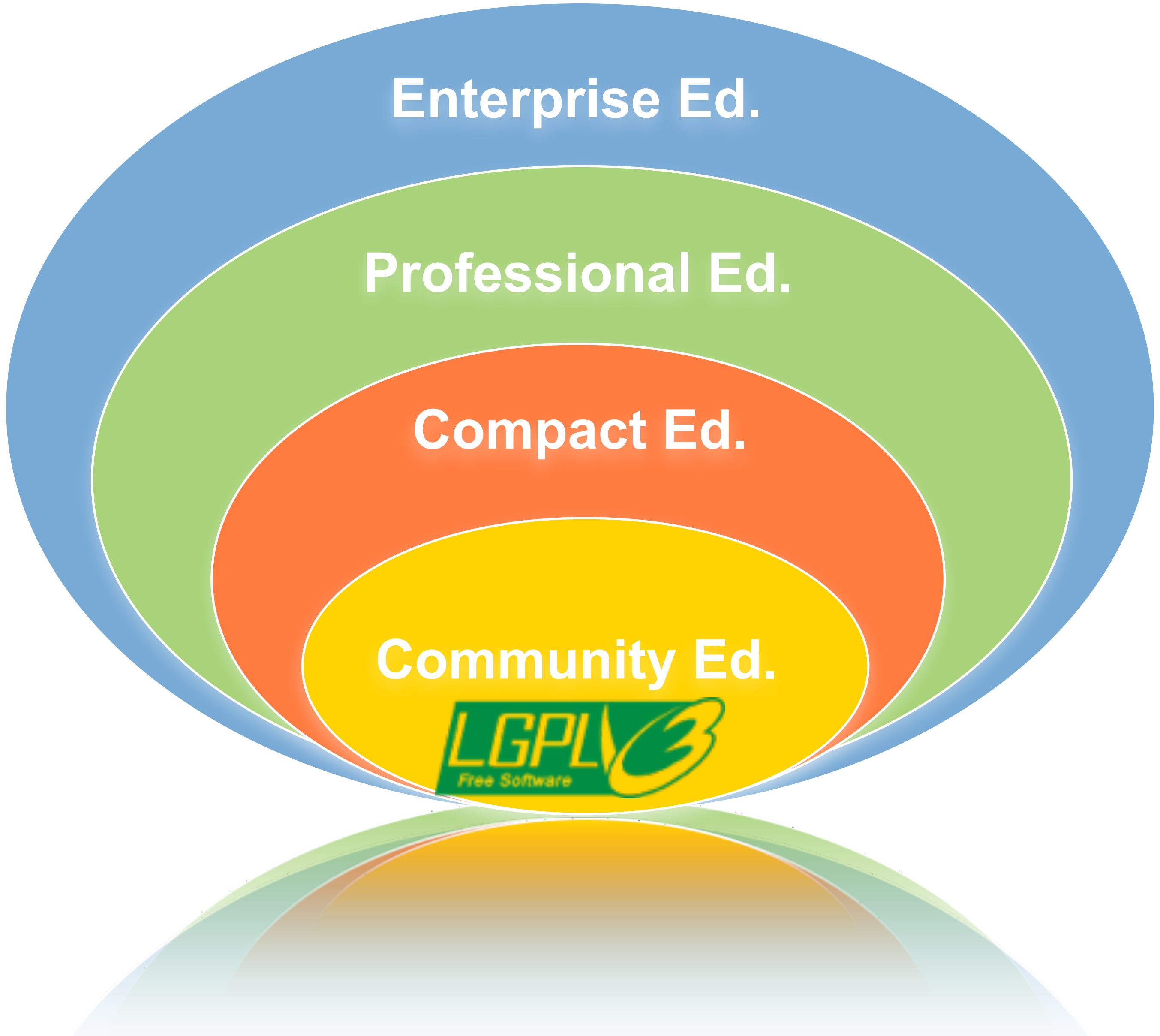
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

OPENSPLICE DDS

OpenSplice DDS v5.1

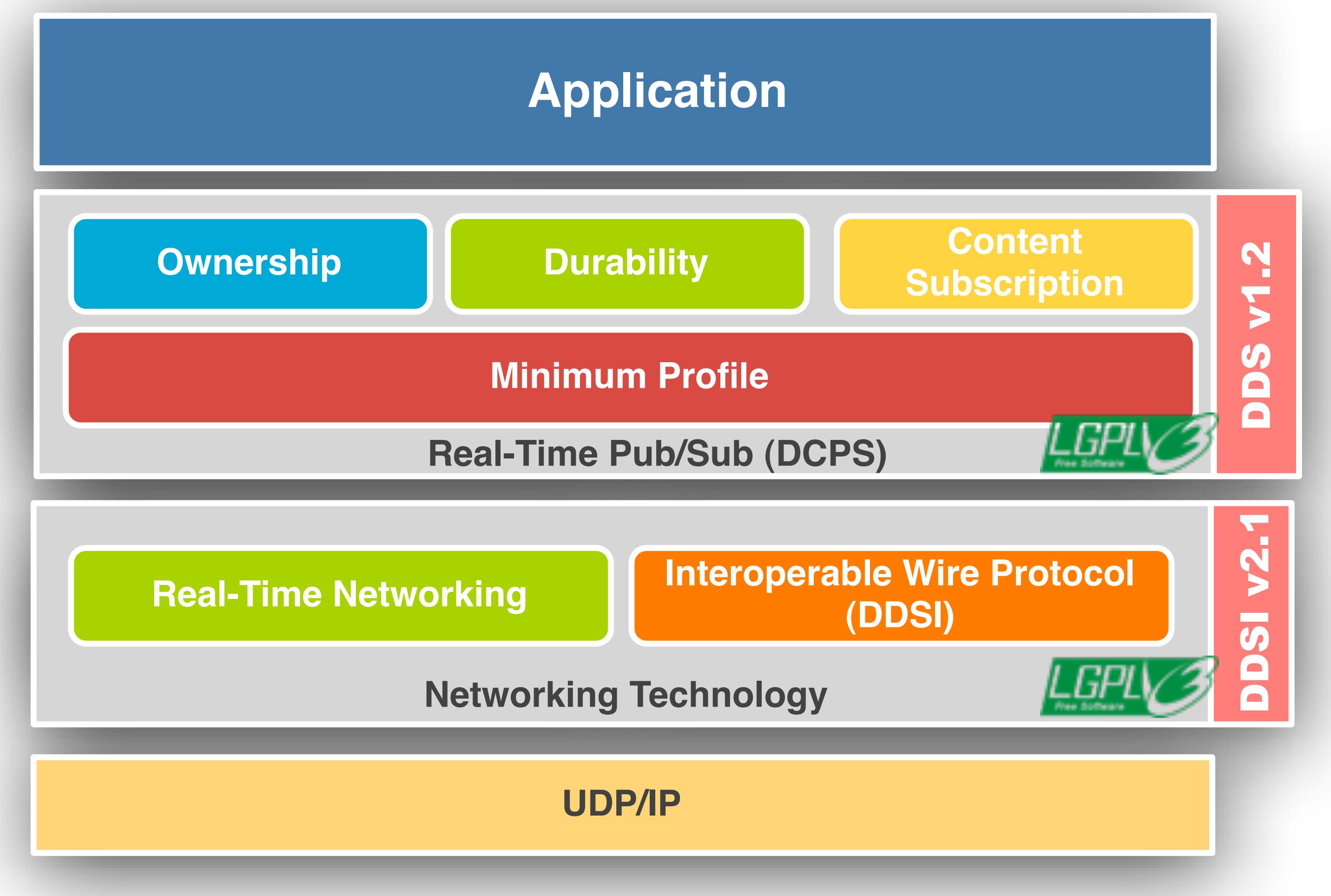
- Four different editions providing a growing set of functionalities:
  - Community Edition
  - Compact Edition
  - Professional Edition
  - Enterprise Edition

- The Community Edition is Open Source. Compact, Enterprise and Professional Edition are available only through Commercial Subscriptions



Community Edition

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control



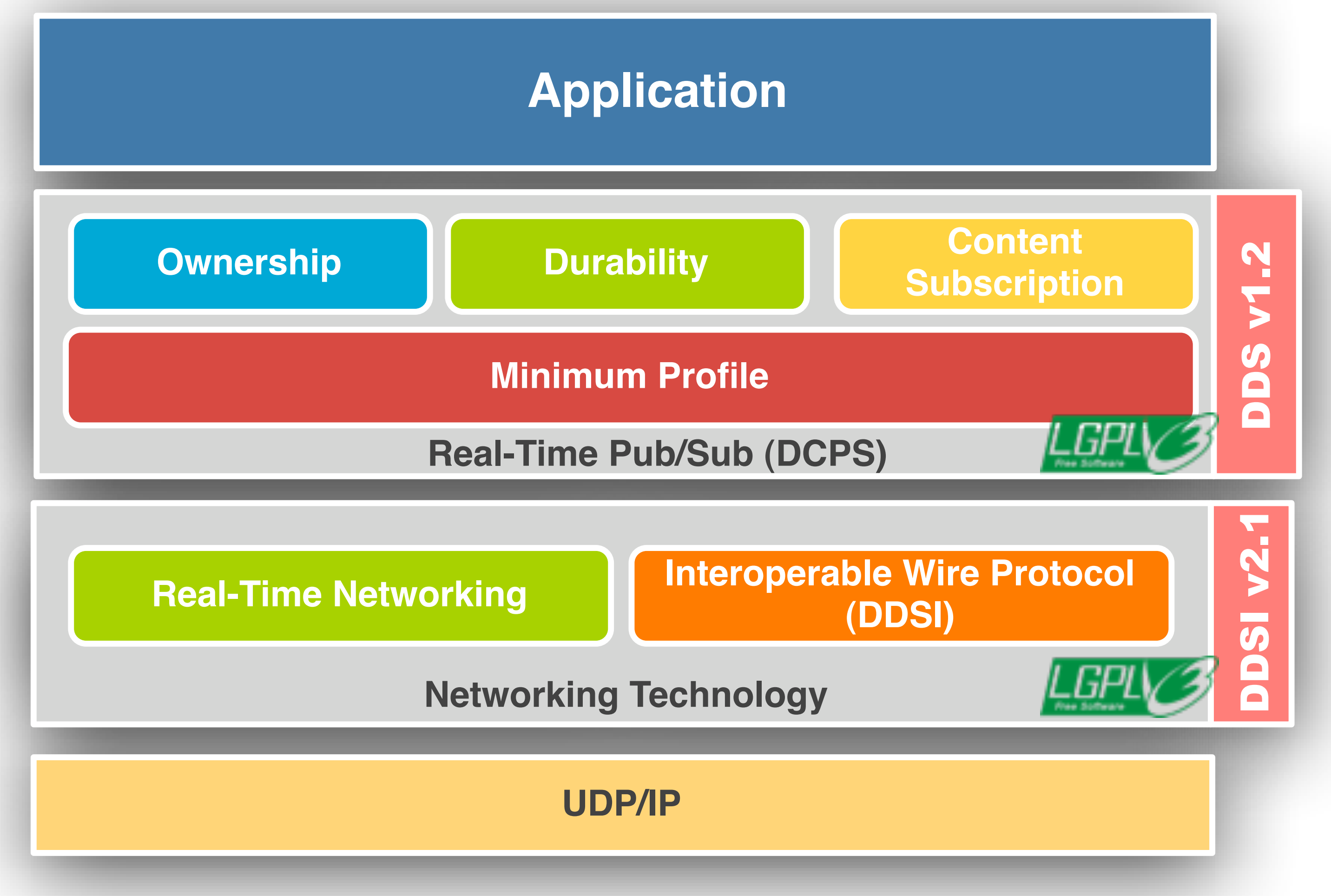
Licensing

- Open Source (LGPLv3)

Community Edition

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control



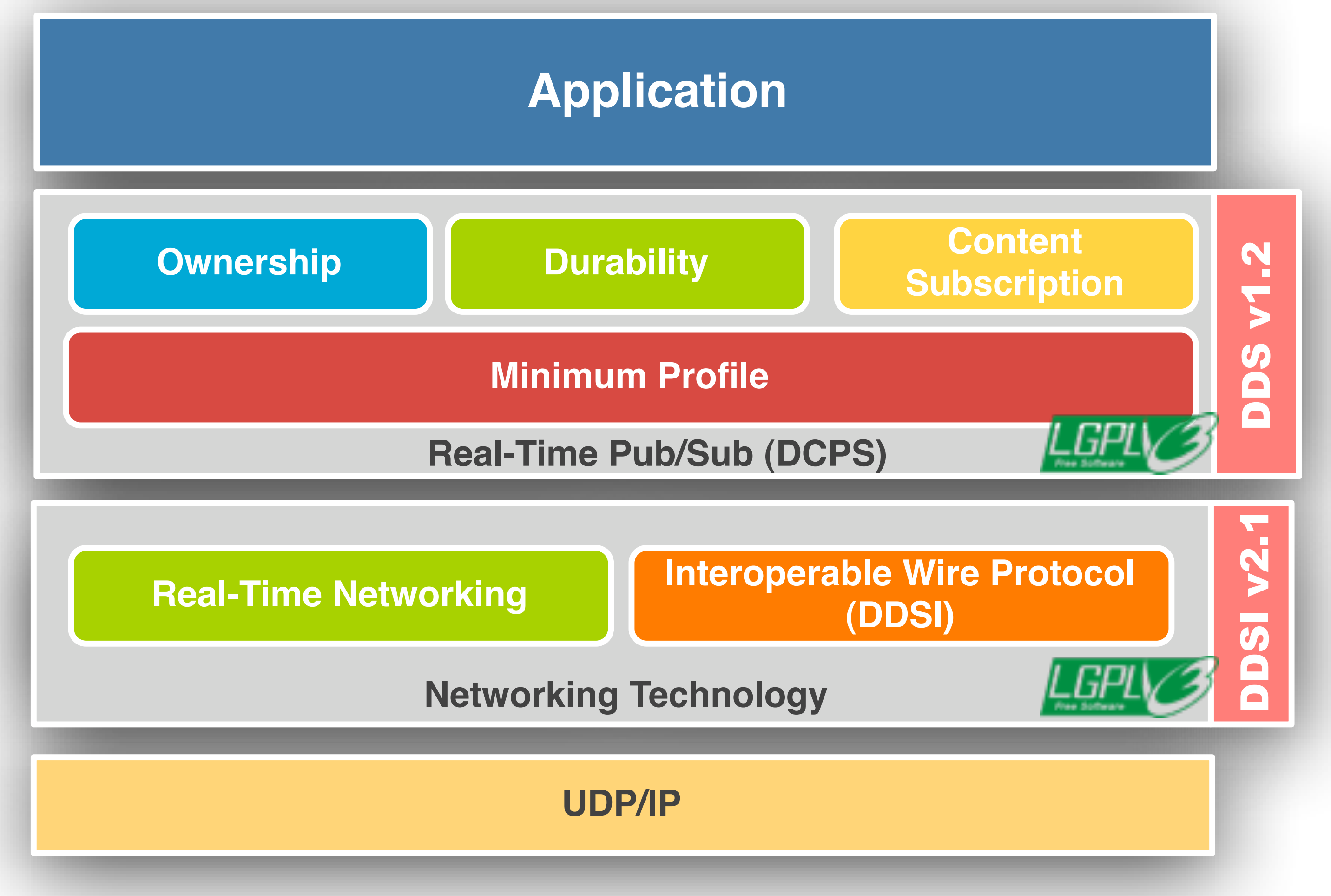
Licensing

- Open Source (LGPLv3)

Community Edition

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control

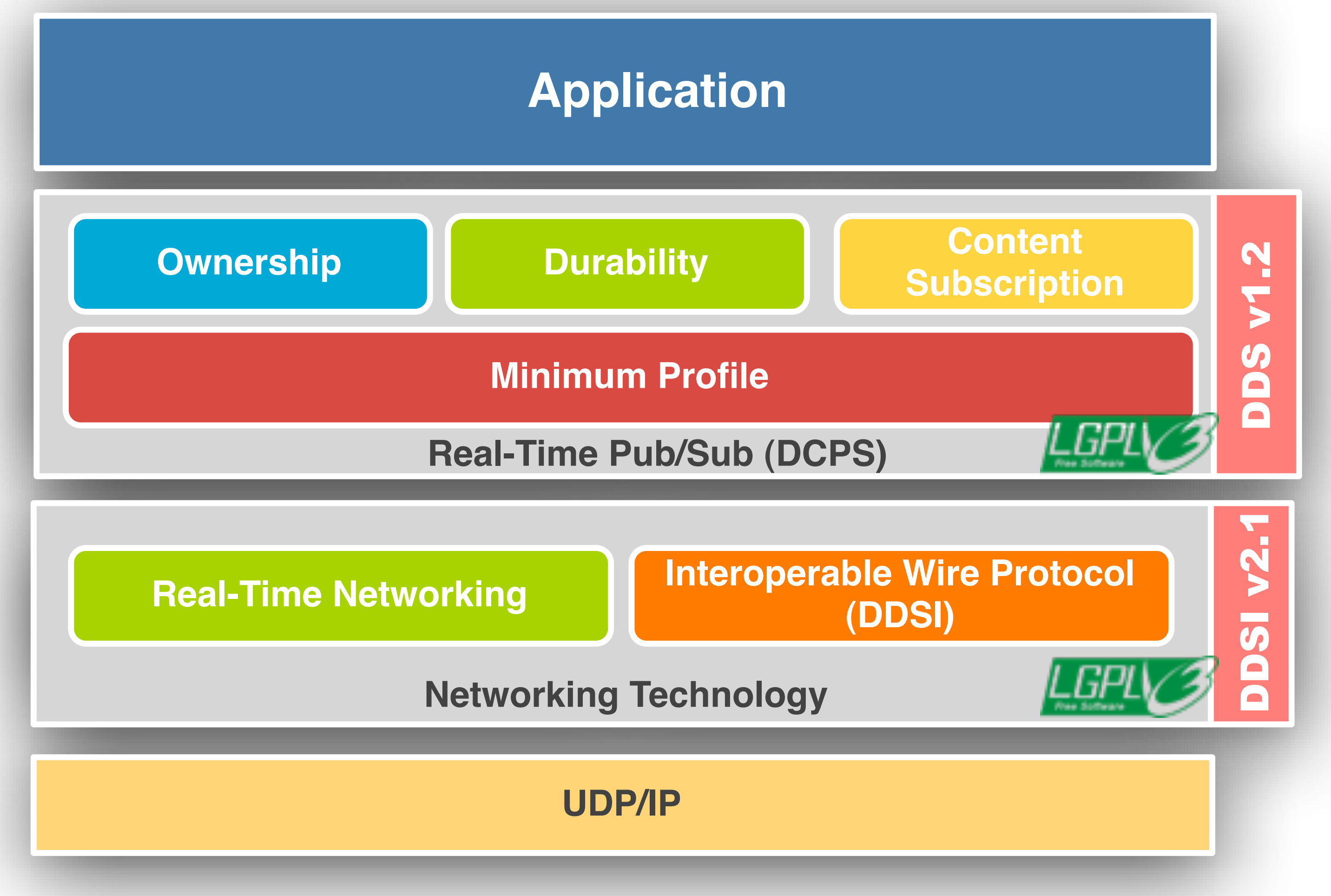


Licensing

- Open Source (LGPLv3)

Community Edition

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control



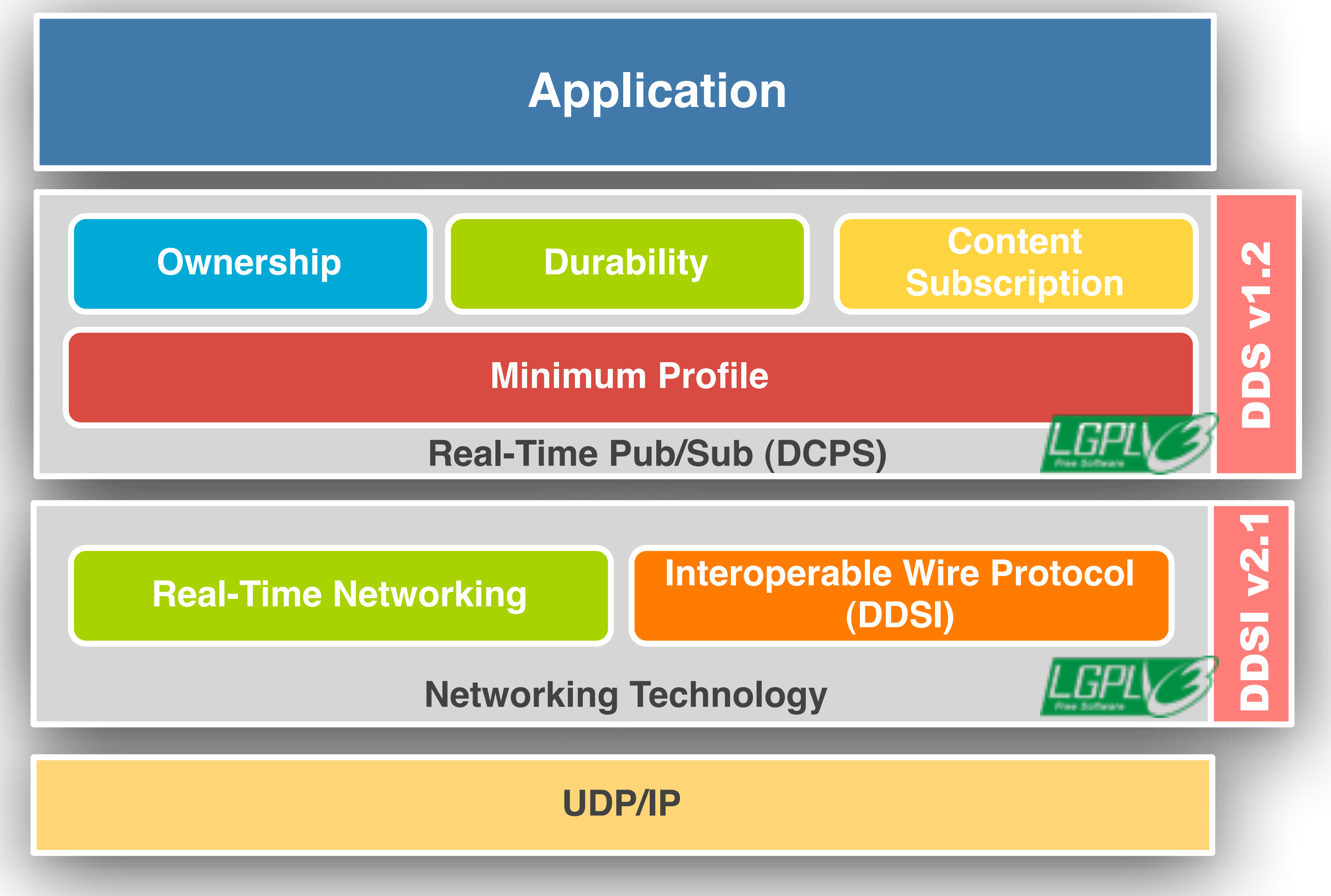
Licensing

- Open Source (LGPLv3)

Community Edition

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control



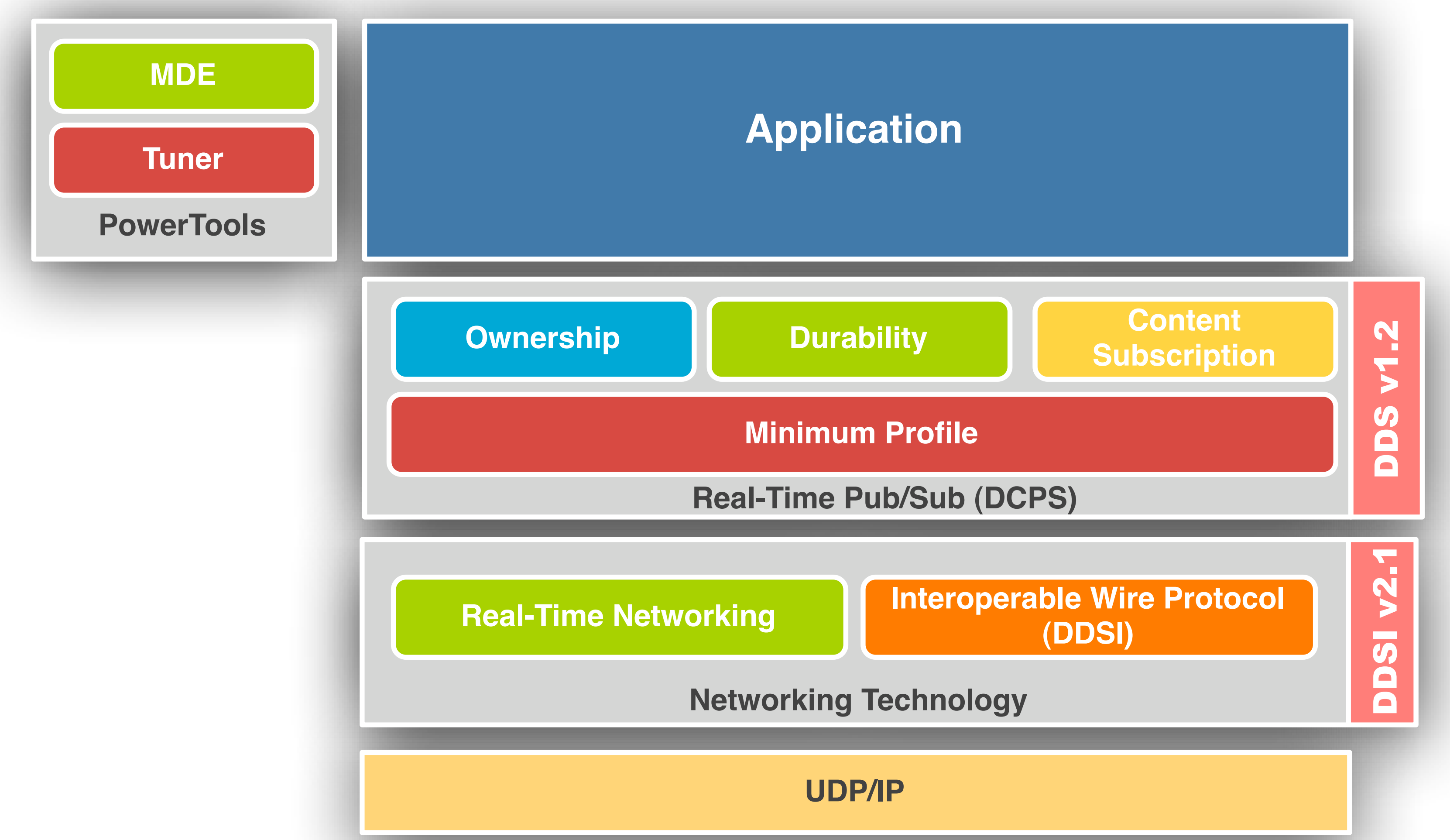
Licensing

- Open Source (LGPLv3)

Compact Edition

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

- All Community ed Features
- Power Tools
  - Eclipse Visual Modeling Tool
  - Tuner Tool



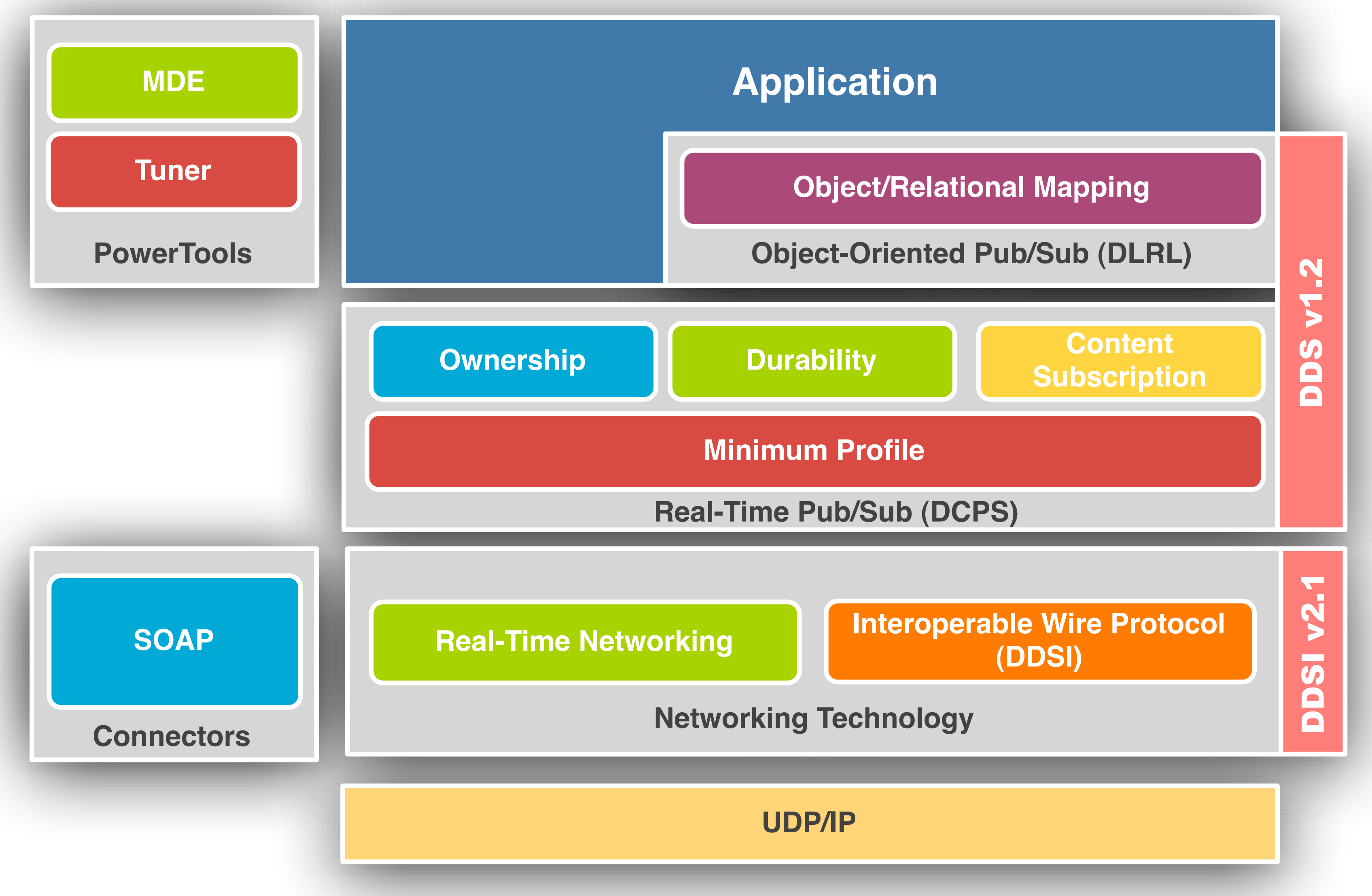
Licensing

- Commercial Subscription / PrismTech Source Code License (non-copy-left)

Professional Edition

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

- All Compact Ed. Features
- DLRL v1.2
- **Power Tools**
  - Eclipse Visual Modeling Tool
  - Tuner Tool
- Connectors
  - SOAP



Licensing

- Commercial Subscription / PrismTech Source Code License (non-copy-left)

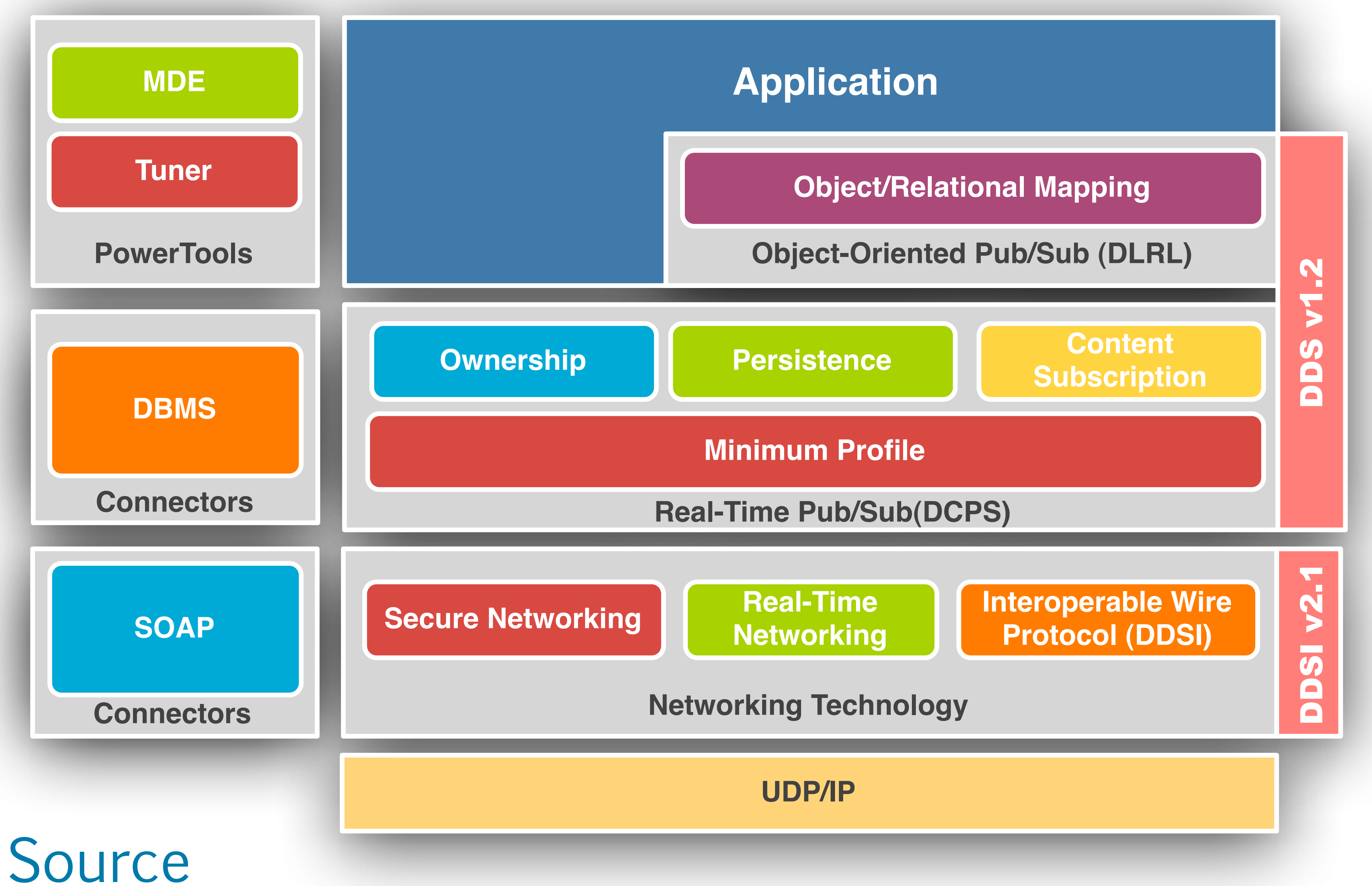
Enterprise Edition

Features

- All Professional Ed. Features
- Security
- Connectors
  - DBMS

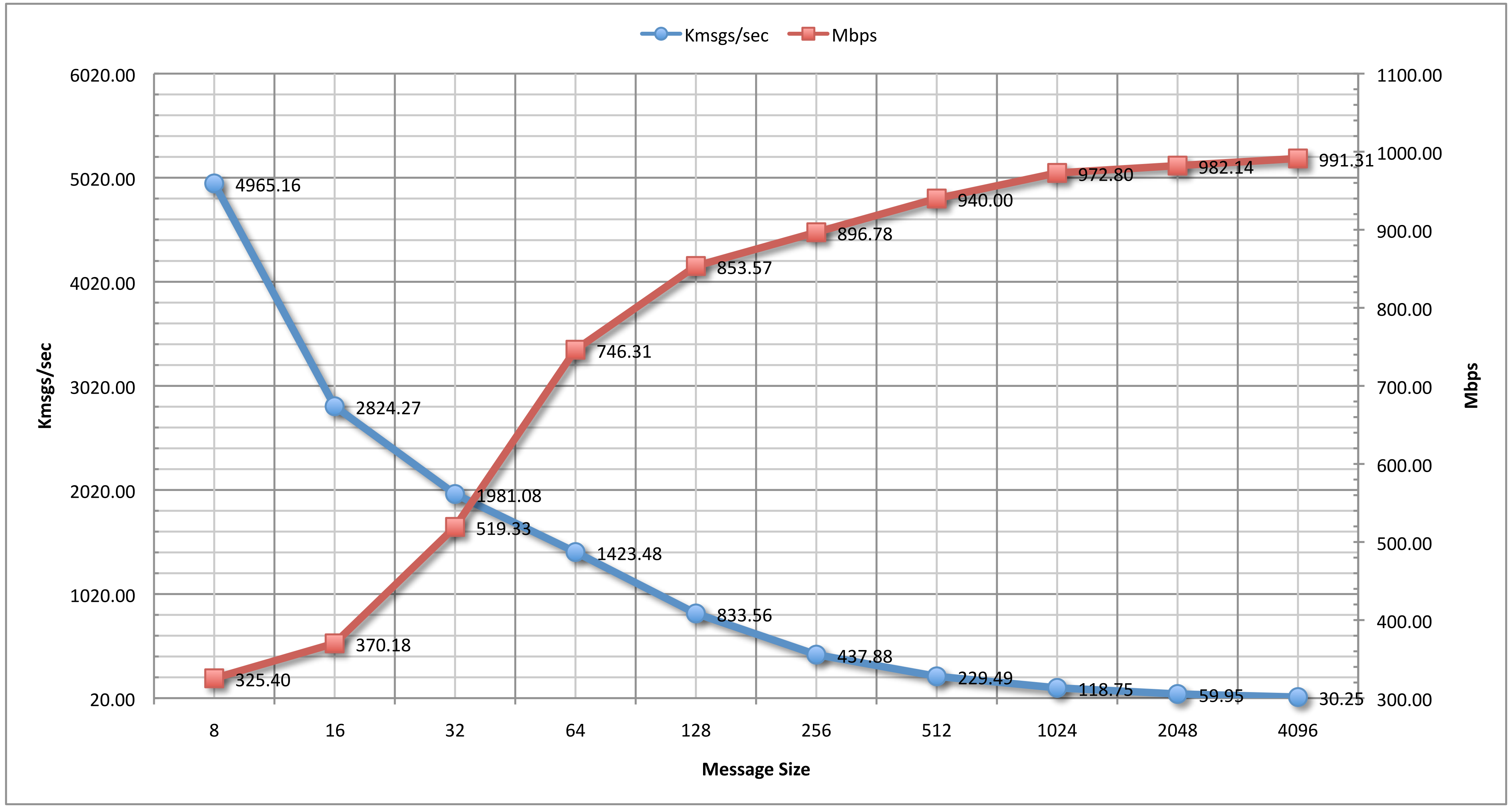
Licensing

- Commercial Subscription / PrismTech Source Code License (non-copy-left)



MDE
Tuner
PowerTools
DBMS
Connectors
SOAP
Connectors
Application
Object/Relational Mapping
Object-Oriented Pub/Sub (DLRL)
Ownership
Persistence
Content Subscription
Minimum Profile
Real-Time Pub/Sub(DCPS)
DDS v1.2
Secure Networking
Real-Time Networking
Interoperable Wire Protocol (DDSI)
Networking Technology
DDSI v2.1
UDP/IP

Throughput



Inter-Node Latency

- 60 usec

Inter-Core Read-Latency

- 2 usec

Inter-Core Latency

- <10 usec

Test Scenario

- Single Threaded Application
- 8192 bit message batches

HW:

- Dell blade-server
- Dual-core, Dual-CPU, AMD Opteron 2.4 Ghz

OS

- Linux 2.6.21-1.3194.fc7

Network

- Gigabit Ethernet cards
- Dell PowerConnect 5324 switch

OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

USE CASES

TACTICOS CMS

- OpenSplice DDS enables the core infrastructure of the TACTICOS Naval Combat Management System from THALES Naval Netherlands'

- OpenSplice DDS provide TACTICOS with is renown high-availability, reconfigurability, and scalability which make it possible to scale from small ships to carrier-grade ships

- TACTICOS numbers are today:
  - Deployed on 26 ships classes, >100 ships
  - 2.000+ deployed runtimes (running on Solaris-servers, Linux-consoles, & vxWorks embedded subsystems)
  - 15 Navies worldwide (Netherlands, Germany, Turkey, Greece, Oman, Qatar, Portugal, South Korea, Japan, Poland,...)

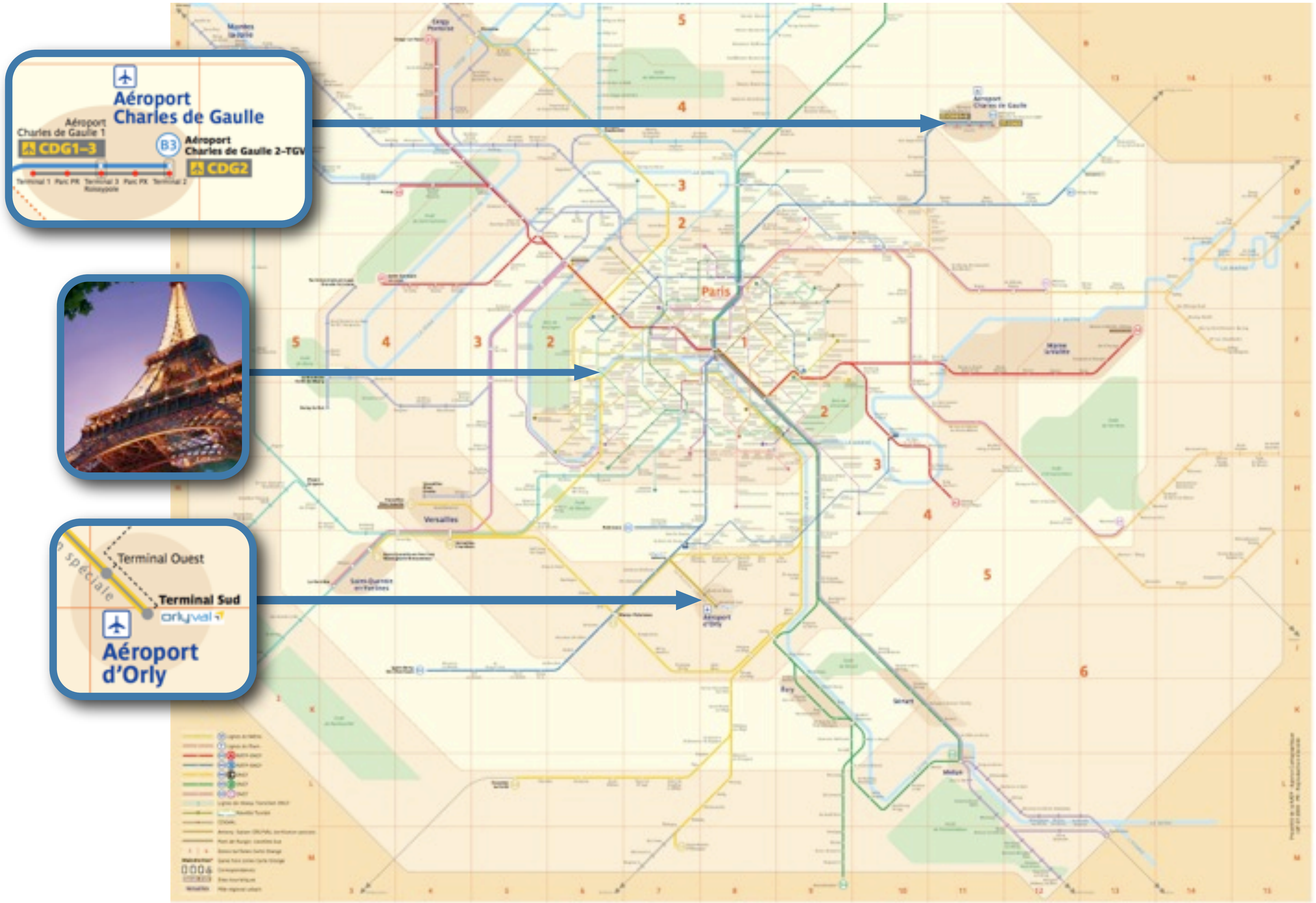
Vetronics

- OpenSplice DDS adopted as the foundation for the electronic architecture of next generation Vehicle Architecture
- OpenSplice DDS takes care of distributing real-time sensor data for all sources but video
- An OpenSplice DDS Remote Method Invocation extension is also used to issues all commands

Regie Autonome des Transports Parisiens

Transport Parisien

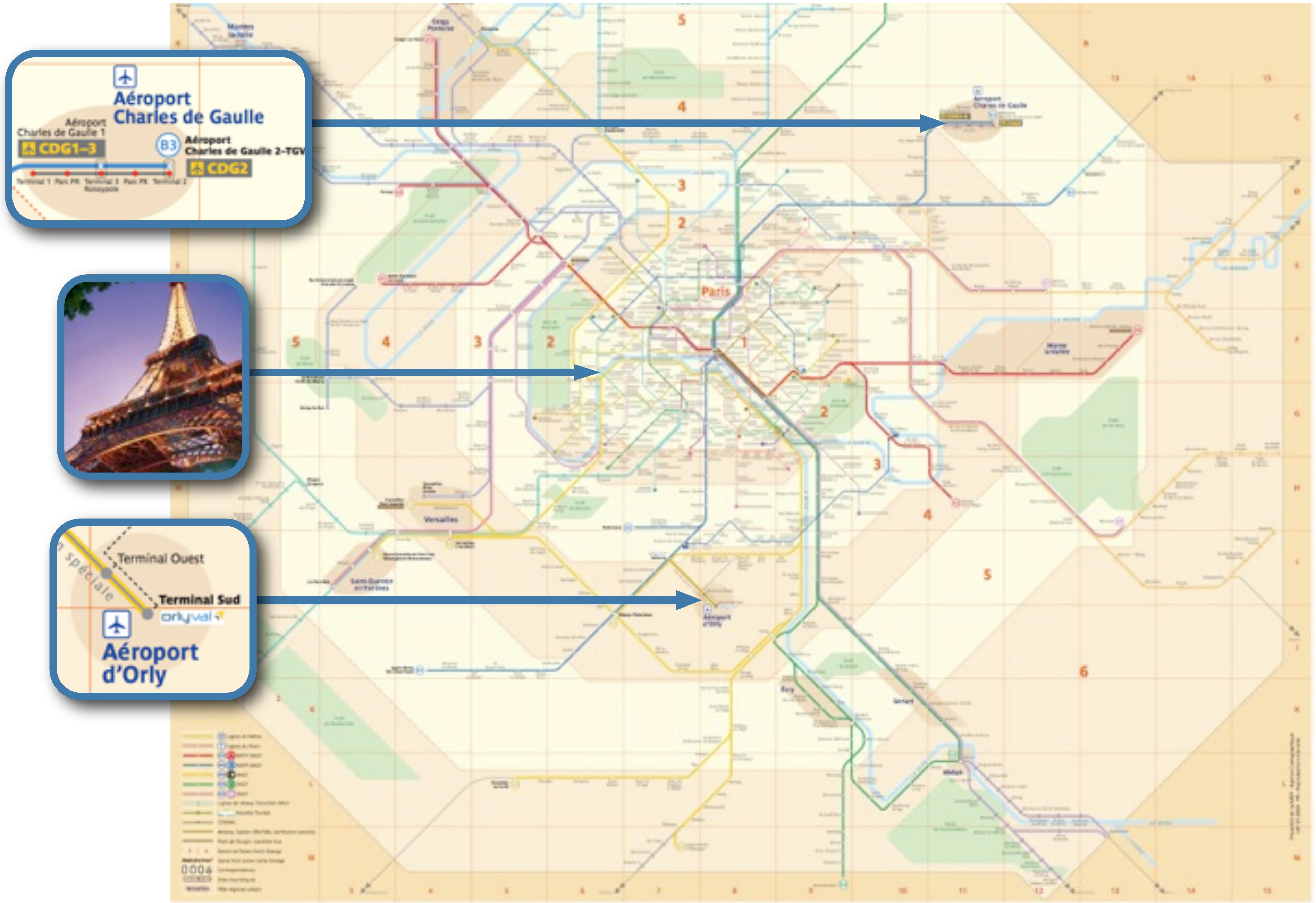
- Manage and Supervise the equipment deployed in the stations of Paris Metro/RER/ Bus/Tram


Aéroport Charles de Gaulle
Aéroport Charles de Gaulle 1
CDG1–3
B3
Aéroport Charles de Gaulle 2-TGV
CDG2
Terminal Ouest
Terminal Sud
orlyval
Aéroport d'Orly
Paris
Versailles

Regie Autonome des Transports Parisiens

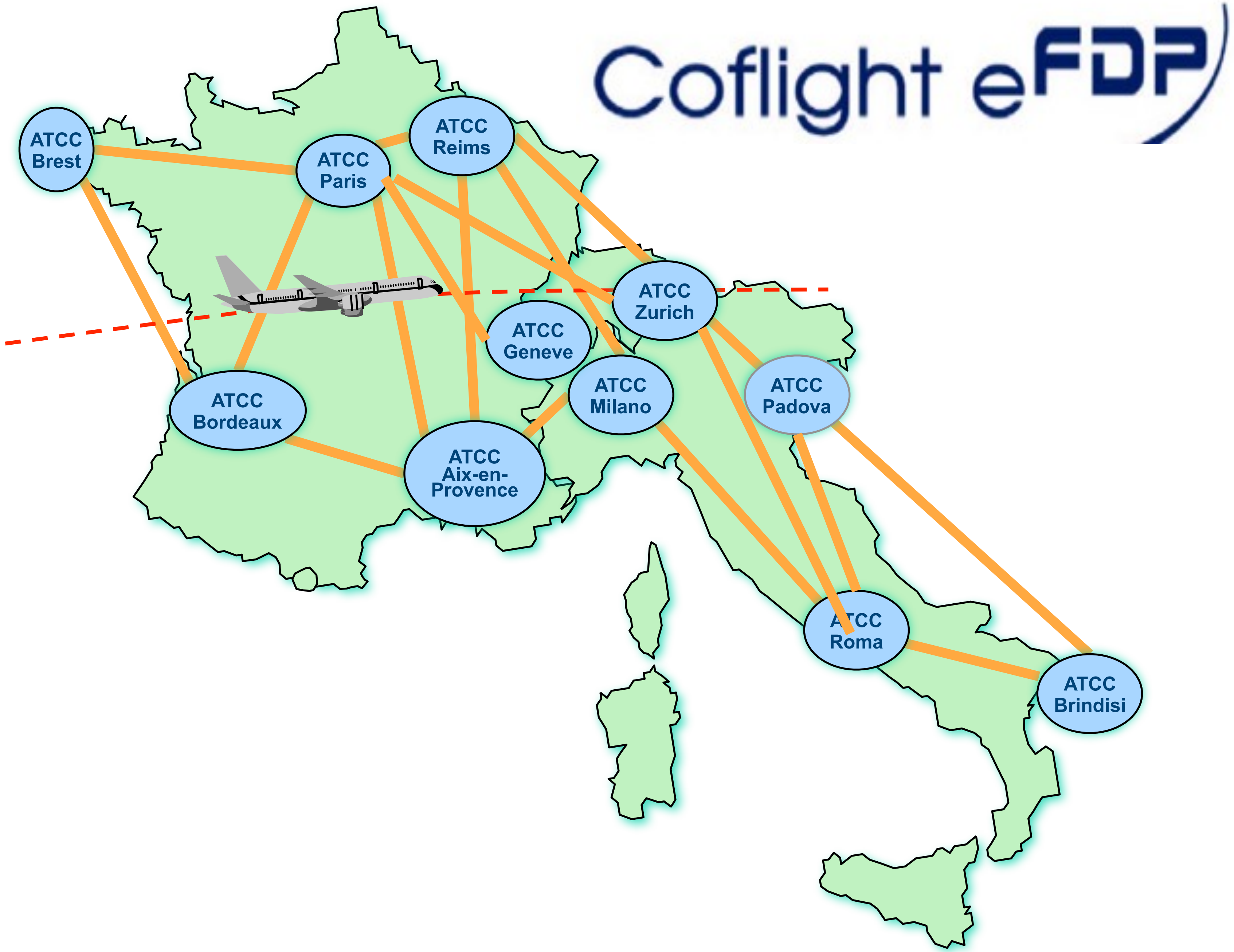
Some Numbers

- 214 Km of Metro lines with over 300 stations
- 115 Km of rapid transit network (RER)
- 30.2 Km of Tramway lines and over 569 Km of bus lines
- Several millions of passengers per day
- 50000 Supervised Equipments



EU Flight Data Processor

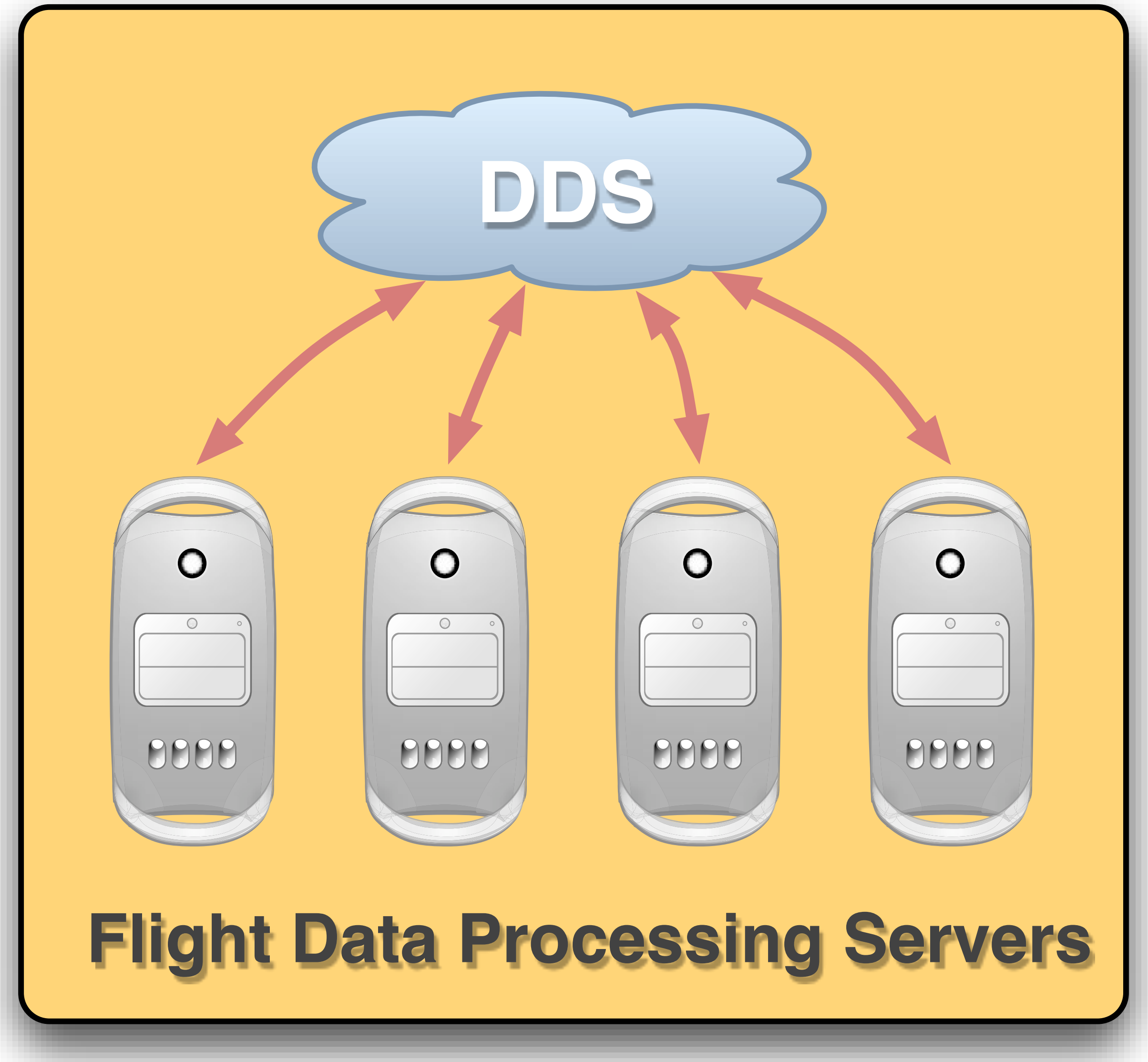
- Large program to replace existing Flight Data Processors (FDPs)
  - 5 Centers in France
  - 4 Centers in Italy
  - 2 Centers in Switzerland



DDS in CoFlight -- FDP Core

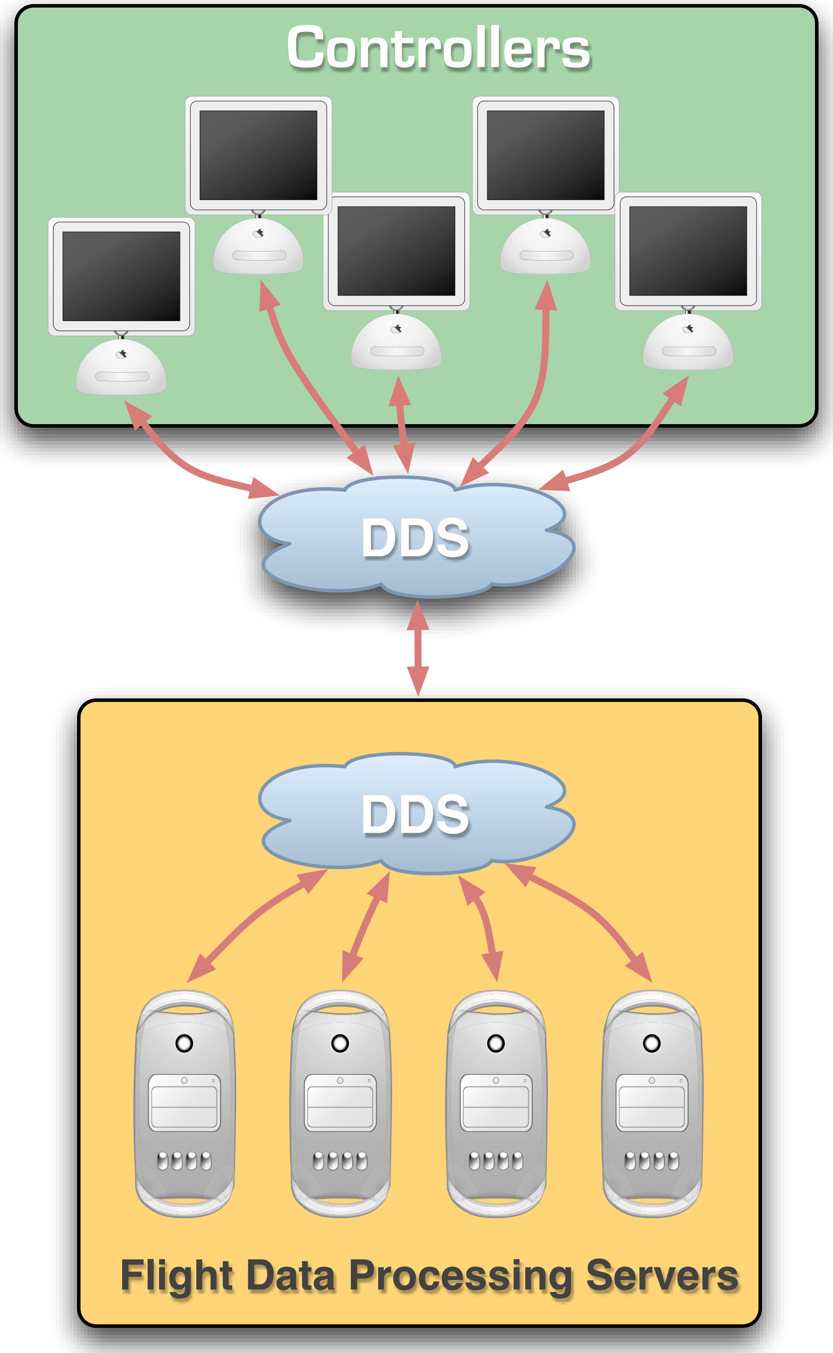
- OpenSplice DDS glues together the most critical components of the CoFlight FDP running at a **SWAL-2** (same as **DO-178B Level B**) assurance level

- In this context OpenSplice DDS distributes flights data plans of redundant LANs



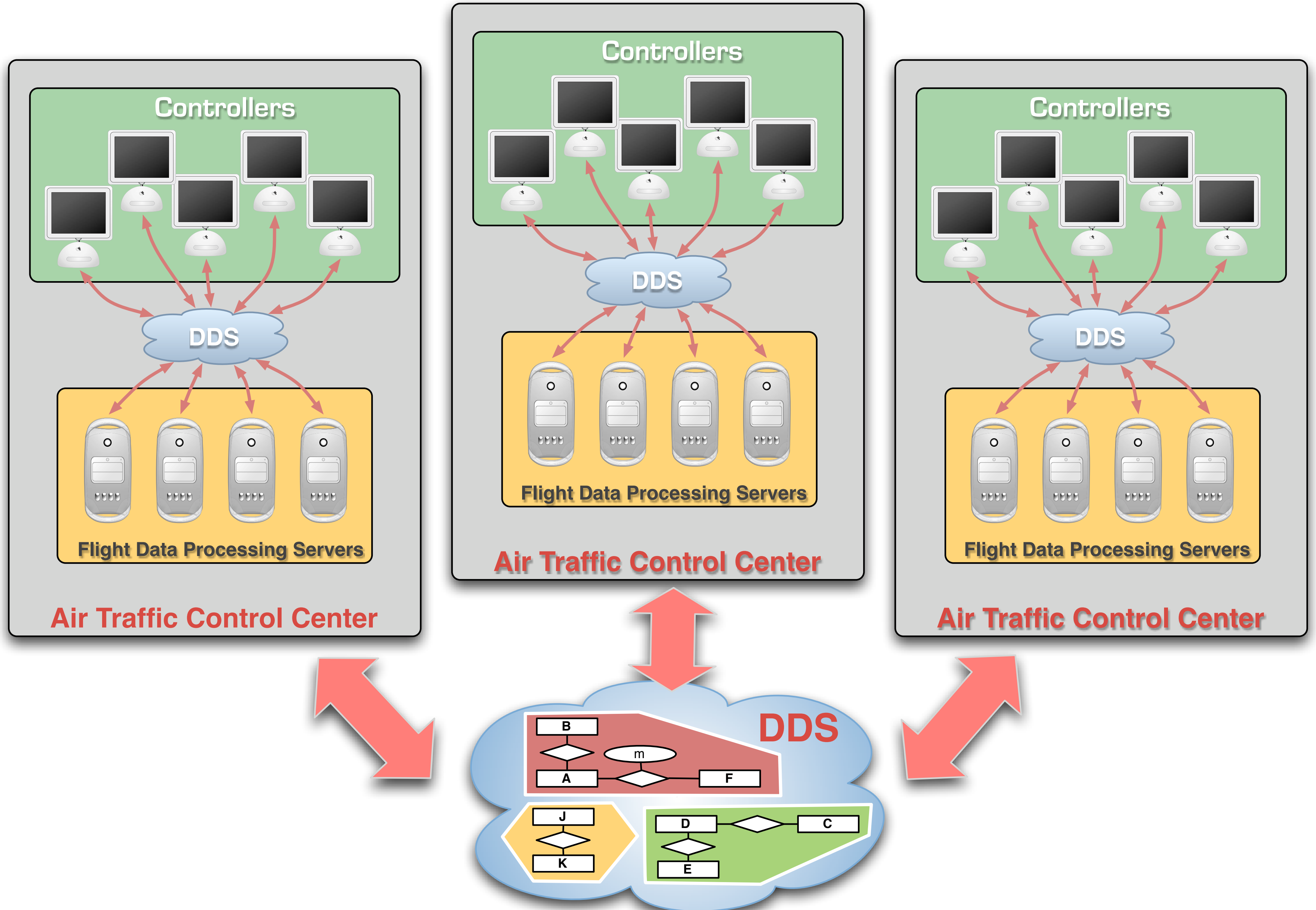
DDS in CoFlight -- CWP

- OpenSplice DDS is used within CoFlight to distribute the "external" Flight Data Plan to Controller Working Positions, to Control Towers, etc.



DDS in CoFlight -- IOP

- OpenSplice DDS is used to integrate CoFlight-based Centers
- OpenSplice DDS is used to provide interoperability with other Interoperable Centers (as per ICOG-2)


Controllers
DDS
Flight Data Processing Servers
Air Traffic Control Center
Controllers
DDS
Flight Data Processing Servers
Air Traffic Control Center
Controllers
DDS
Flight Data Processing Servers
Air Traffic Control Center
DDS
B
m
A
F
J
K
D
C
E

More Use Cases...

Defense & Aerospace

- Naval Combat Management Systems
- Submarines
- Vetronics
- Tactical Links
- Simulation
- Cybercrime
- Flycatcher Systems
- Data Fusion
- Battle Transformation Center

More Use Cases...

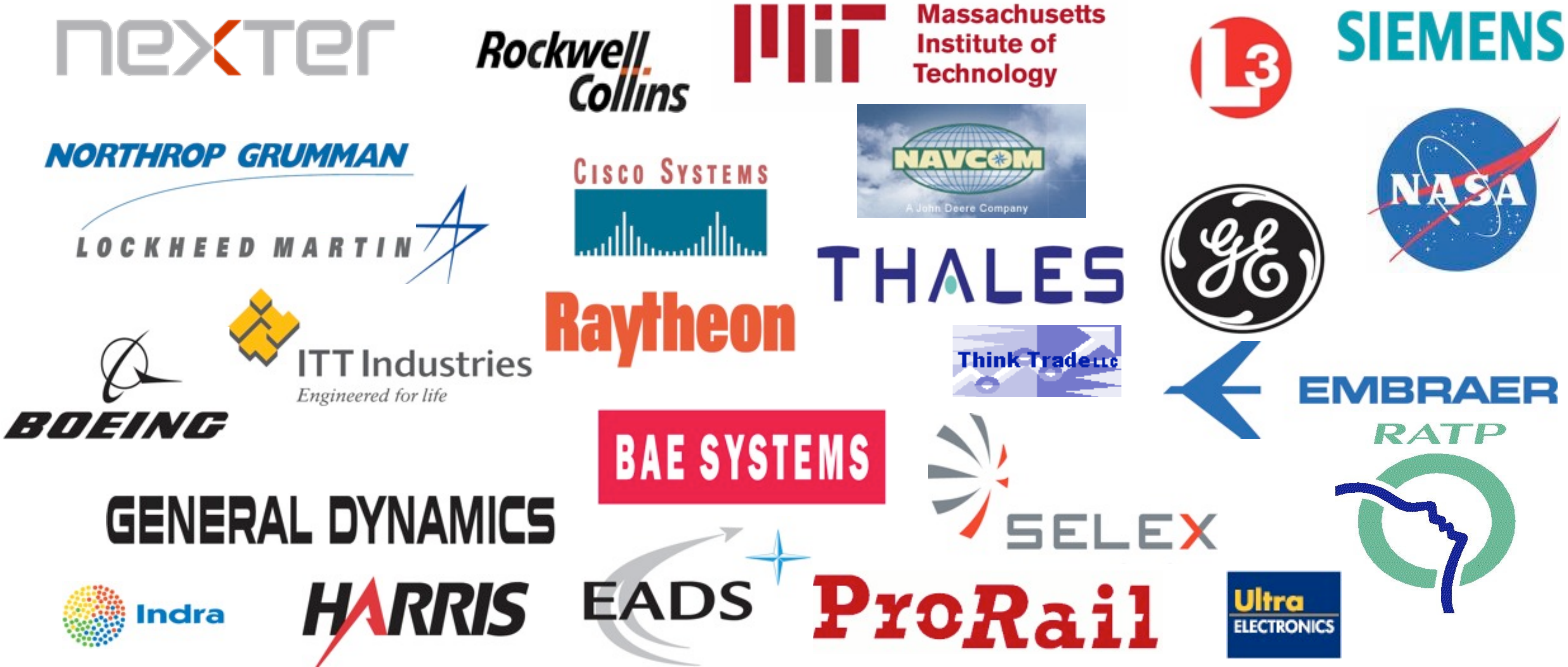
Transportation

- Drones
- Air Traffic Control & Management
- Metropolitan Transportation

Financial Services

- Automated Trading Firms
- Risk Management Firms

Some OpenSplice DDS Users



OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

SUMMiNG-UP

+/-*

In Summary

- The OMG DDS is the most advanced standard-based technology for real-time data distribution
- DDS provides a powerful set of abstractions to build distributed high-availability and high performance systems
- Implementation of the standard, such as OpenSplice DDS are deployed in several extremely challenging mission/business critical systems

REFERENCES

OpenSplice | DDS

Delivering Performance, Openness, and Freedom

- http://www.opensplice.com/
- http://www.opensplice.org/
- emailto:opensplicedds@prismtech.com

webex™

- http://bit.ly/1Sreg

YouTube

- http://www.youtube.com/OpenSpliceTube

slideshare

Present Yourself

- http://www.slideshare.net/angelo.corsaro

twitter

- http://twitter.com/acorsaro/

Blogger

- http://opensplice.blogspot.com

DownLoad OpenSplice DDS at
http://www.opensplice.org

OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

THANK YOU!

P.S. Enjoyed the talk? Found it
Bloody boring? Send me your
comments at:
angelo.corsaro@prismtech.com

OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

Angelo Corsaro, Ph.D.
Chief Technology Officer
PrismTech
OMG DDS SIG Co-Chair
angelo.corsaro@prismtech.com

OpenSplice|DDS

The Universal Data Bus

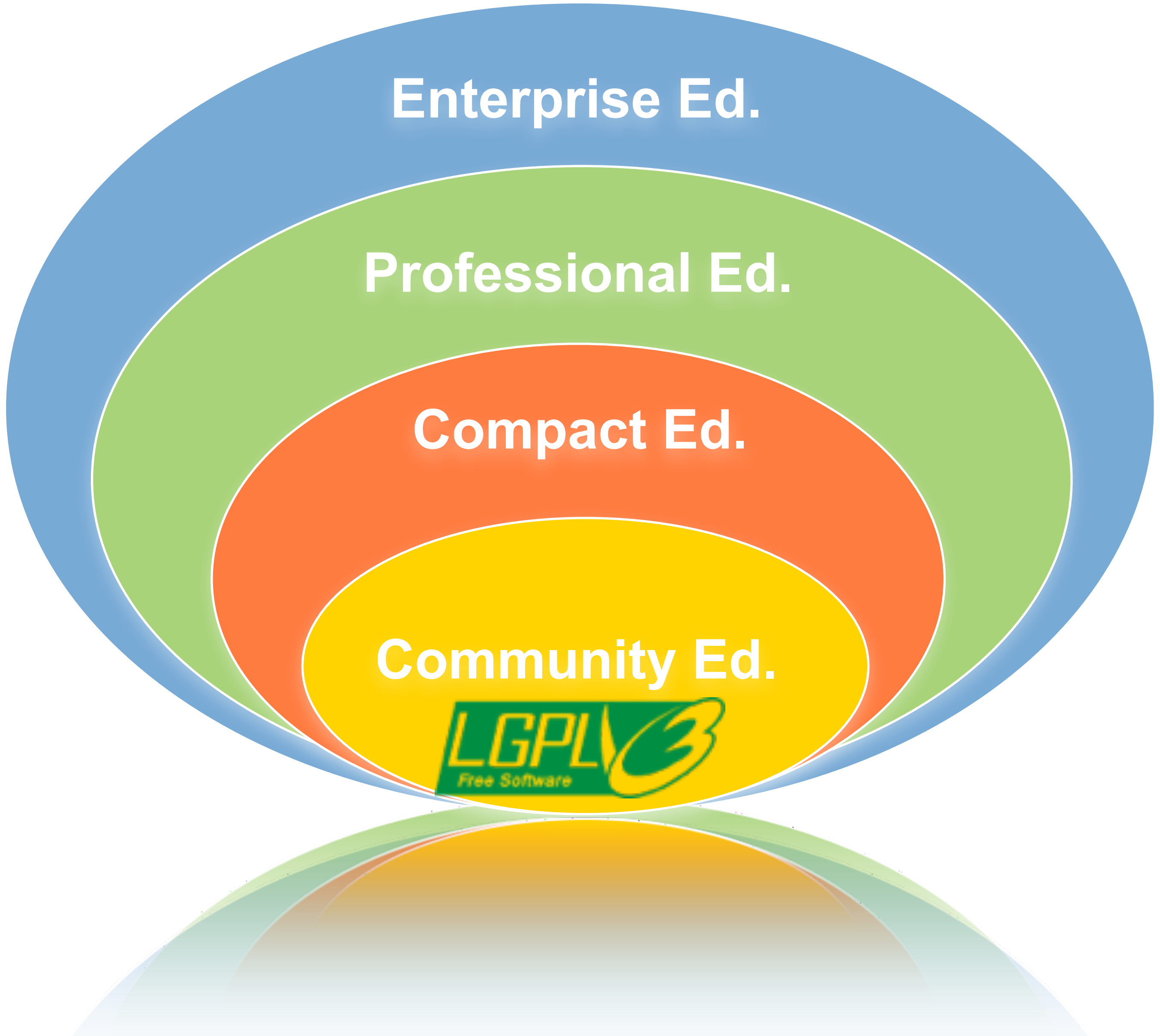
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

PRODUCT STRUCTURE

OpenSplice DDS v5.1

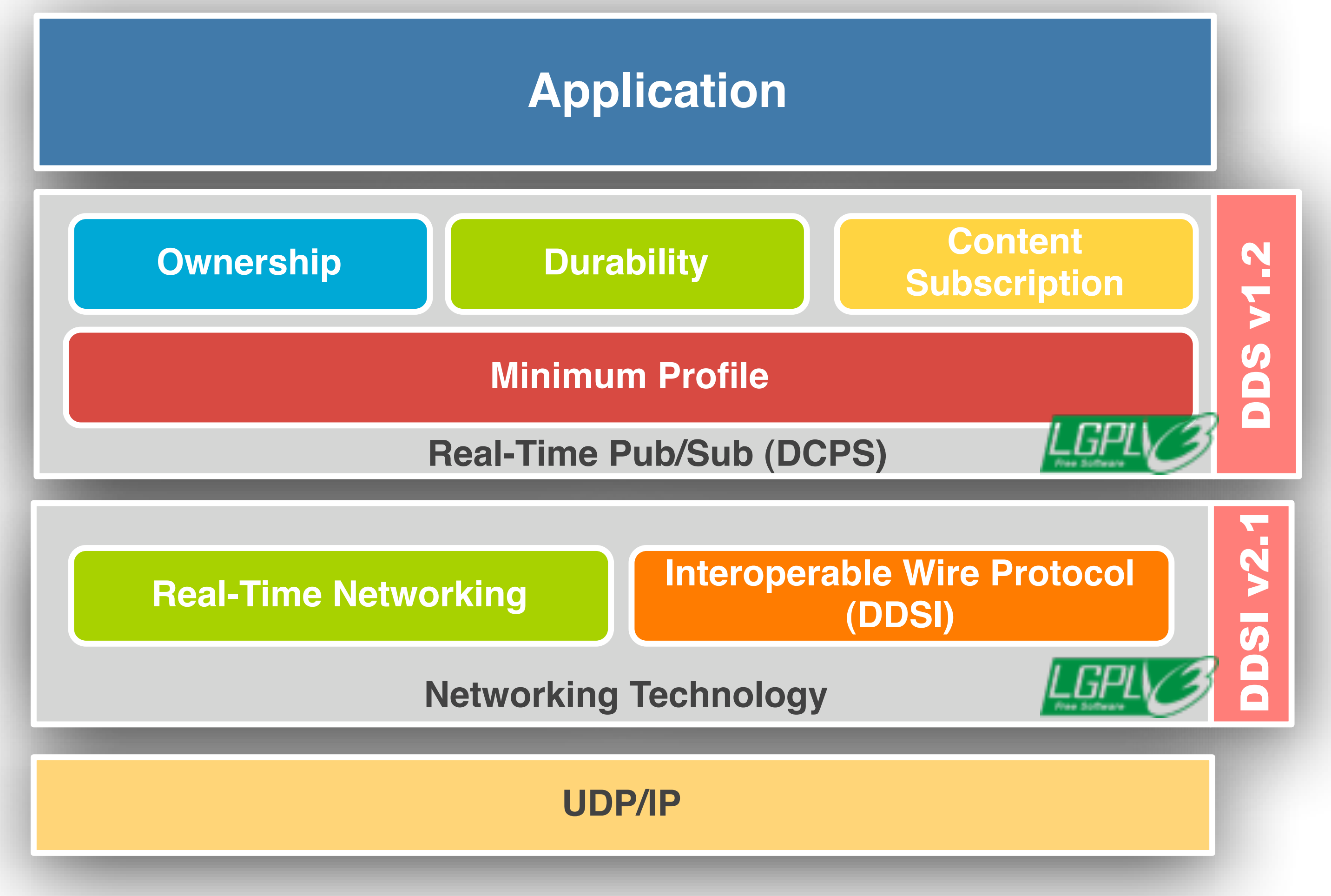
- Four different editions providing a growing set of functionalities:
  - Community Edition
  - Compact Edition
  - Professional Edition
  - Enterprise Edition

- The Community Edition is Open Source. Compact, Enterprise and Professional Edition are available only through Commercial Subscriptions



Community Edition

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control



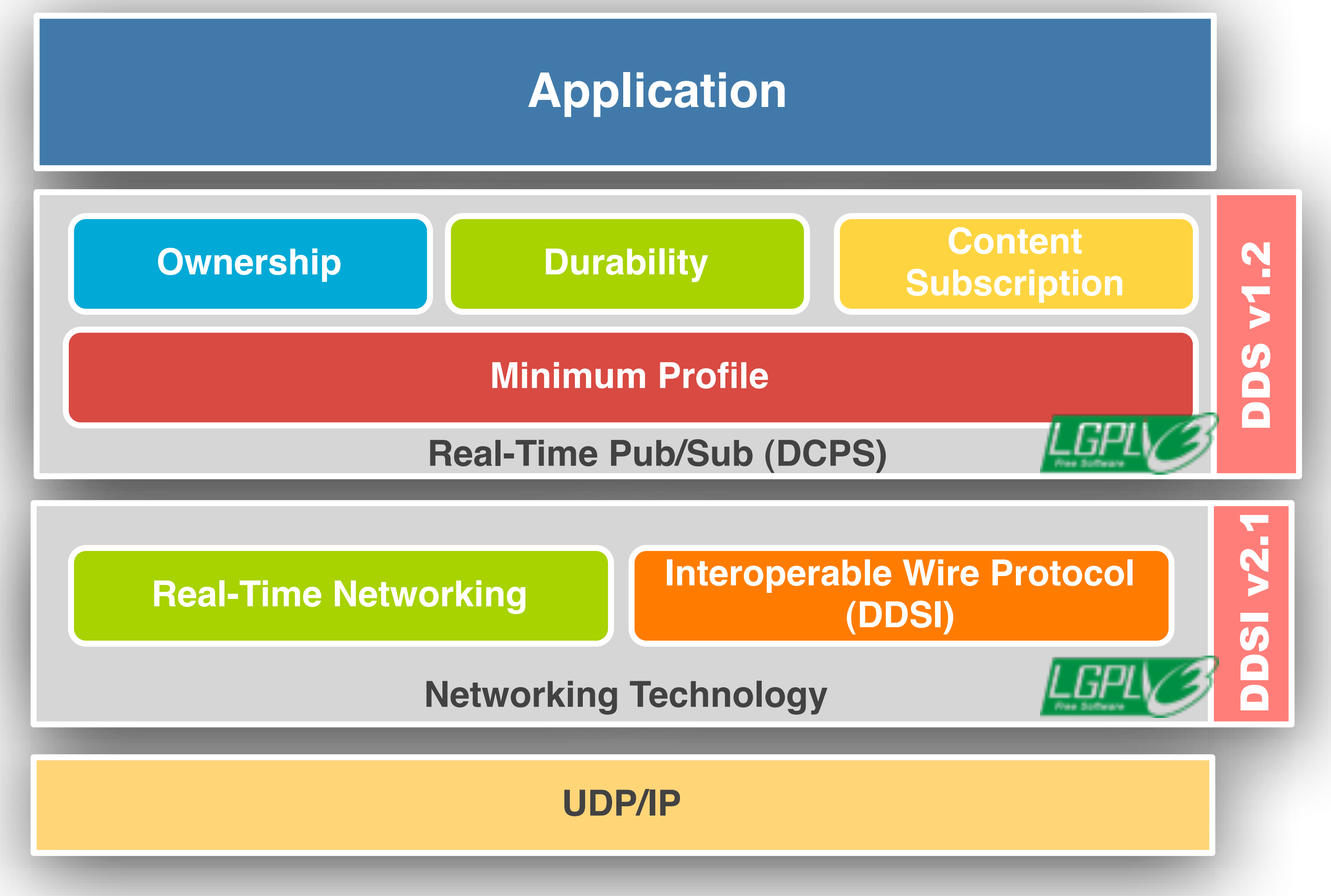
Licensing

- Open Source (LGPLv3)

Community Edition

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control

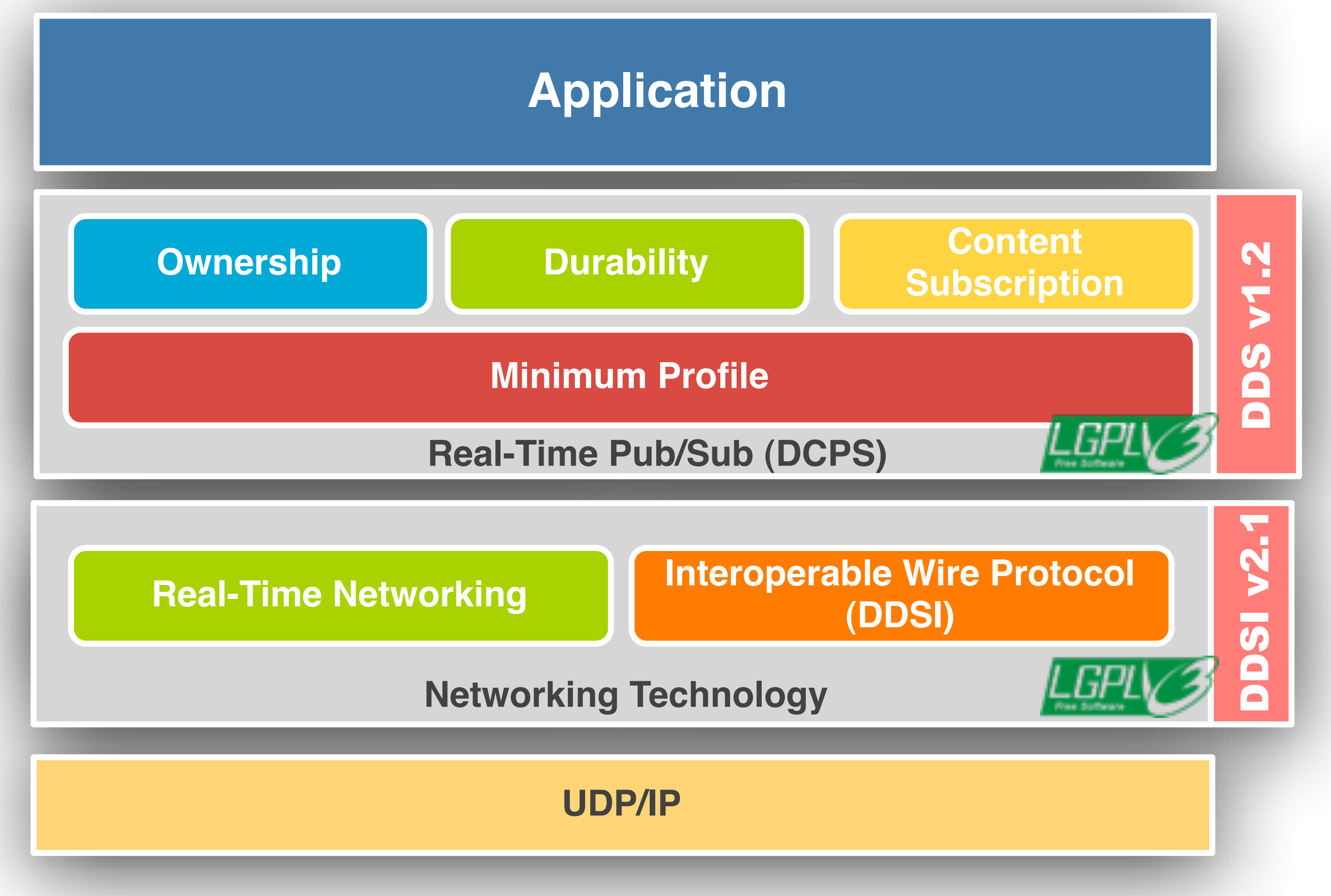


Licensing

- Open Source (LGPLv3)

Community Edition

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control

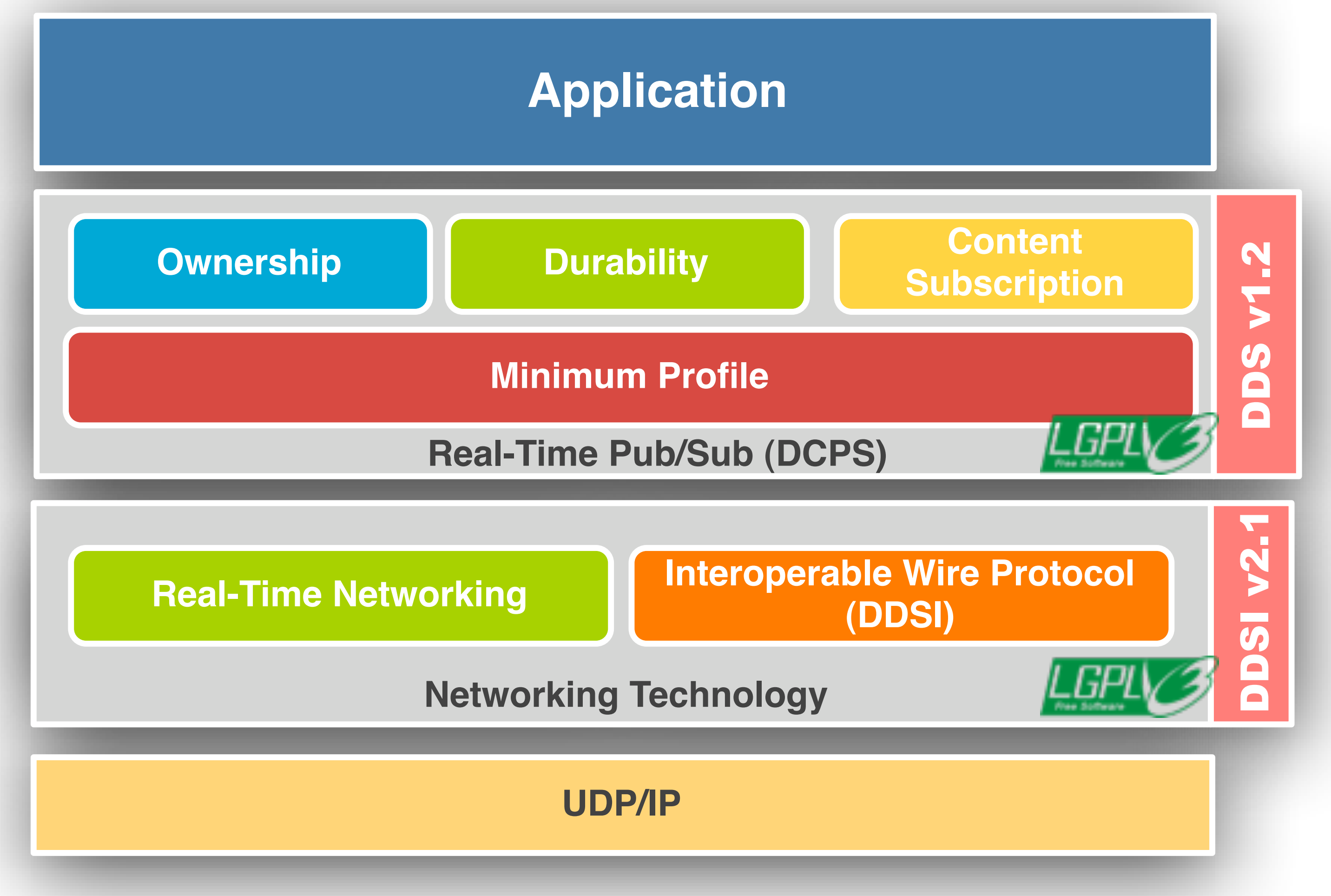


Licensing

- Open Source (LGPLv3)

Community Edition

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control

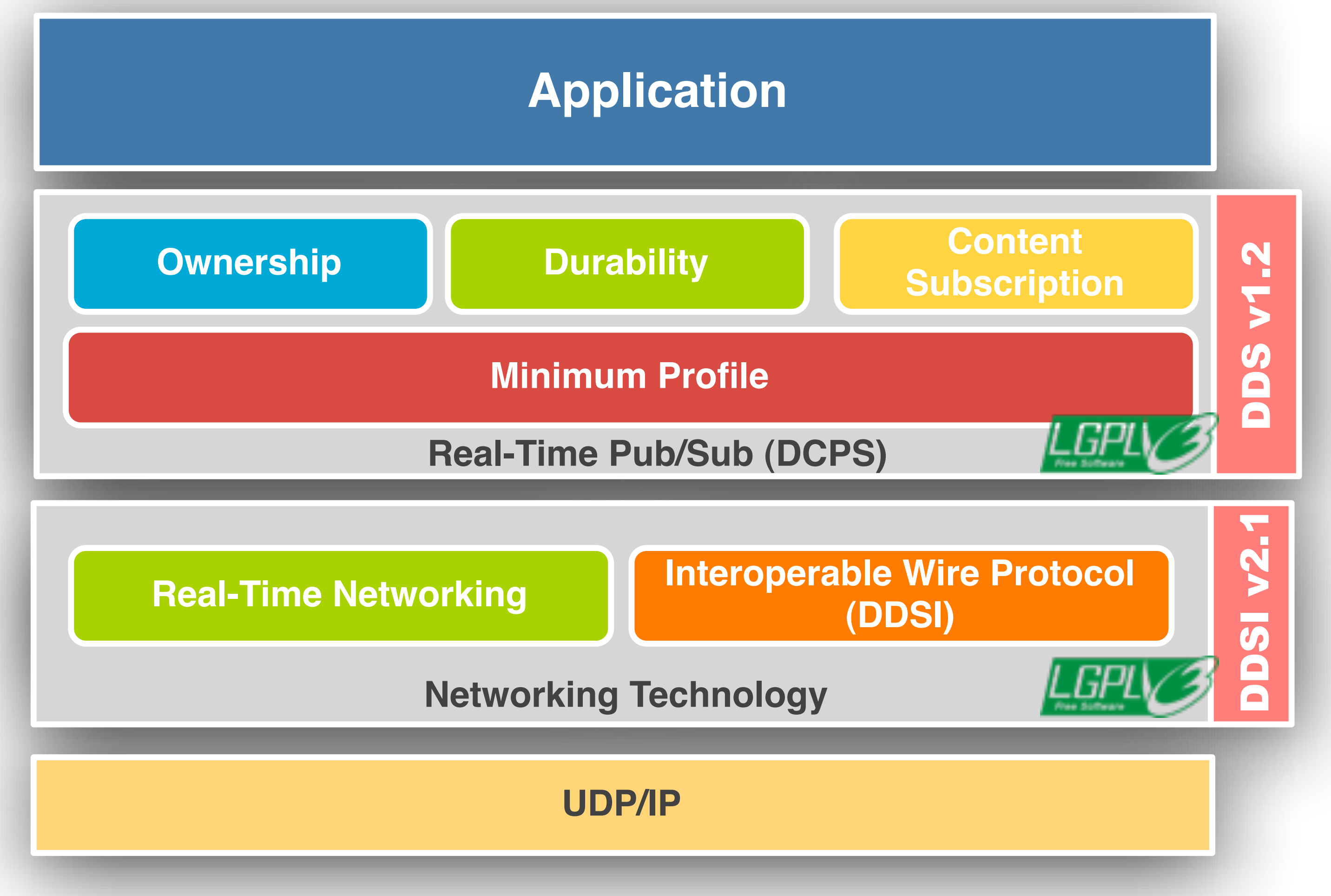


Licensing

- Open Source (LGPLv3)

Community Edition

- Full OMG DDS v1.2 DCPS
- Real-Time Networking
- DDSI v2.1 beta
- WAN Discovery
- Compression
- Quorum Writers
- Read/Write Access Control



Licensing

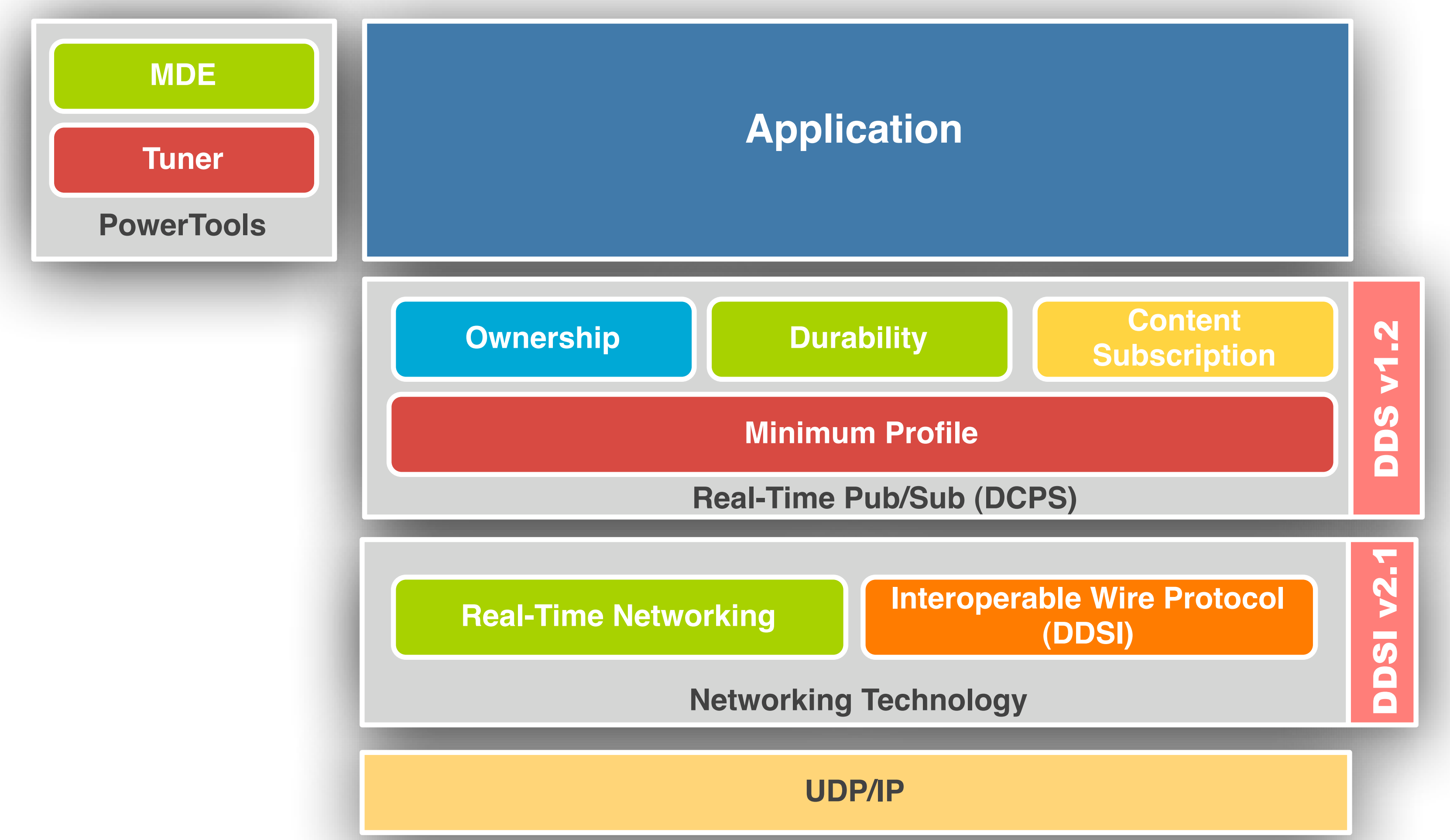
- Open Source (LGPLv3)

Compact Edition

- All Community ed Features
- Power Tools
  - Eclipse Visual Modeling Tool
  - Tuner Tool

Licensing

- Commercial Subscription / PrismTech Source Code License (non-copy-left)



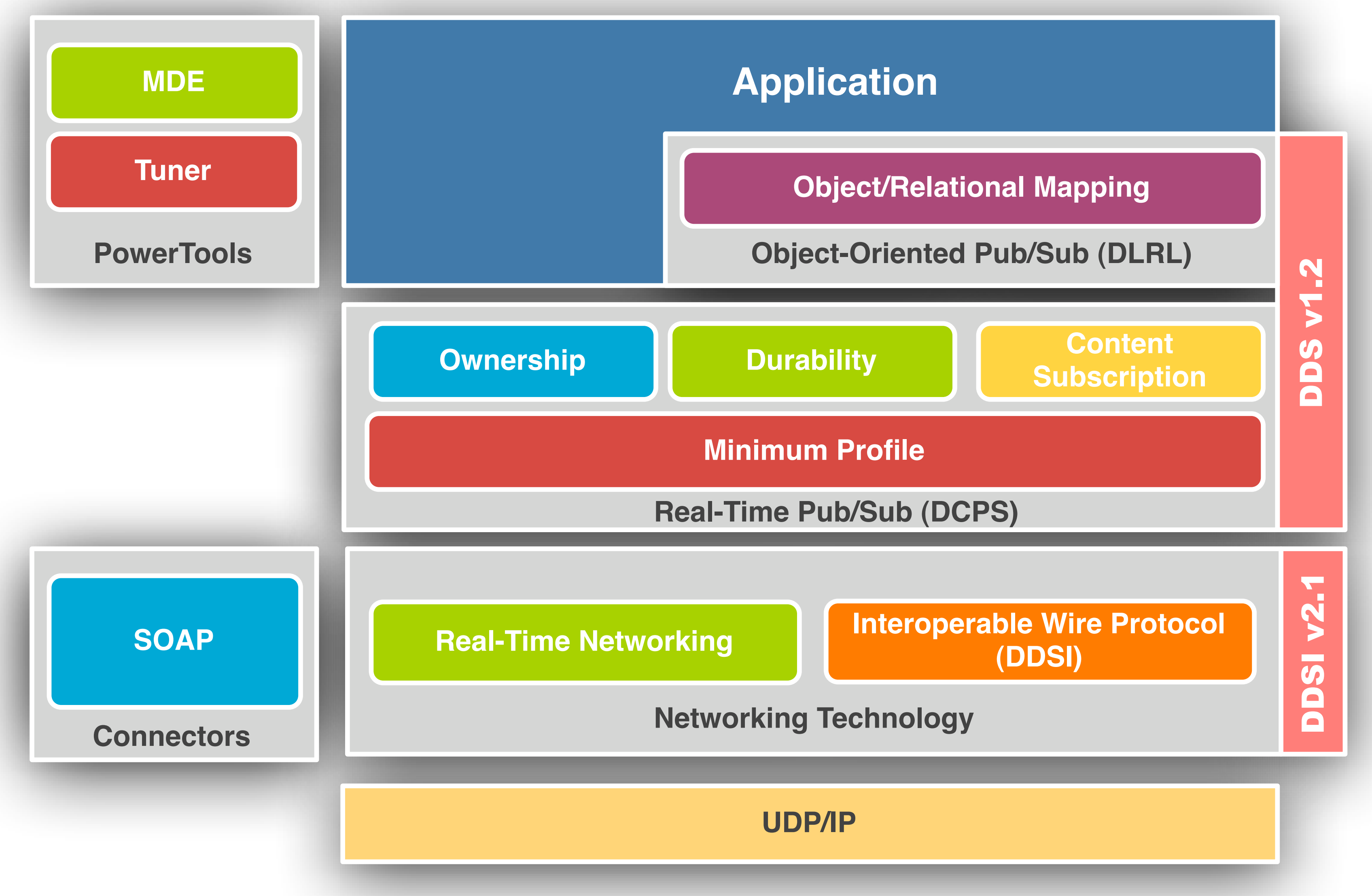
Professional Edition

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

- All Compact Ed. Features
- DLRL v1.2
- **Power Tools**
  - Eclipse Visual Modeling Tool
  - Tuner Tool
- Connectors
  - SOAP

Licensing

- Commercial Subscription / PrismTech Source Code License (non-copy-left)



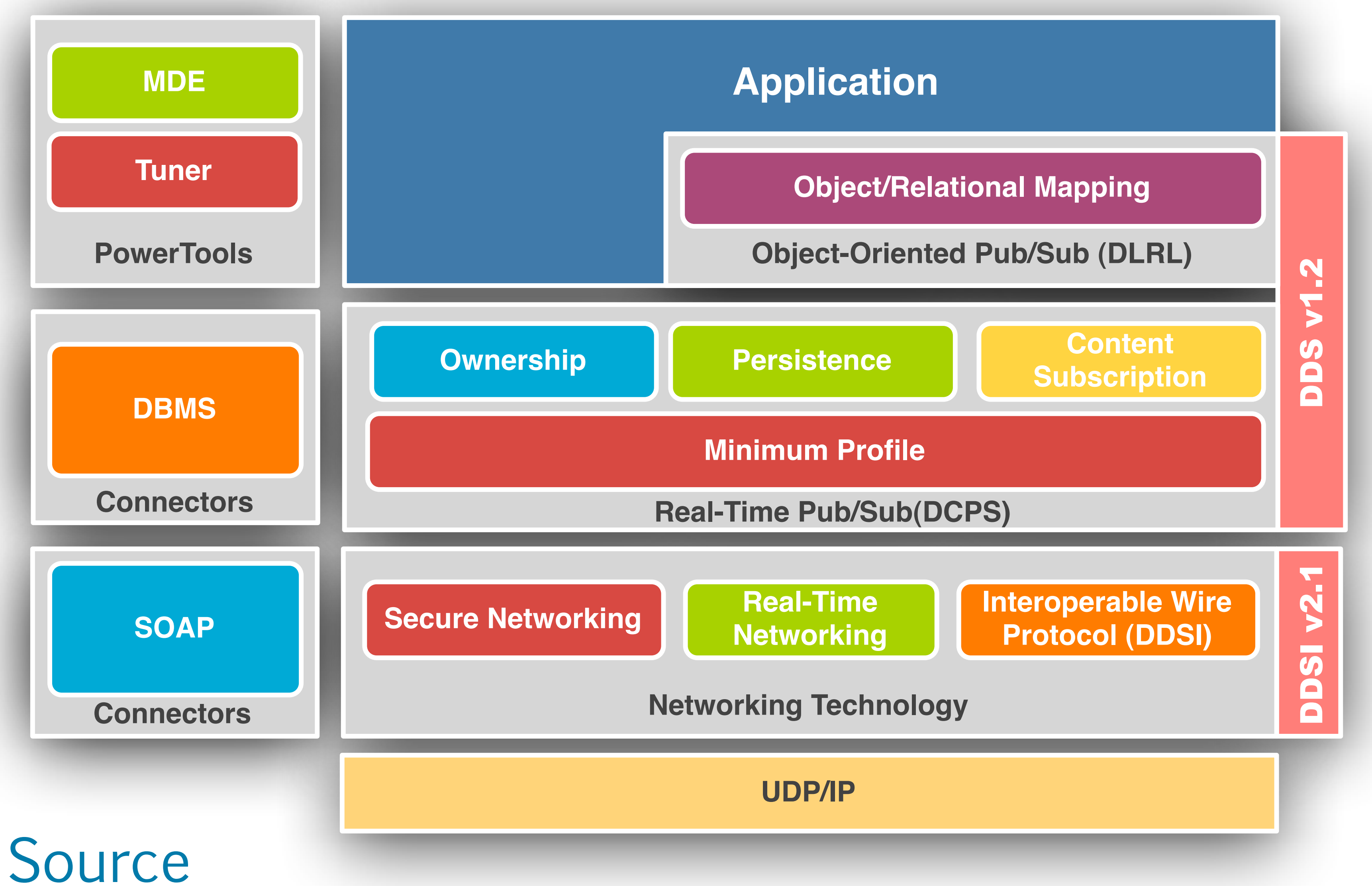
Enterprise Edition

Features

- All Professional Ed. Features
- Security
- Connectors
  - DBMS

Licensing

- Commercial Subscription / PrismTech Source Code License (non-copy-left)

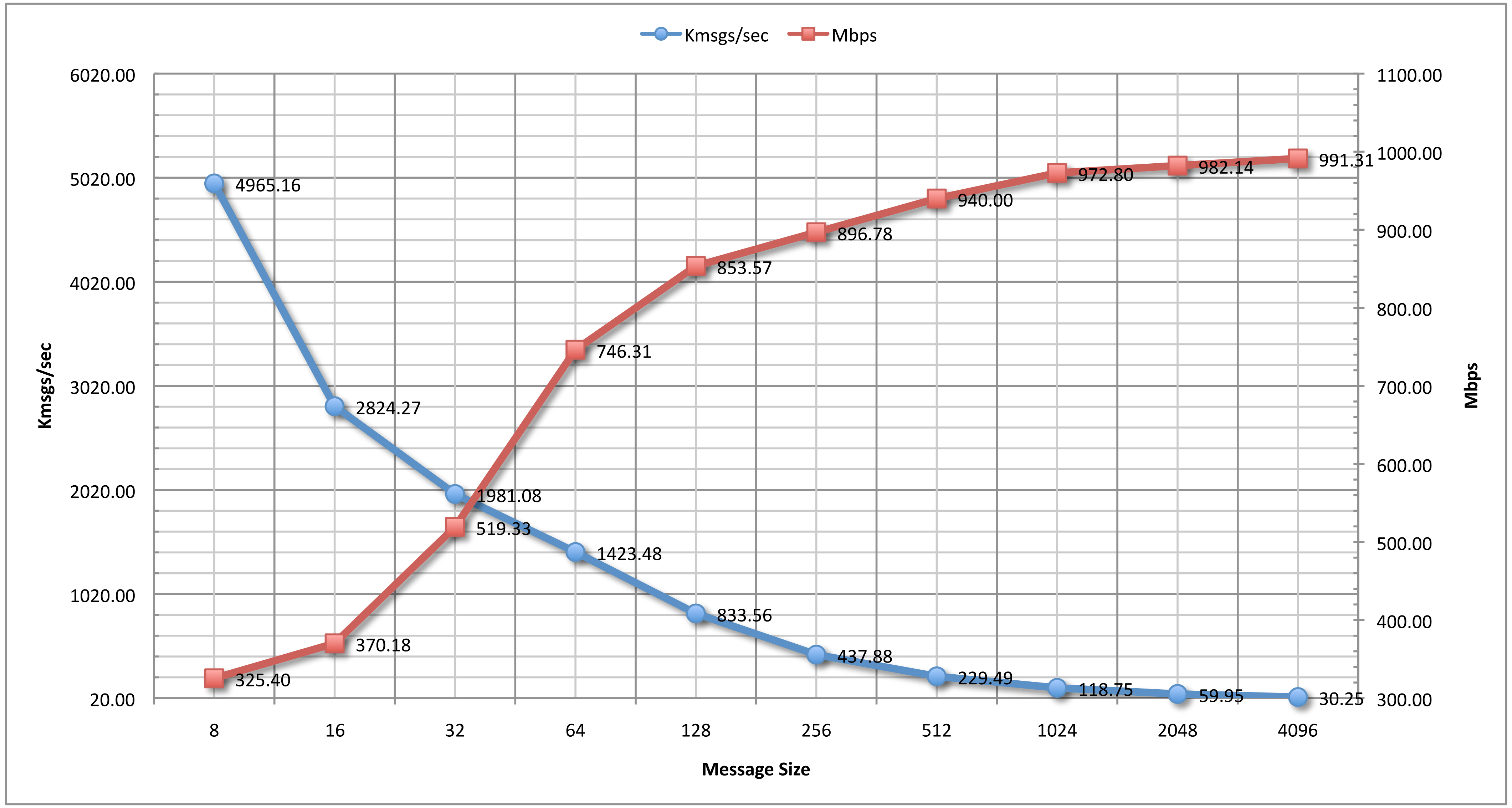


Standing on Giant Shoulders

	OMG DDS Standard Compliance					
	DCPS Profiles				DLRL	DDSI/ RTPS
	Minimum	*Content*	*Ownership*	*Durability*		
OpenSplice DDS Community Ed.	Yes	Yes	Yes	Yes	No	Yes
Other DDS (Best Case)	Yes	Partial	Yes	No*	No	Yes

(*) DURABILITY not built-in the DDS Implementation but available as stand-alone service

Throughput



Inter-Node Latency

- 60 usec

Inter-Core Read-Latency

- 2 usec

Inter-Core Latency

- <10 usec

Test Scenario

- Single Threaded Application
- 8192 bit message batches

HW:

- Dell blade-server
- Dual-core, Dual-CPU, AMD Opteron 2.4 Ghz

OS

- Linux 2.6.21-1.3194.fc7

Network

- Gigabit Ethernet cards
- Dell PowerConnect 5324 switch

OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

OPEN SOURCE ECOSYSTEM

SimD: Simple DDS

- Incubator project for the upcoming ISO C++ DDS PSM
- Simple, safe, and efficient
- Available at:
  - http://code.google.com/p/simd-cxx/

Twitting with SimD

Writing Tweets

```
dds::Topic<TweetType> topic("TweetTopic");

dds::DataWriter<TweetType> dw(topic);

TweetType tt = {
   "@bird",
   "Writing next-gen tweeter in DDS"
};

dw.write(tt);
```

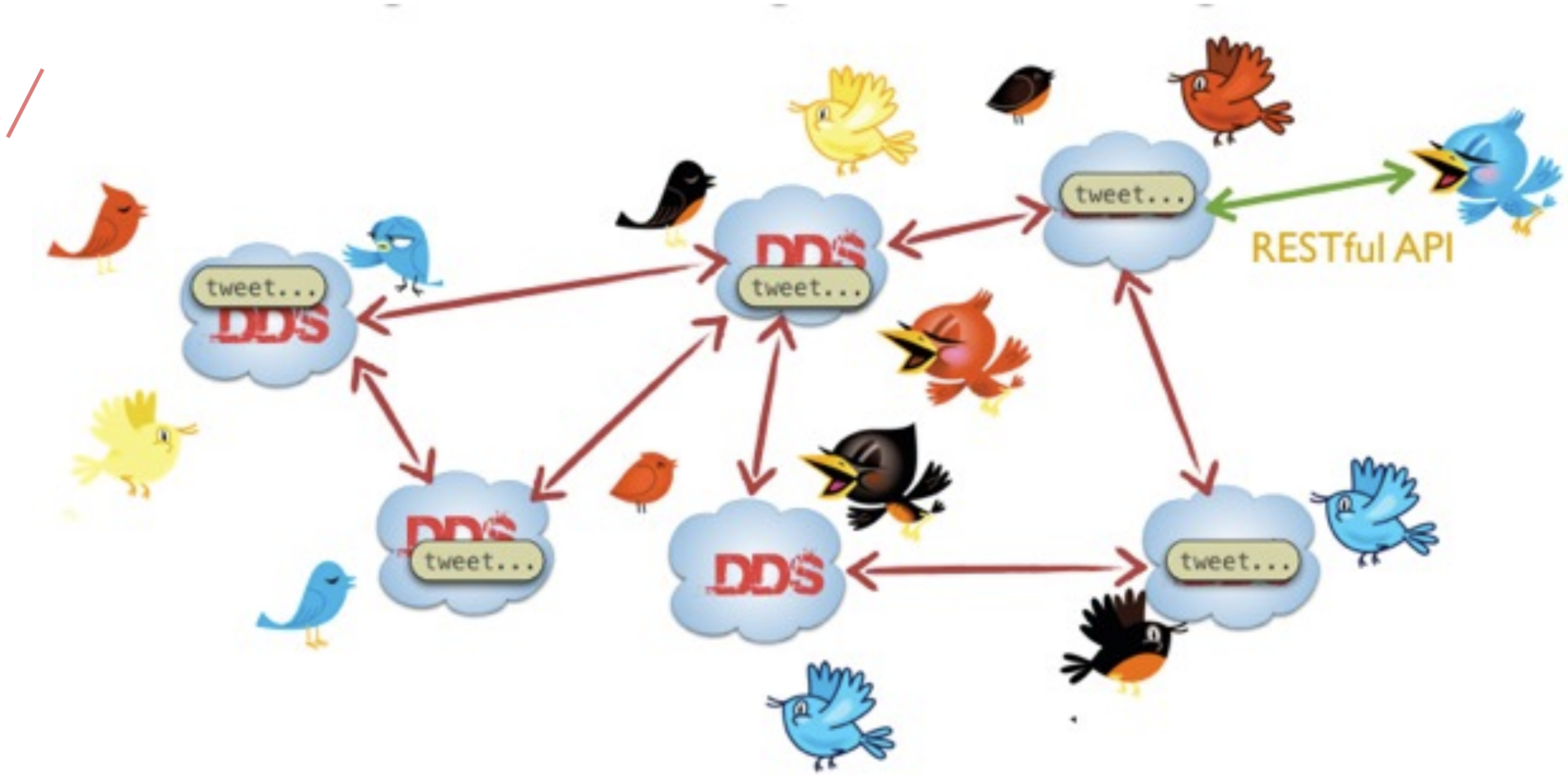
Reading Tweets

```
dds::Topic<TweetType> topic("TweetTopic");

dds::DataReader<TweetType> dr(topic);

std::vector<TweetType> data;
std::Vector<SampleInfo> info;

dr.read(data, info);
```

RESTful Connector

- Provides a RESTful API for performing the basic **CRUD** (C=Create, R=Reads, U=Update, D=Delete) operations on DDS

- Available at:
  - http://code.google.com/p/restful-dds/



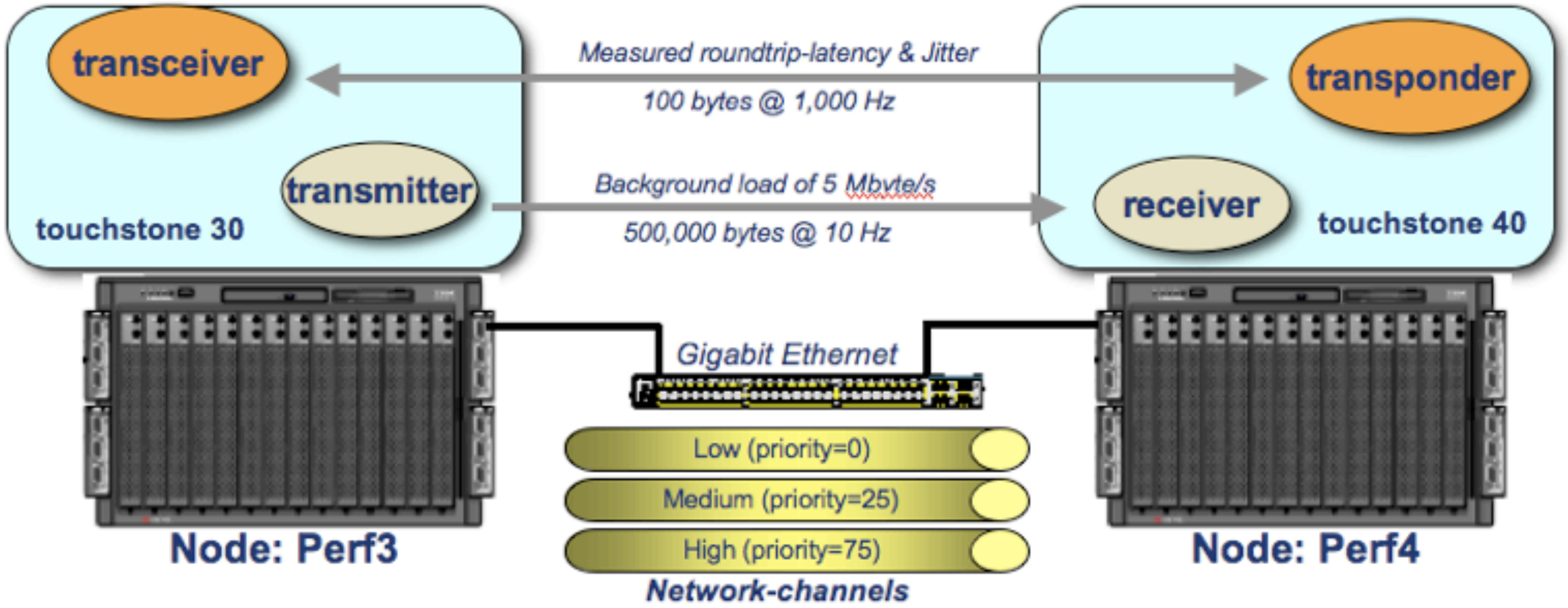
CamelOS

- OpenSplice DDS Apache Connector
- Provides you access to DDS from the 80 connectors currently available in Apache Camel
- Available at:
  - http://fusesource.com/wiki/display/CAMELOPENSPLICE/Home

DDS TouchStone

- Scenario-driven Benchmarking Framework allowing to quickly measure measure latencies and throughputs for user-specified scenarios

- DDS TouchStone provides a time-effective and meaningful way of assessing OpenSplice DDS suitability for a specific application



OpenSplice|DDS

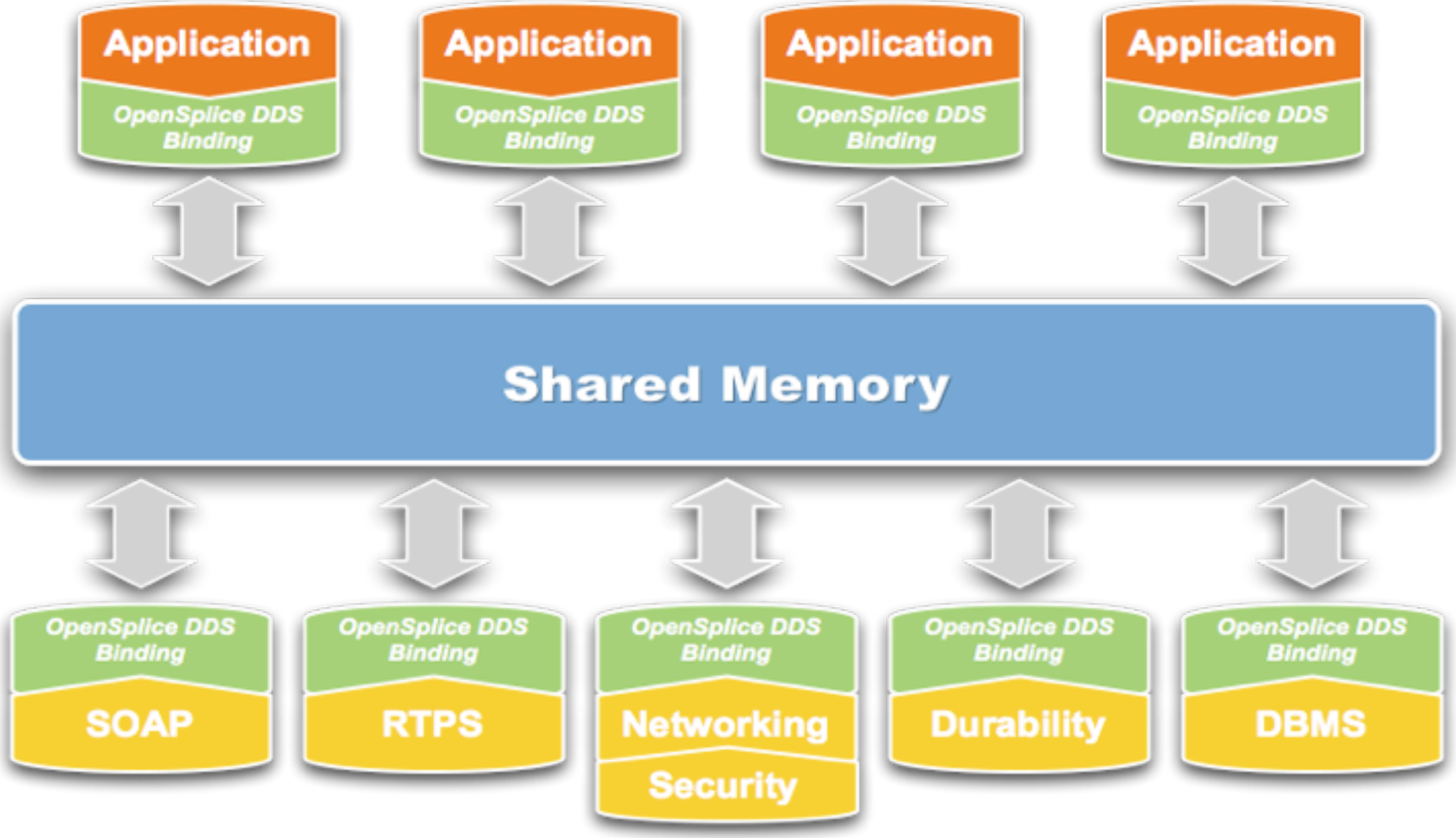
The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

TECHNICAL HIGHLIGHTS

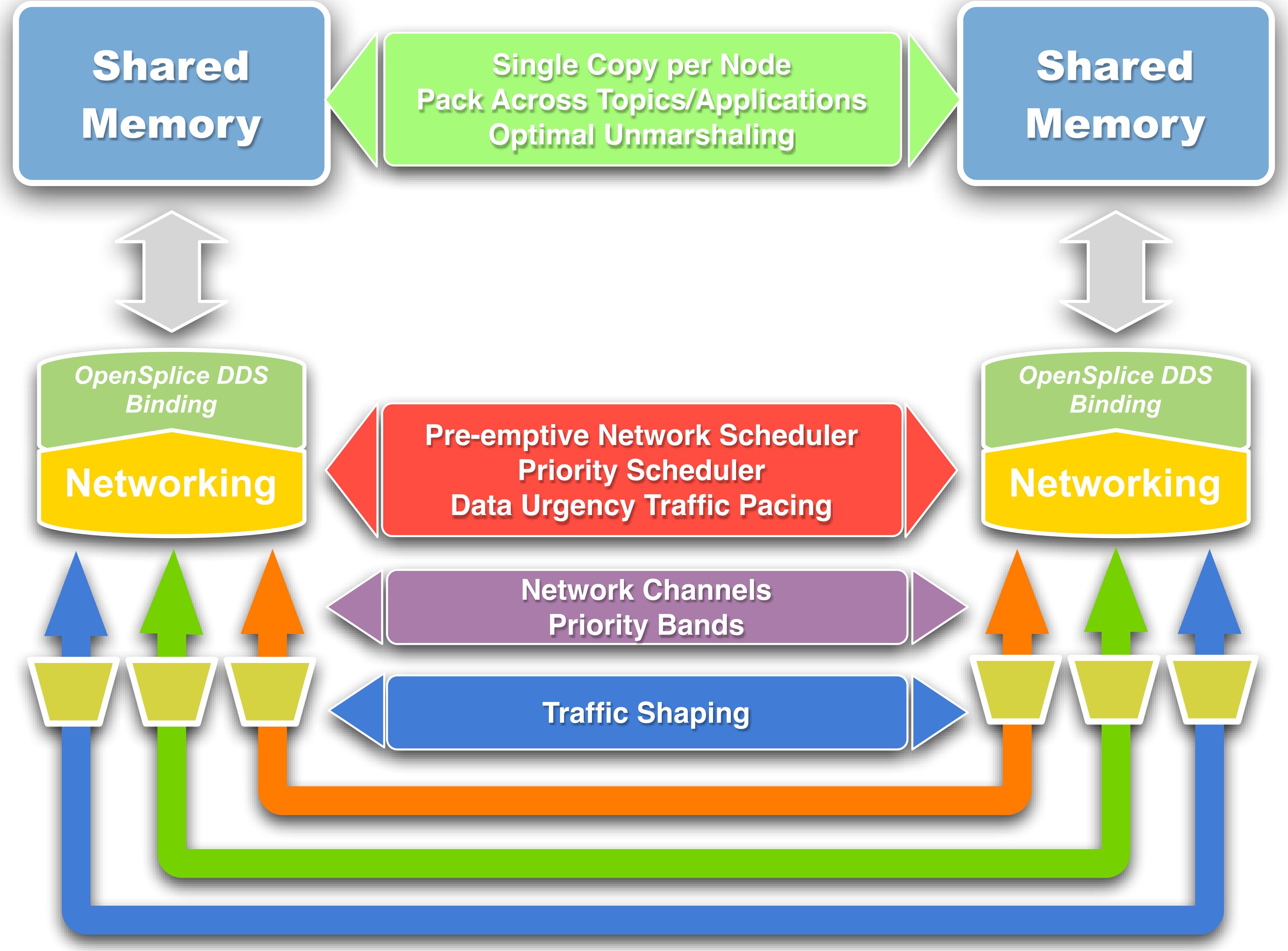
Architectural Outlook

- Shared-Memory based architecture for **minimizing inter-core latency**, and maximizing nodal scalability
- Optional Library-Version
- Plug-in Architecture
- Full control over network scheduling



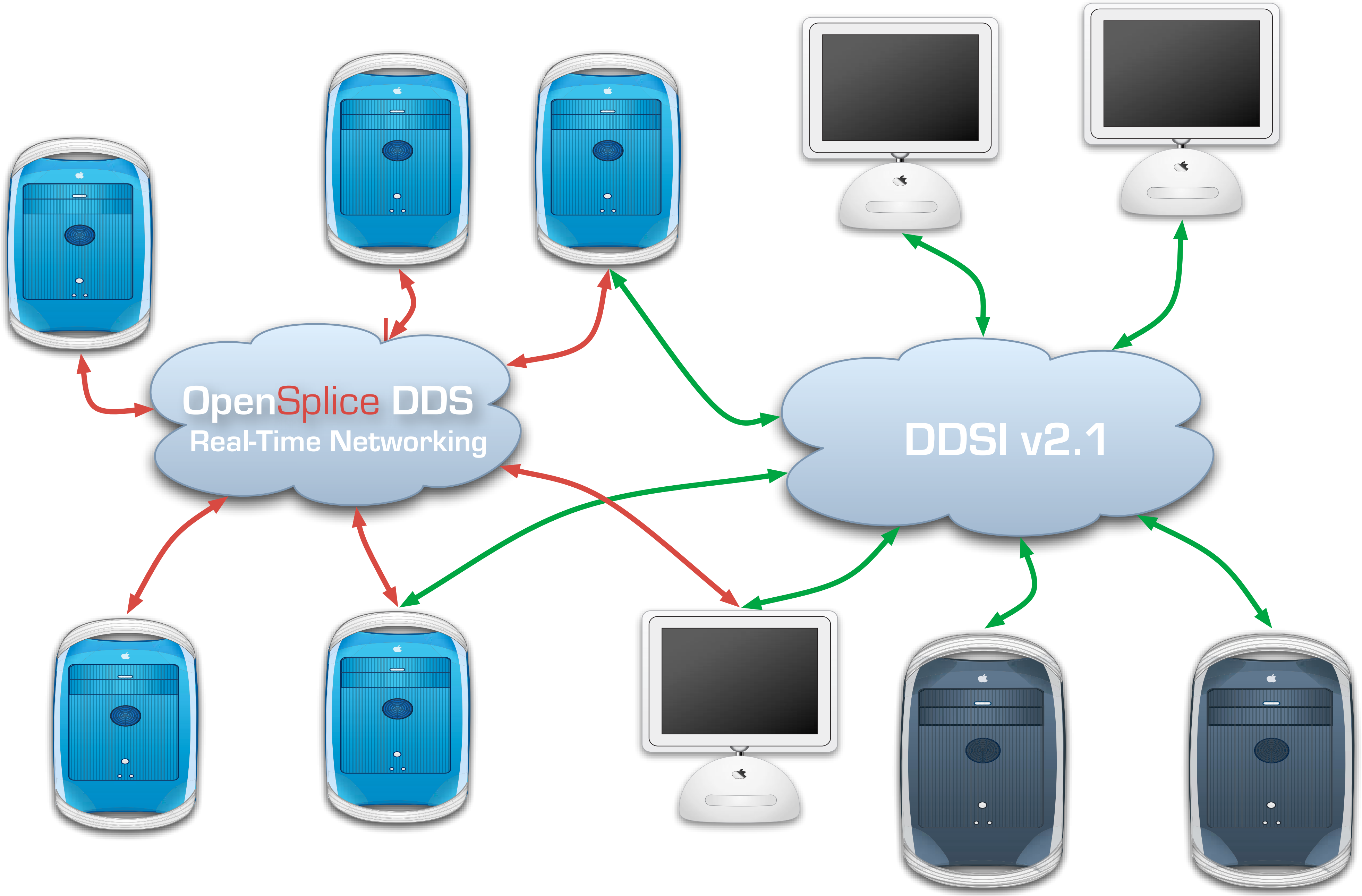
Real-Time Networking

- Real-Time Transport
- End-to-end priority preservation
  - Full Control over Priority Inversion
  - No Head-of-Line Blocking
- Node-wide traffic scheduling and shaping



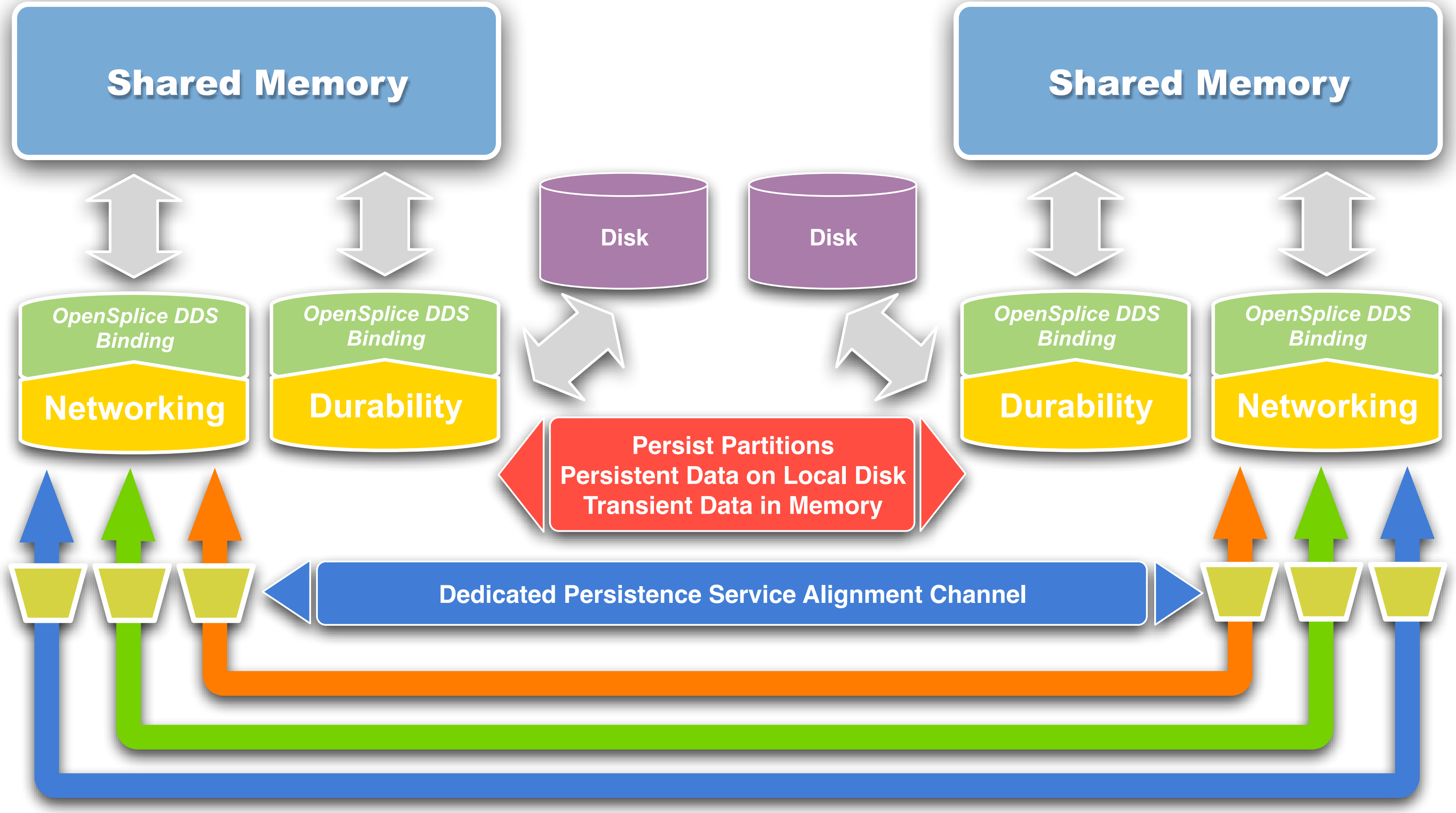
Multi-Protocol

- OpenSplice DDS Plug-in architecture allows you to run concurrent protocols
- This freedom of choice allows you to select the most appropriate protocol w/o compromising interop.



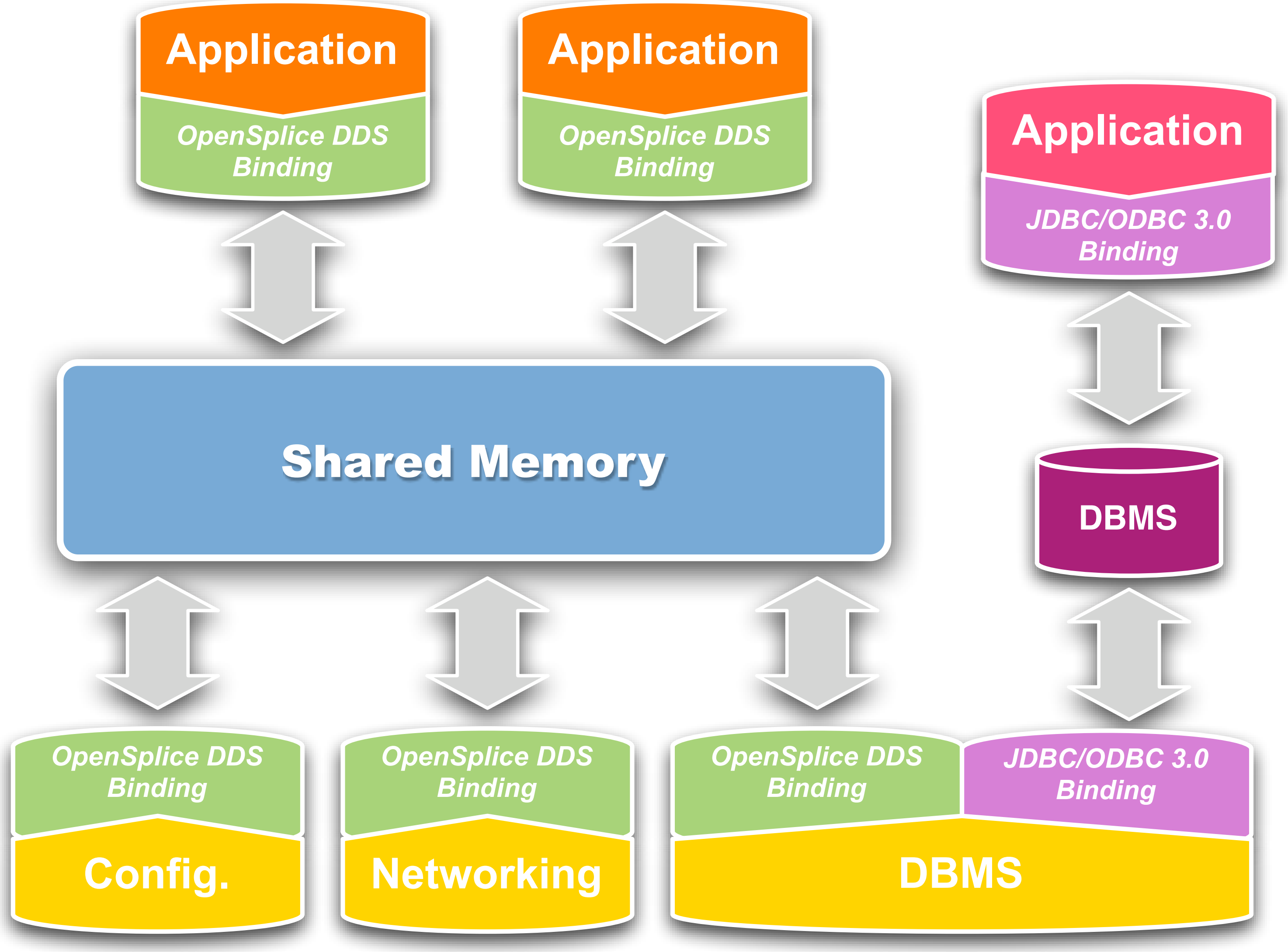
Durable Data Technology

- Fully distributed high-performance data durability
- Data alignment mediated by the networking service to control the impact over real-time data
- Flexible mechanism to control what-to, where-to and how-to to replicate data



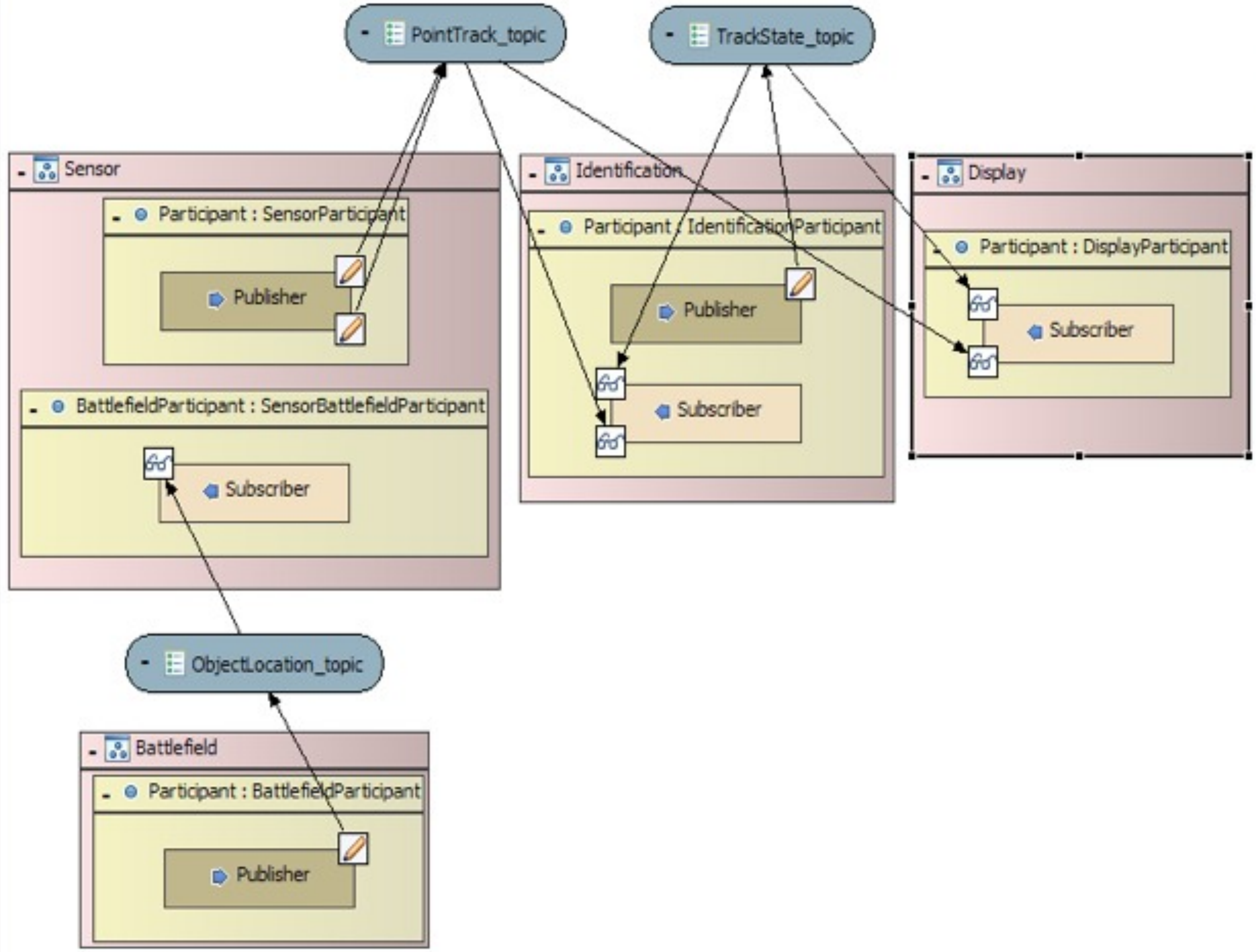
DBMS Connector

- Declarative two way mapping from/to DDS Topics to/from DBMS Tables
- Transparent two-way data-flow DDS<=>DBMS
- Support for any ODBC 3.0, JDBC DBMS
- Support for In-Memory DBMS



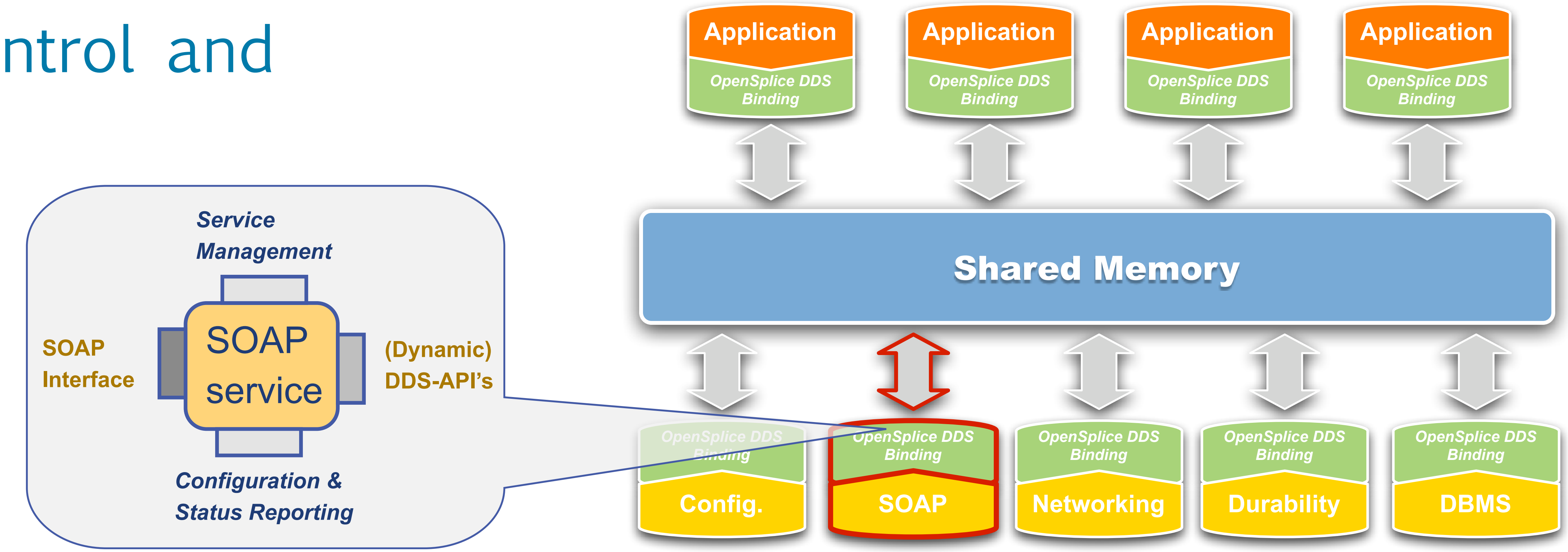
MDE Tools

- Eclipse-based MDE tool
- Geared at application modeling
- Promotes clear separation between technical layer and business logic
- Support for Java/C++


PointTrack_topic
TrackState_topic
Sensor
Participant : SensorParticipant
Publisher
BattlefieldParticipant : SensorBattlefieldParticipant
Subscriber
Identification
Participant : IdentificationParticipant
Publisher
Subscriber
Display
Participant : DisplayParticipant
Subscriber
ObjectLocation_topic
Battlefield
Participant : BattlefieldParticipant
Publisher

SOAP Connector

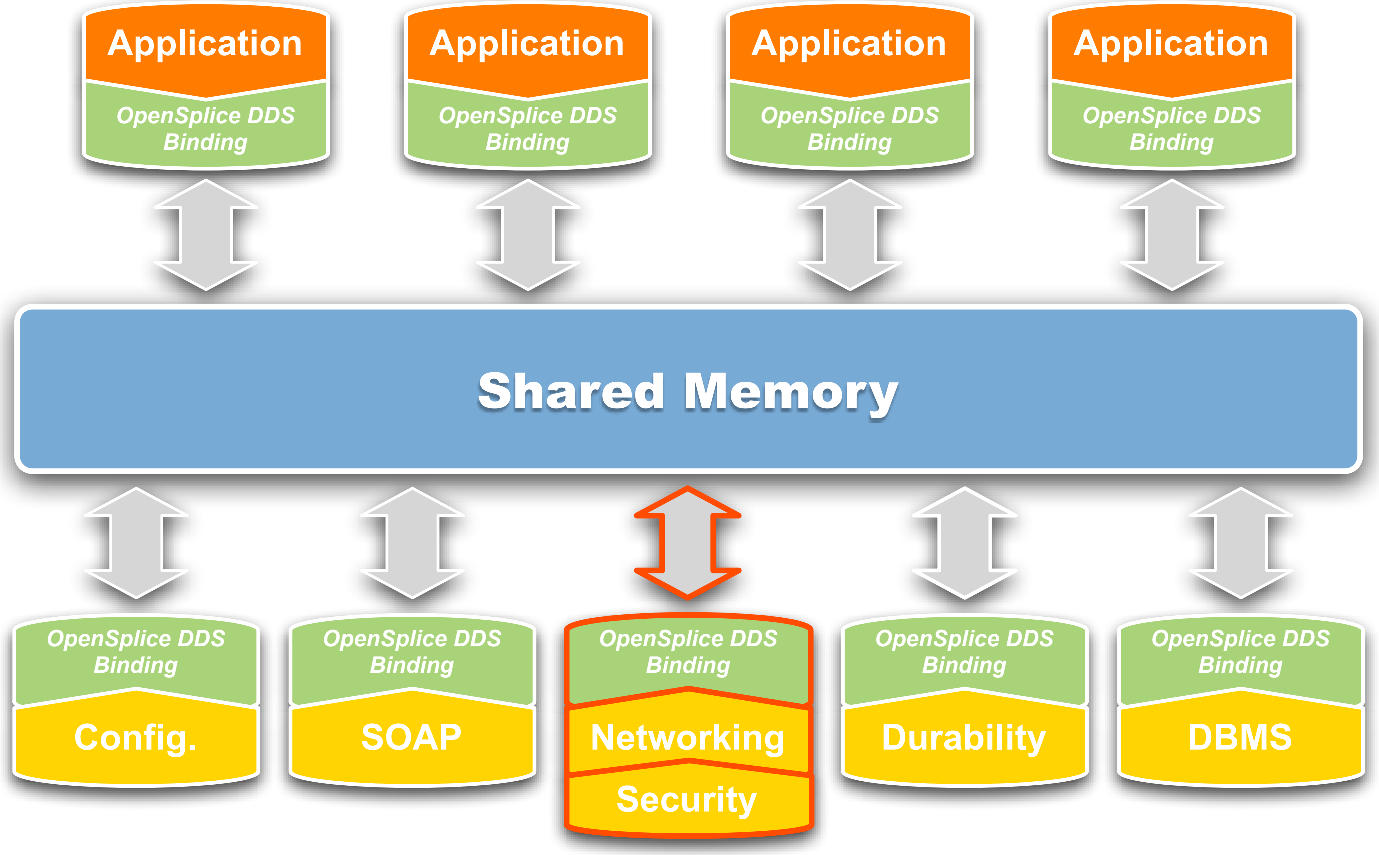
- Full DDS API in XML/SOAP
- Support for Dynamic Topic Types
- XML/SOAP Control and Monitoring API



Secure Networking

Secures DDS communication ensuring:

- **Confidentiality**: Data can be shared only among authorized peers.
- **Integrity**: Data authenticity is asserted via authentication protocols
- **Availability**: Data is available as long as one node is running



OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

LOOKiNG AHEAD

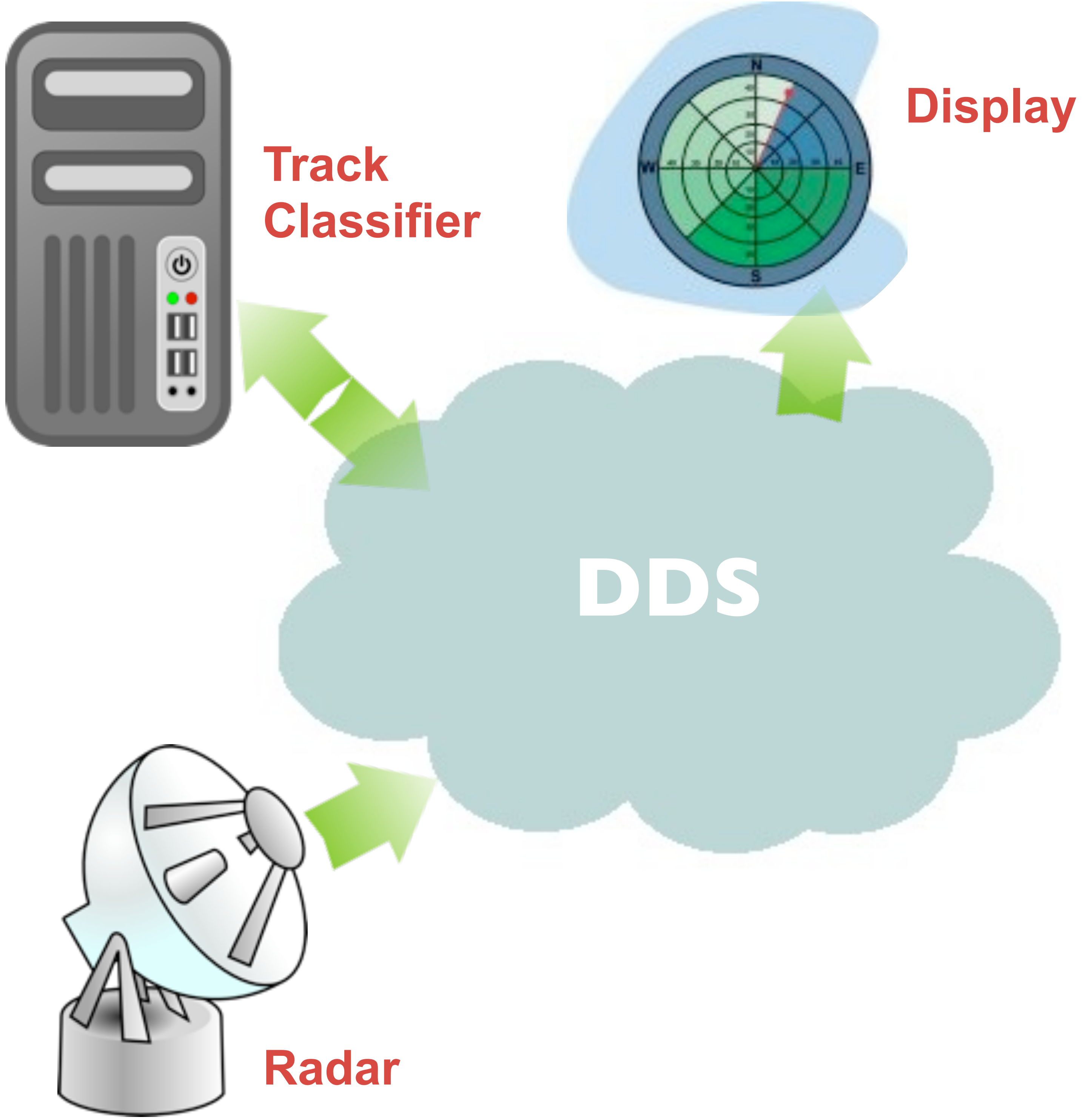
OpenSplice|DDS

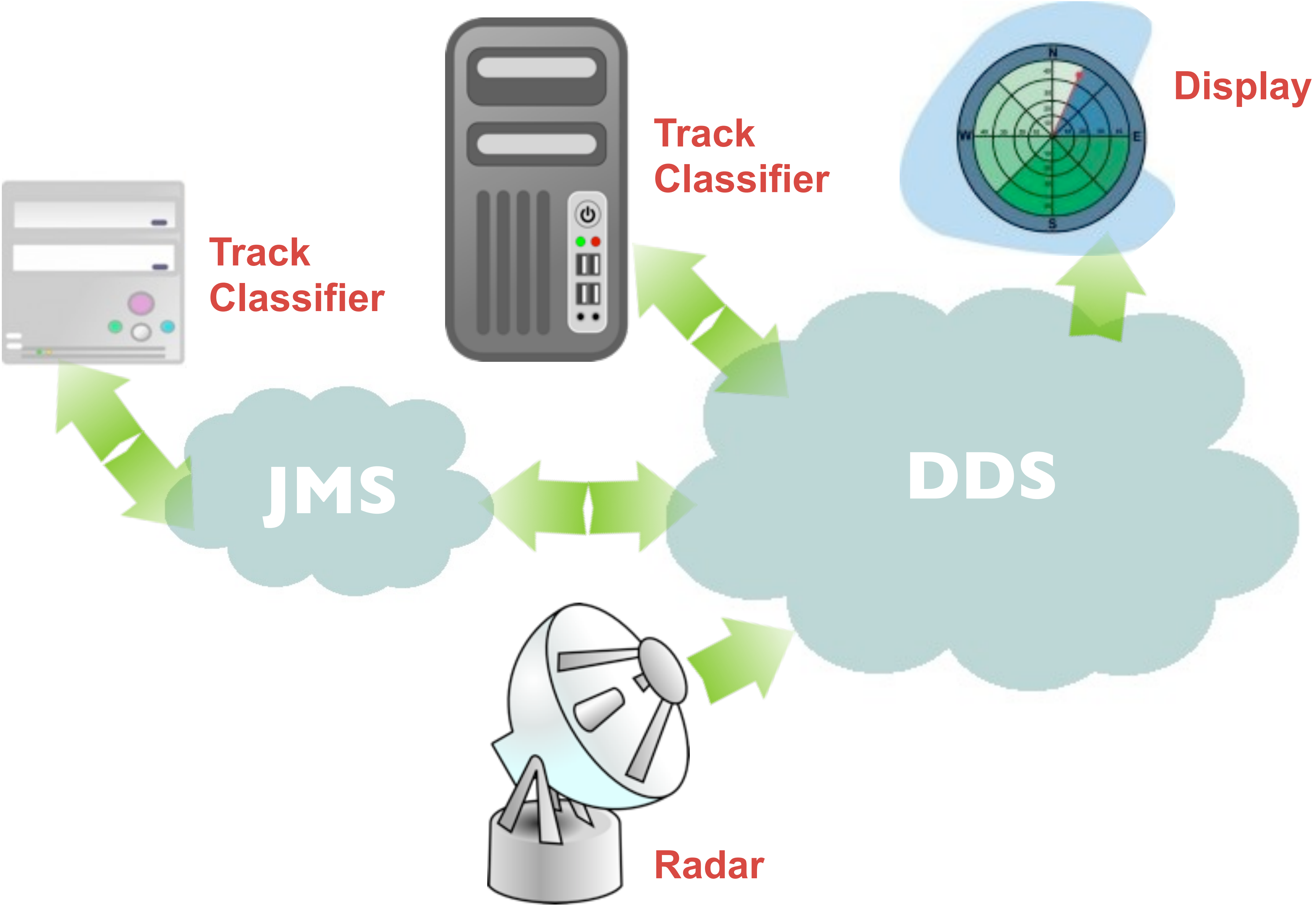
The Universal Data Bus

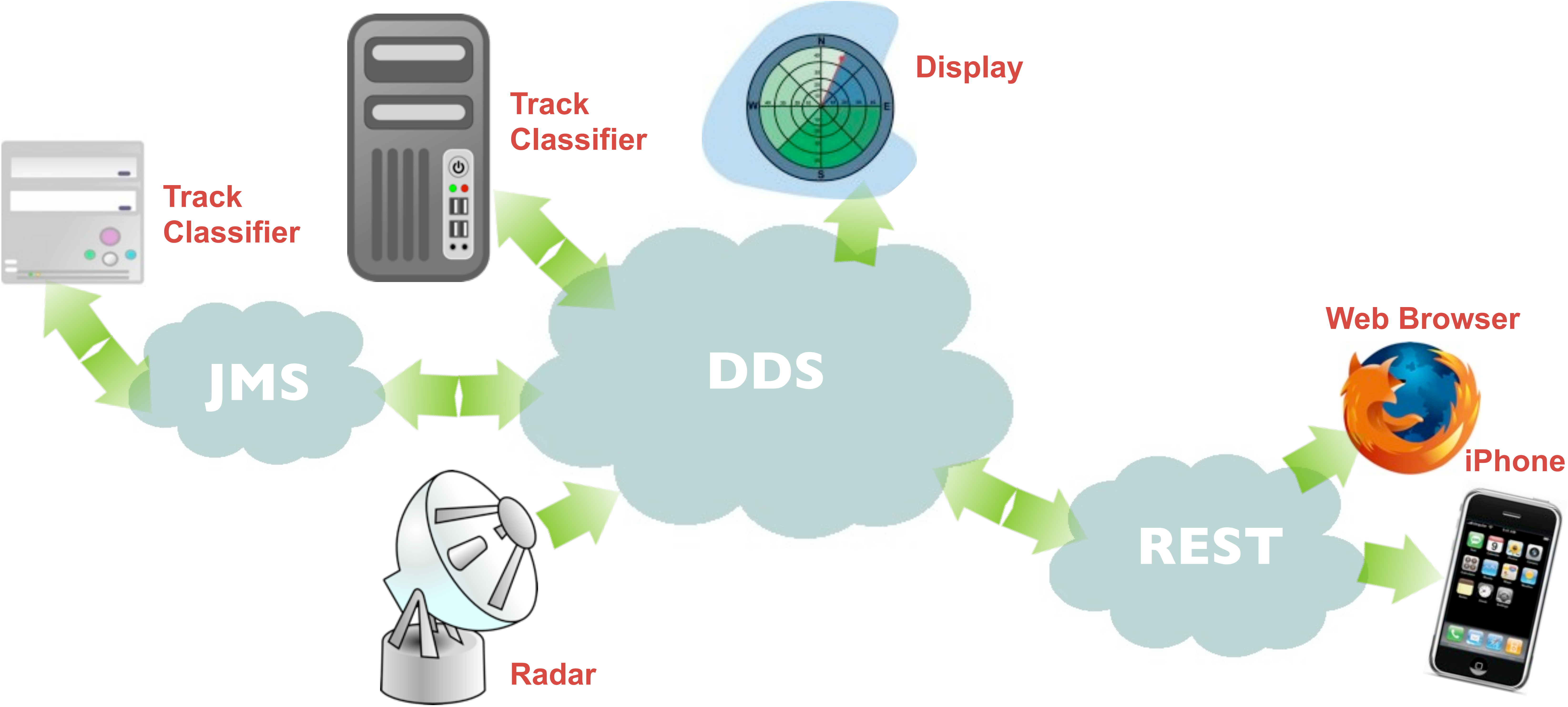
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

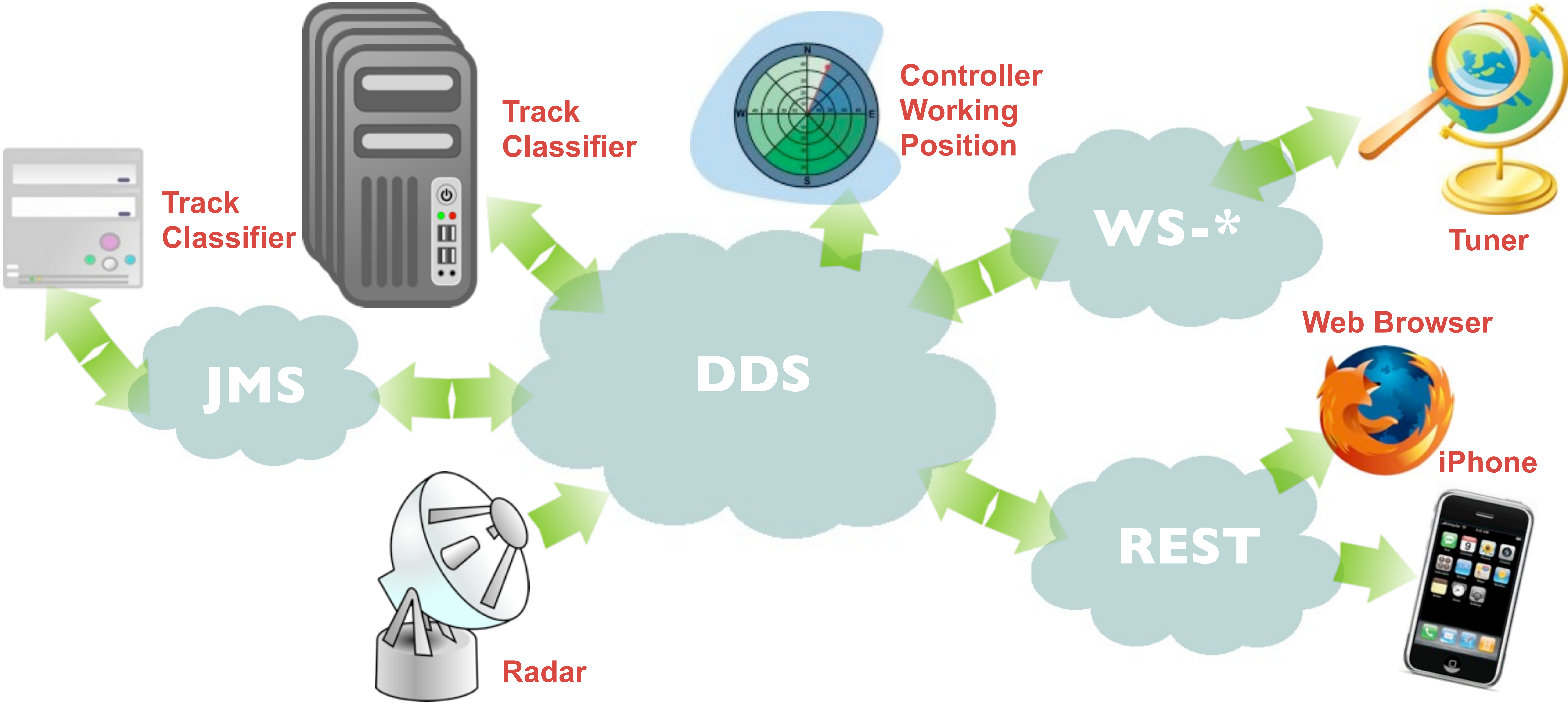
The Technology Integration Challenge

LOOKiNG AHEAD


Track Classifier
Display
DDS
Radar


Track
Classifier
Track
Classifier
Display
JMS
DDS
Radar


Track
Classifier
Track
Classifier
Display
JMS
DDS
Web Browser
iPhone
REST
Radar


Track
Classifier
Controller
Working
Position
Track
Classifier
WS-*
Tuner
Web Browser
JMS
DDS
iPhone
REST
Radar

OpenSplice|DDS

The Universal Data Bus

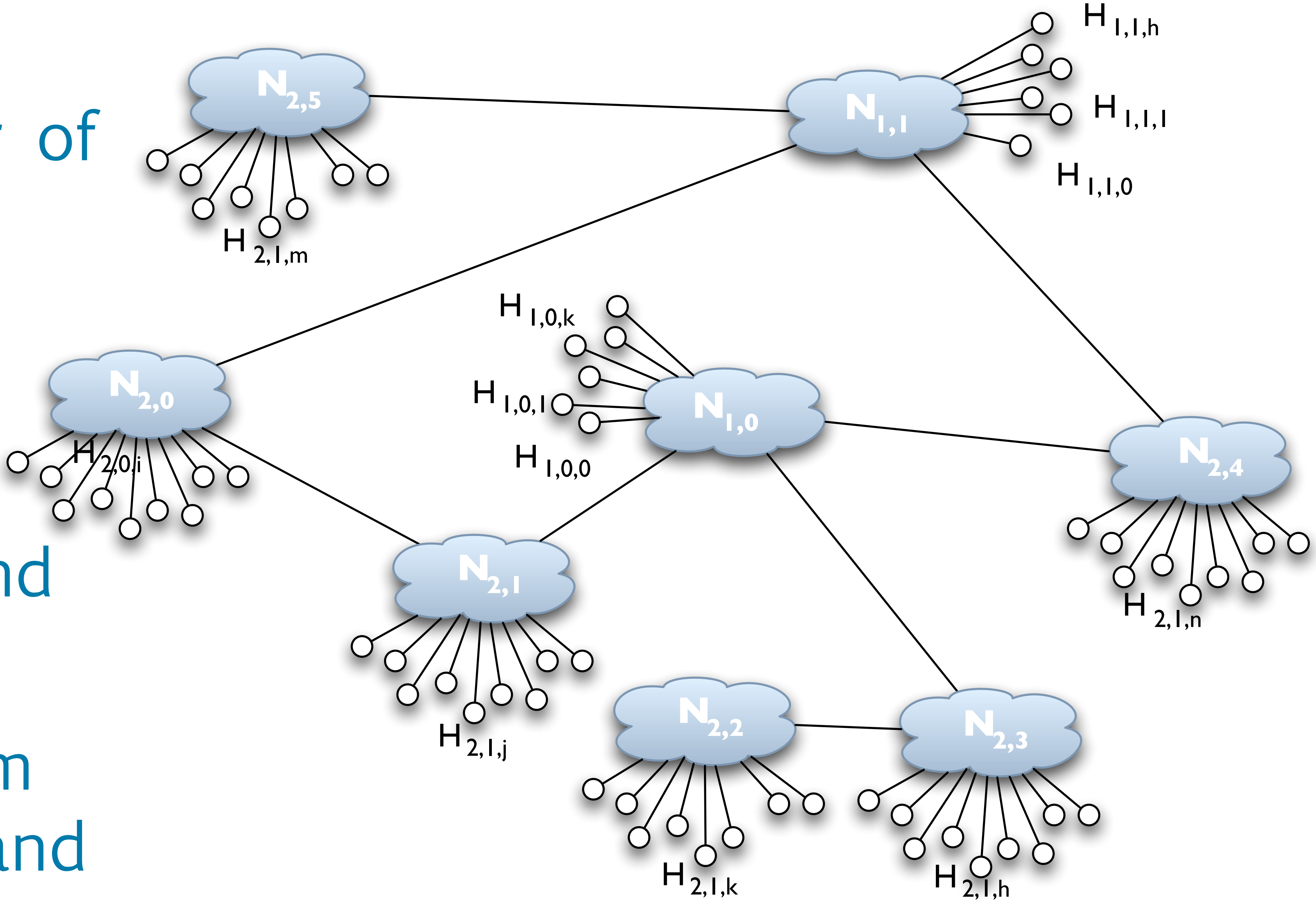
:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

The Ultra Large Scale Systems Challenge

LOOKING AHEAD

Ultra Large Scale (ULS) Systems

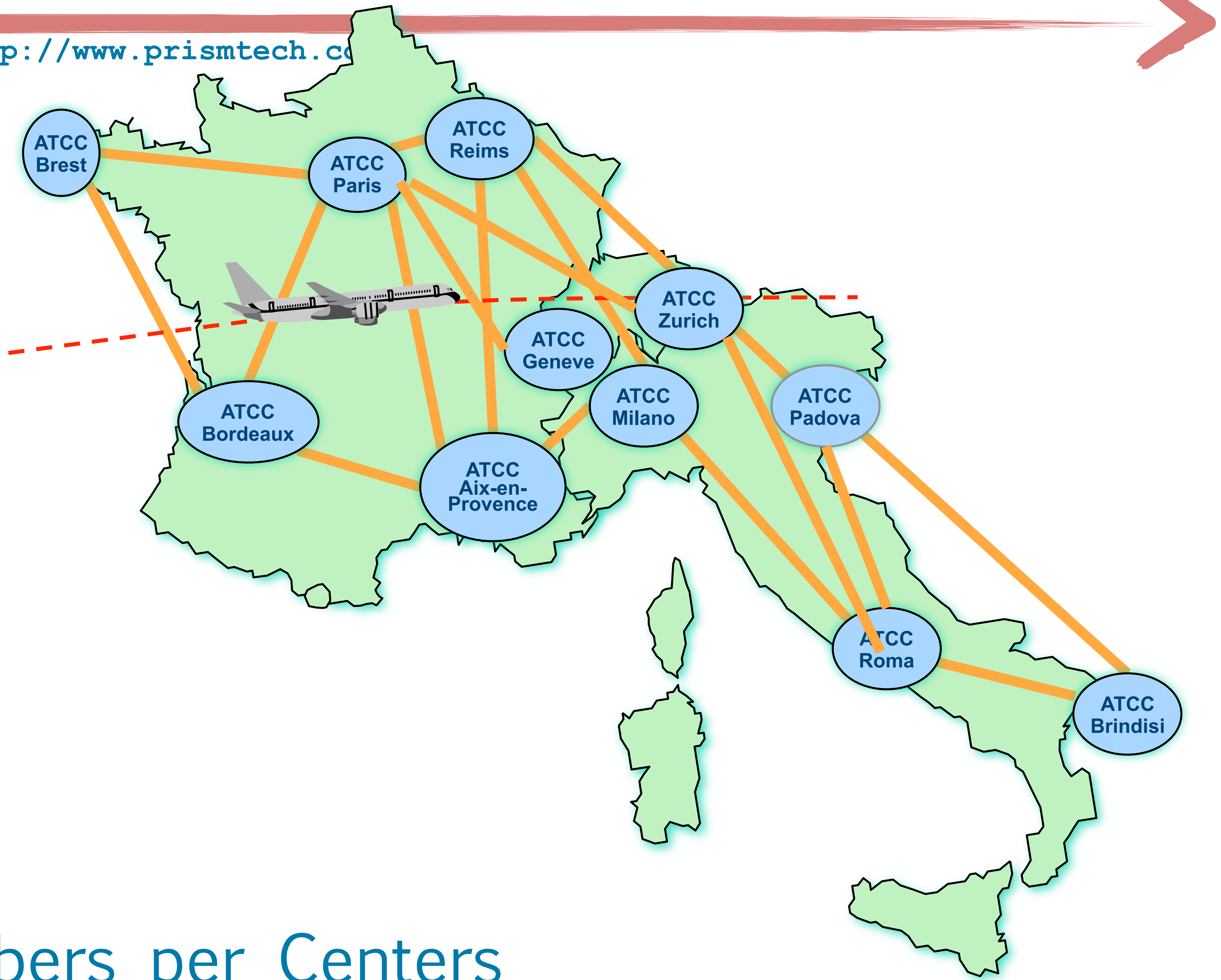
- **System Scale.** Very large number of peer entities that need to communicate in the system
- **Geographical Scale.** The Geographical Scale of the System can span across cities, nations and countries
- **Organizational Scale.** The System can spans across organizational and national boundaries



Air Traffic Control

CoFlight eFDP

- Next-Generation European Flight Data Processor
- All data distribution performed using OpenSplice DDS
- Deployed starting from 2011 in France, Italy, and Switzerland



Some Numbers

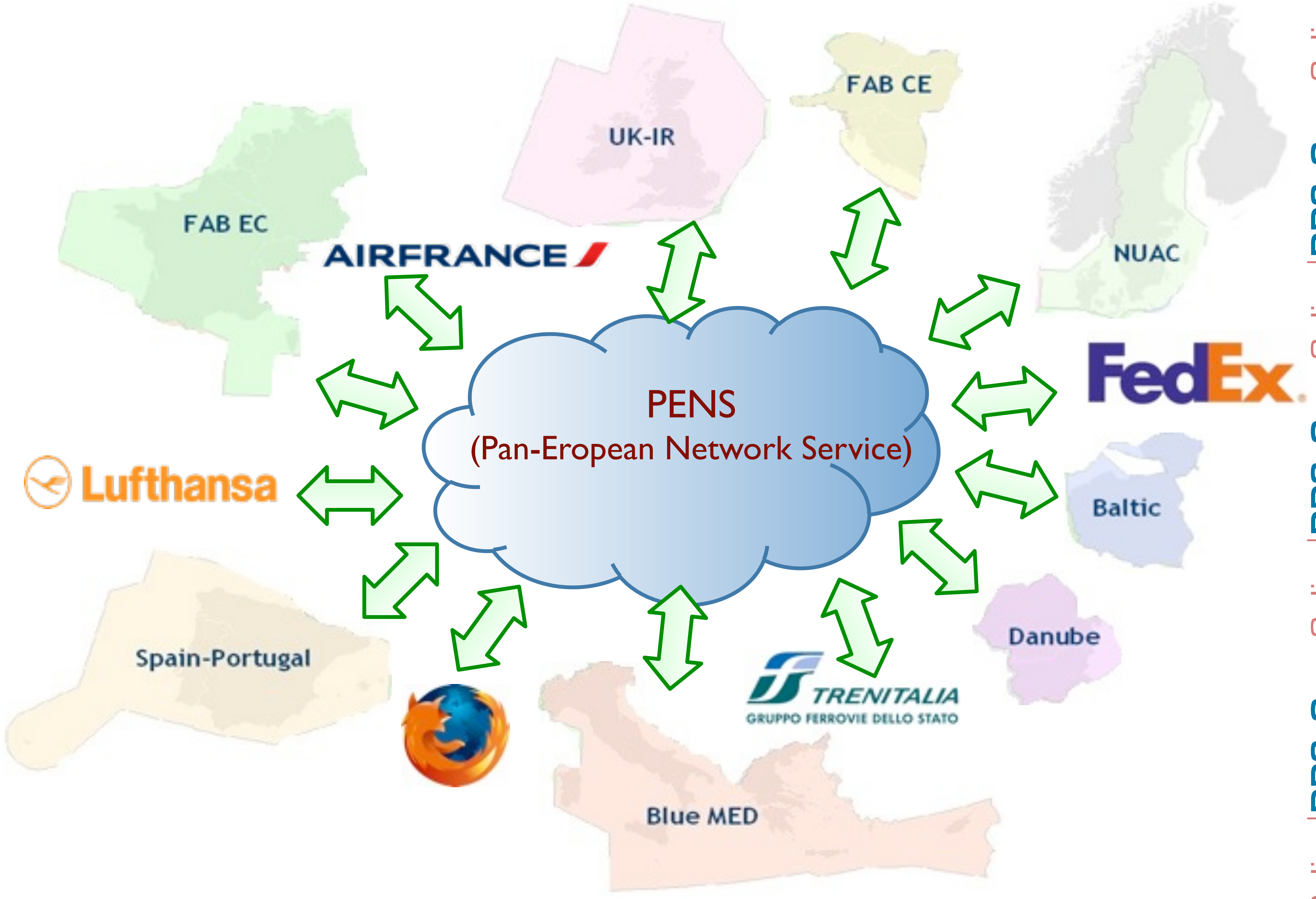
- Hundreds of DDS Publisher/Subscribers per Centers
- 11 Centers Spread across 3 Countries

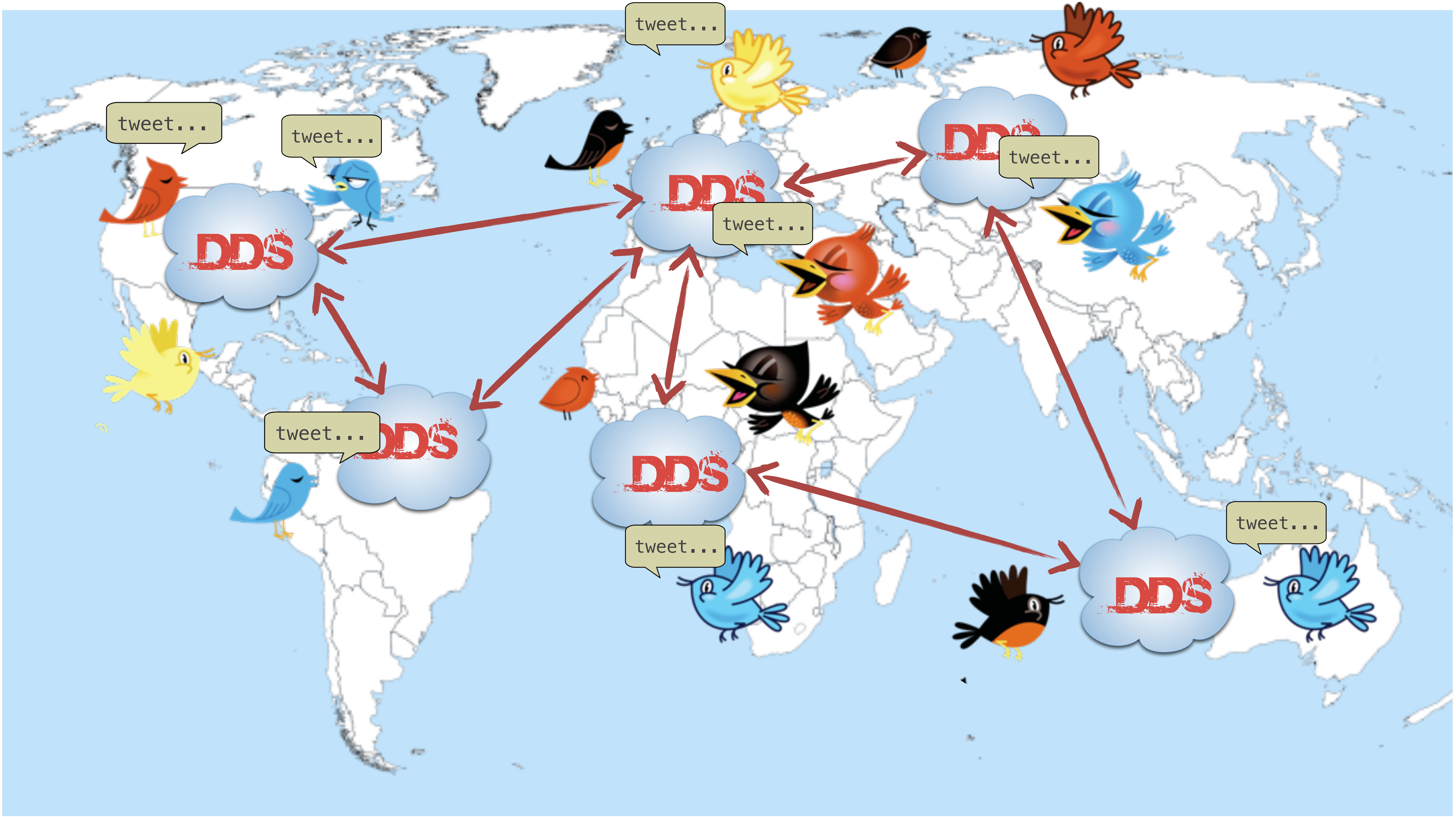
Air Traffic Control

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

SESAR

- Operational integration of all Air Traffic Control Centers Pan-European
- DDS selected as the standard to distribute real-time information Pan-EU




tweet...
tweet...
tweet...
tweet...
tweet...
tweet...
tweet...
tweet...
DDS
DDS
DDS
DDS
DDS
DDS

OpenSplice|DDS

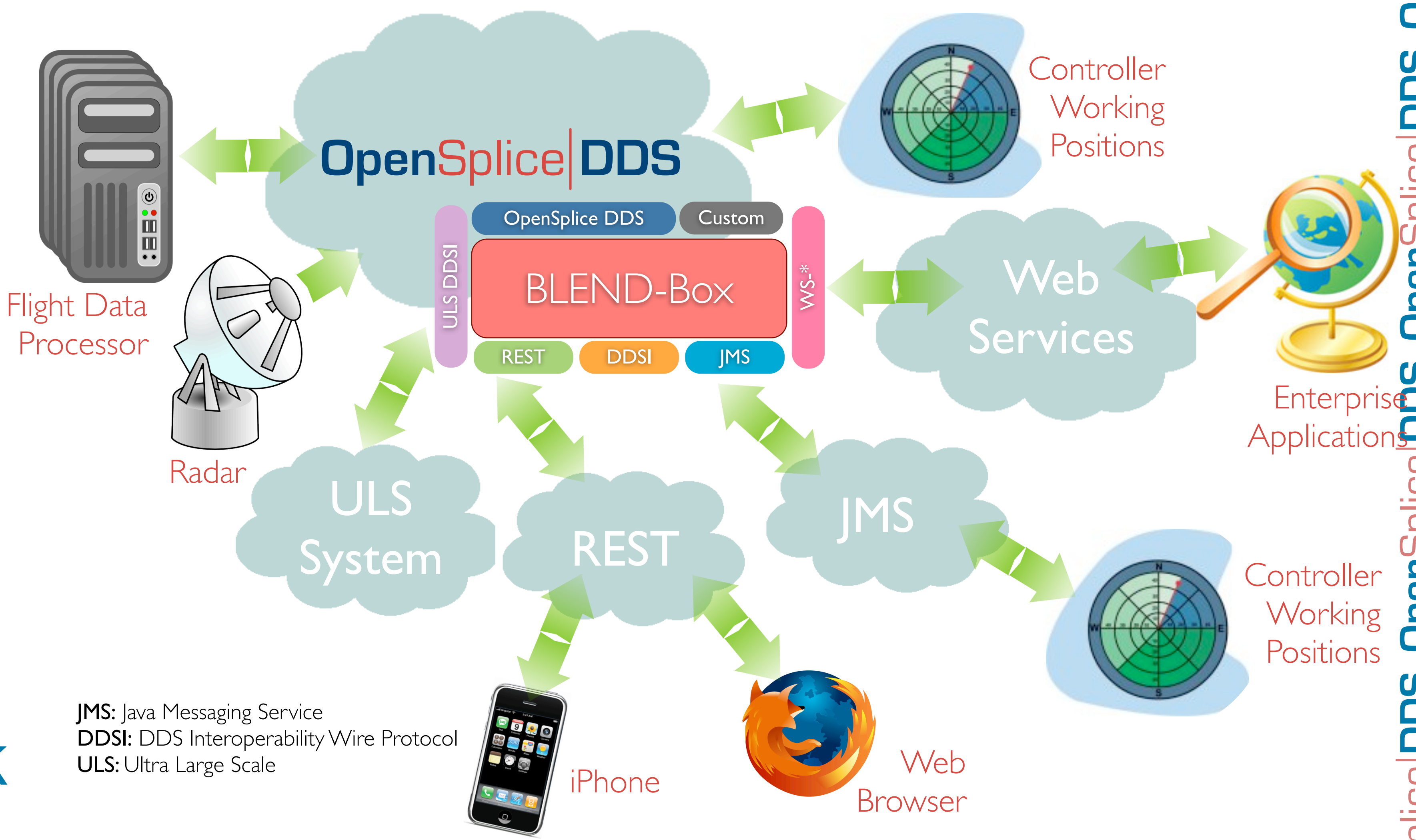
The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

ADDRESSING THE CHALLENGES...

The Blend-Box (B2)

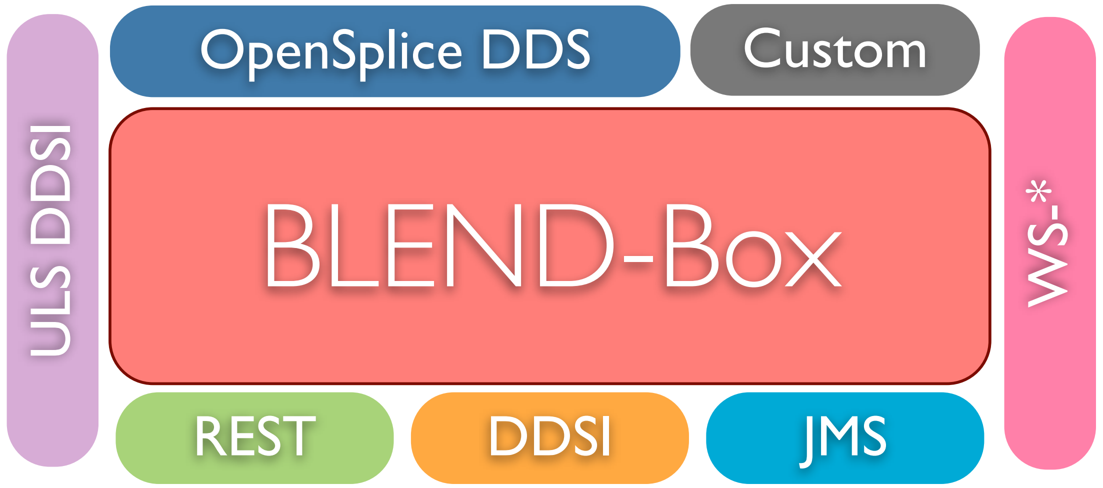
- Real-Time Integration Technology
- Bridges across technologies while adapting Content, Format and QoS
- Full control over what data exposed and how
- Extensible Connector framework compatible with Camel

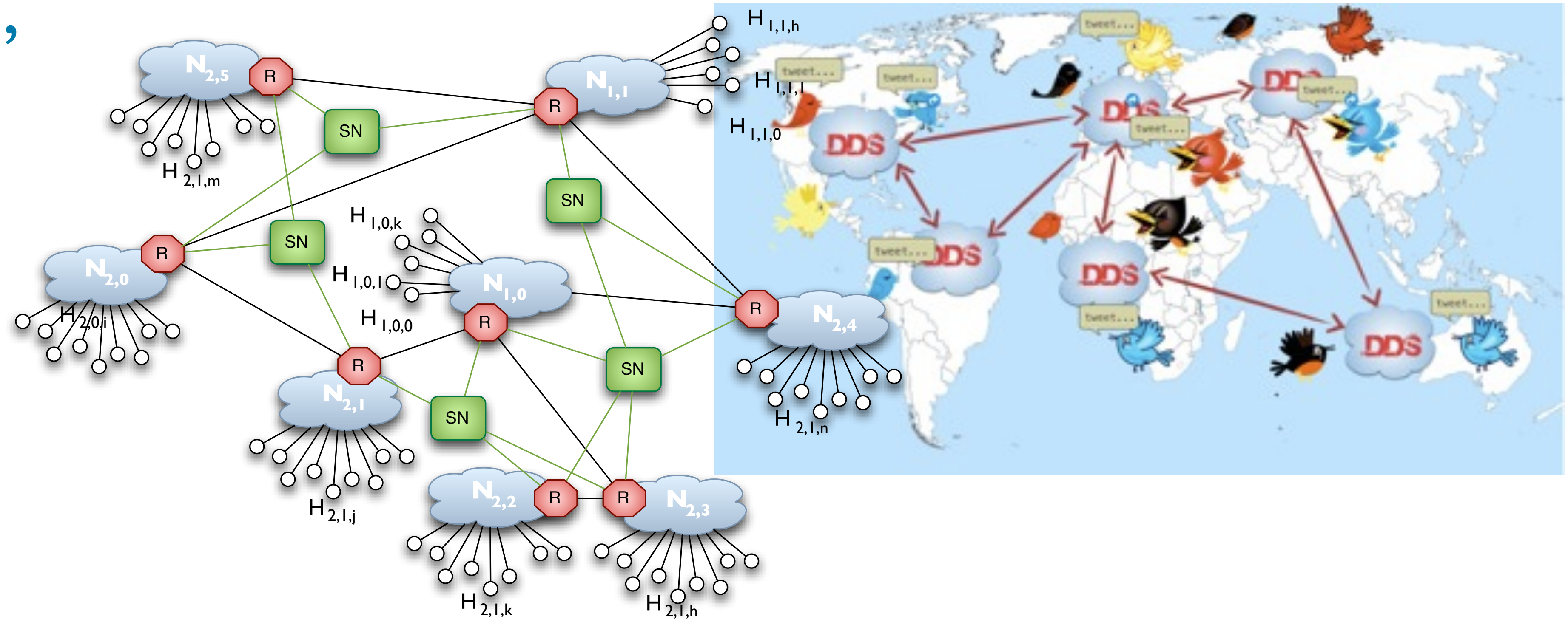


JMS: Java Messaging Service
DDSI: DDS Interoperability Wire Protocol
ULS: Ultra Large Scale

B2 and ULS Systems

- ULS DDSI provides scalable discovery and efficient WAN communication
- WS-*, REST, RSS, Atom, etc., Connectors based on the upcoming Web-Enabled DDS Standard





OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

SELECTED USE-CASES

TACTICOS CMS

- OpenSplice DDS enables the core infrastructure of the TACTICOS Naval Combat Management System from THALES Naval Netherlands'
- OpenSplice DDS provide TACTICOS with is renown high-availability, reconfigurability, and scalability which make it possible to scale from small ships to carrier-grade ships
- TACTICOS numbers are today:
    - Deployed on 26 ships classes, >100 ships
    - 2.000+ deployed runtimes (running on Solaris-servers, Linux-consoles, & vxWorks embedded subsystems)
    - 15 Navies worldwide (Netherlands, Germany, Turkey, Greece, Oman, Qatar, Portugal, South Korea, Japan, Poland,...)

Vetronics

- OpenSplice DDS adopted as the foundation for the electronic architecture of next generation Vehicle Architecture

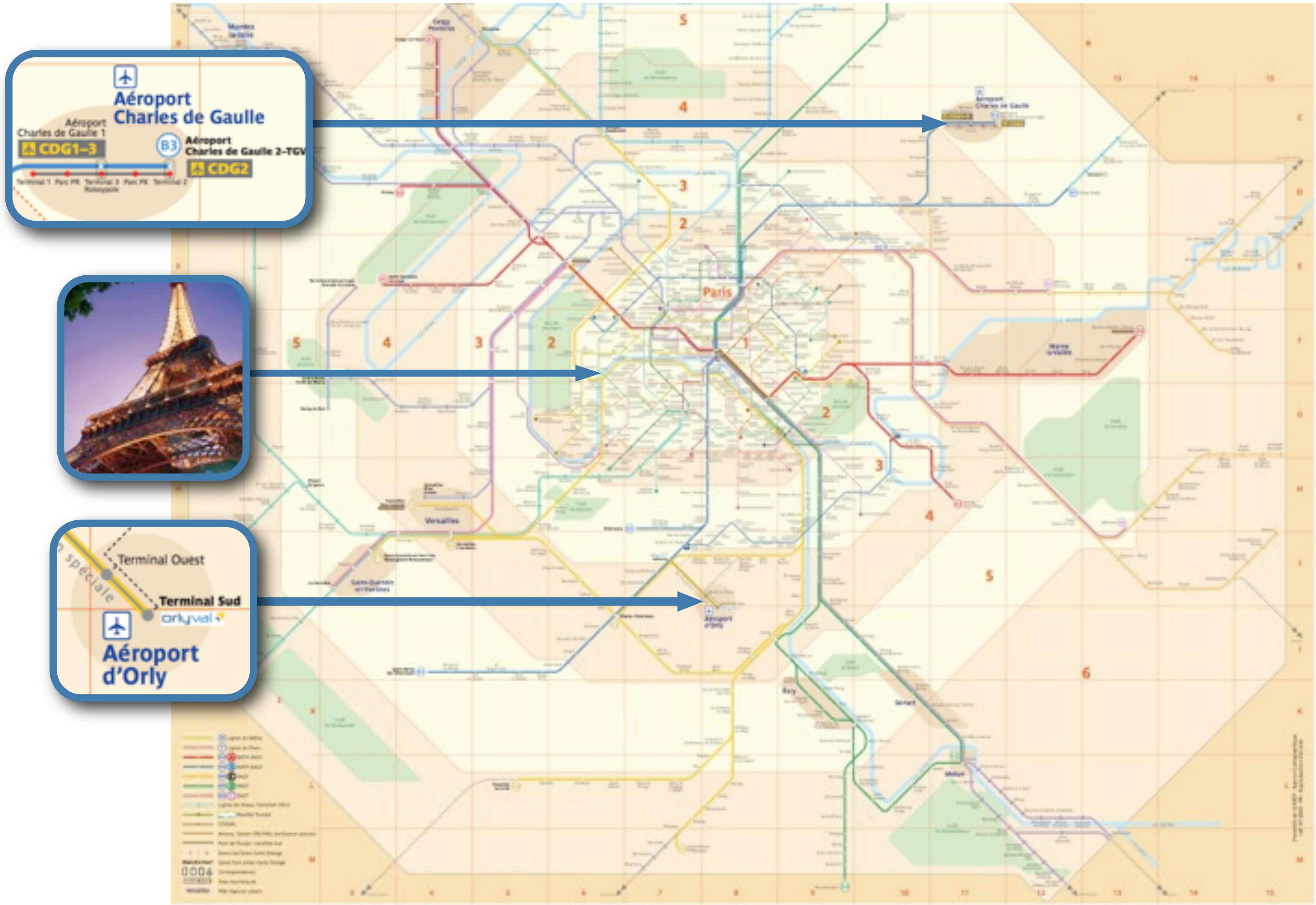
- OpenSplice DDS takes care of distributing real-time sensor data for all sources but video

- An OpenSplice DDS Remote Method Invocation extension is also used to issues all commands

Regie Autonome des Transports Parisiens

Transport Parisien

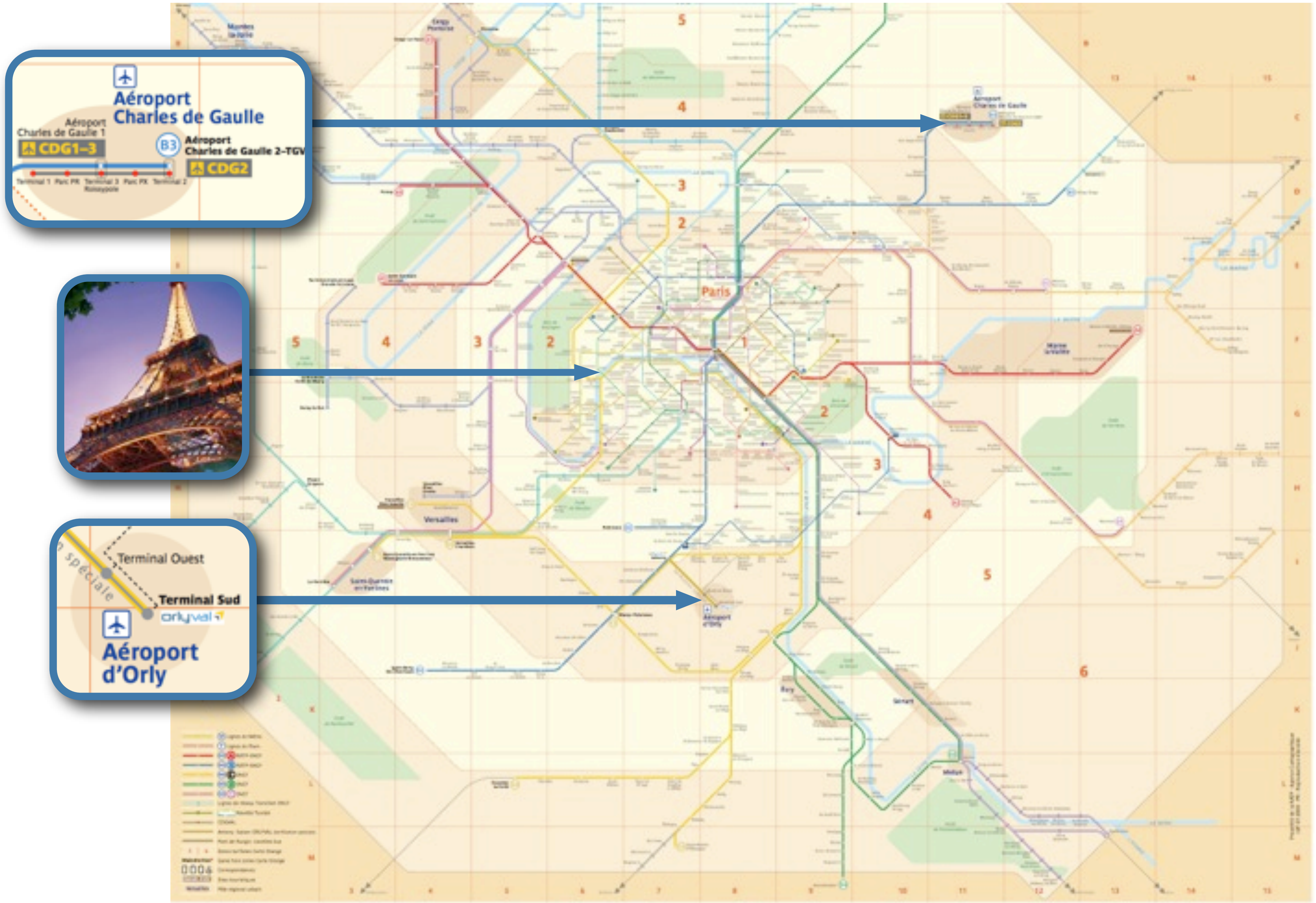
- Manage and Supervise the equipment deployed in the stations of Paris Metro/RER/Bus/Tram



Regie Autonome des Transports Parisiens

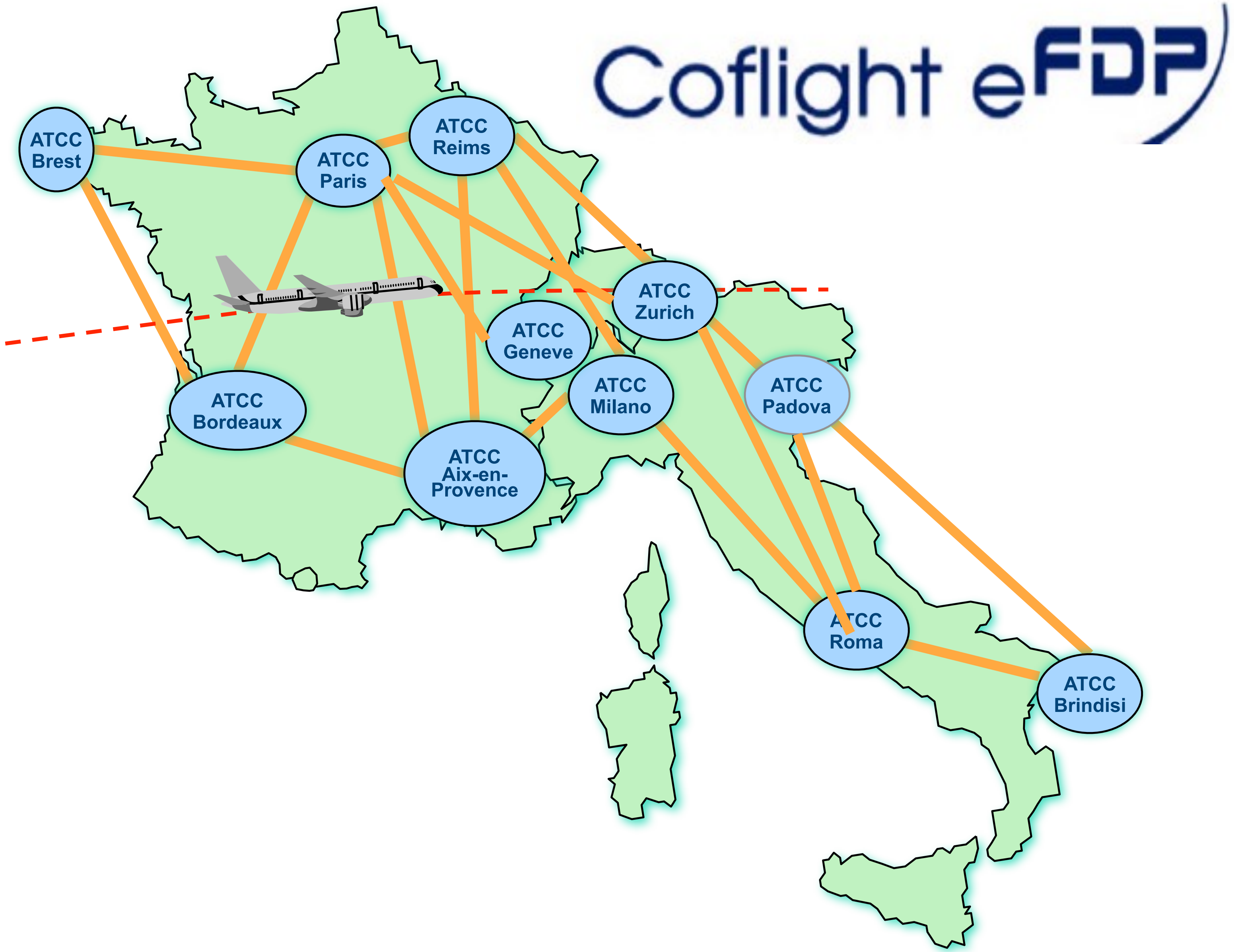
Some Numbers

- 214 Km of Metro lines with over 300 stations
- 115 Km of rapid transit network (RER)
- 30.2 Km of Tramway lines and over 569 Km of bus lines
- Several millions of passengers per day
- 50000 Supervised Equipments



EU Flight Data Processor

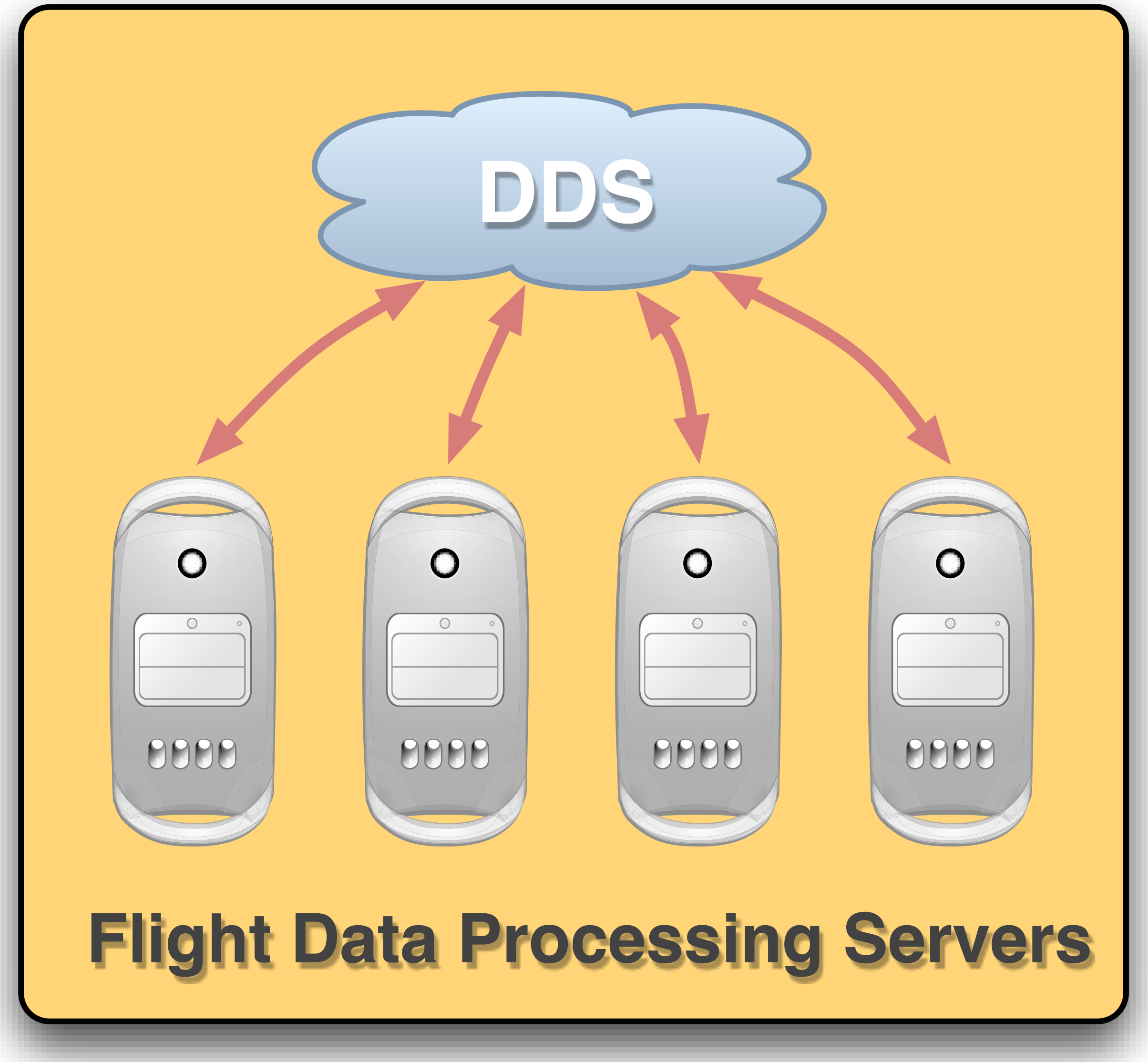
- Large program to replace existing Flight Data Processors (FDPs)
  - 5 Centers in France
  - 4 Centers in Italy
  - 2 Centers in Switzerland



DDS in CoFlight -- FDP Core

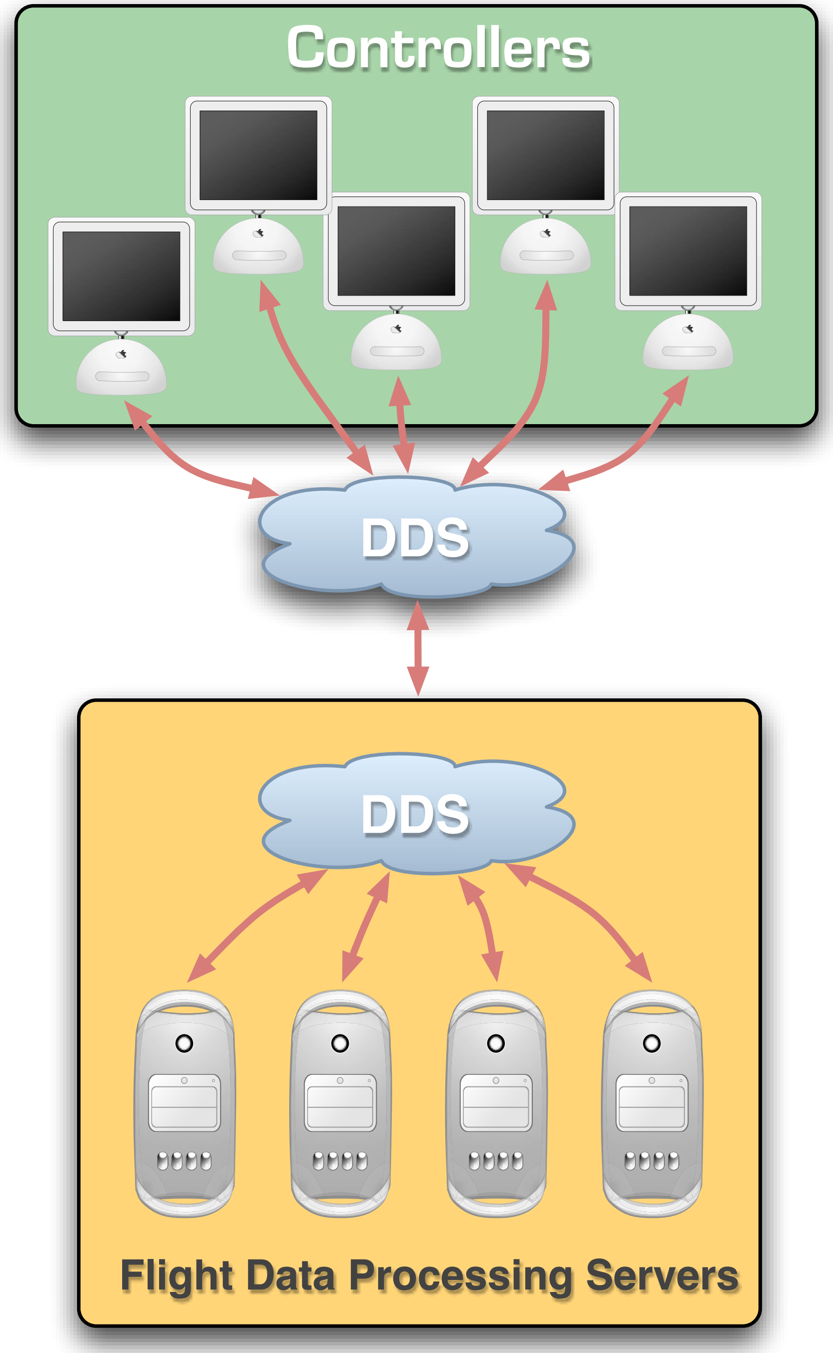
- OpenSplice DDS glues together the most critical components of the CoFlight FDP running at a **SWAL-2** (same as **DO-178B Level B**) assurance level

- In this context OpenSplice DDS distributes flights data plans of redundant LANs



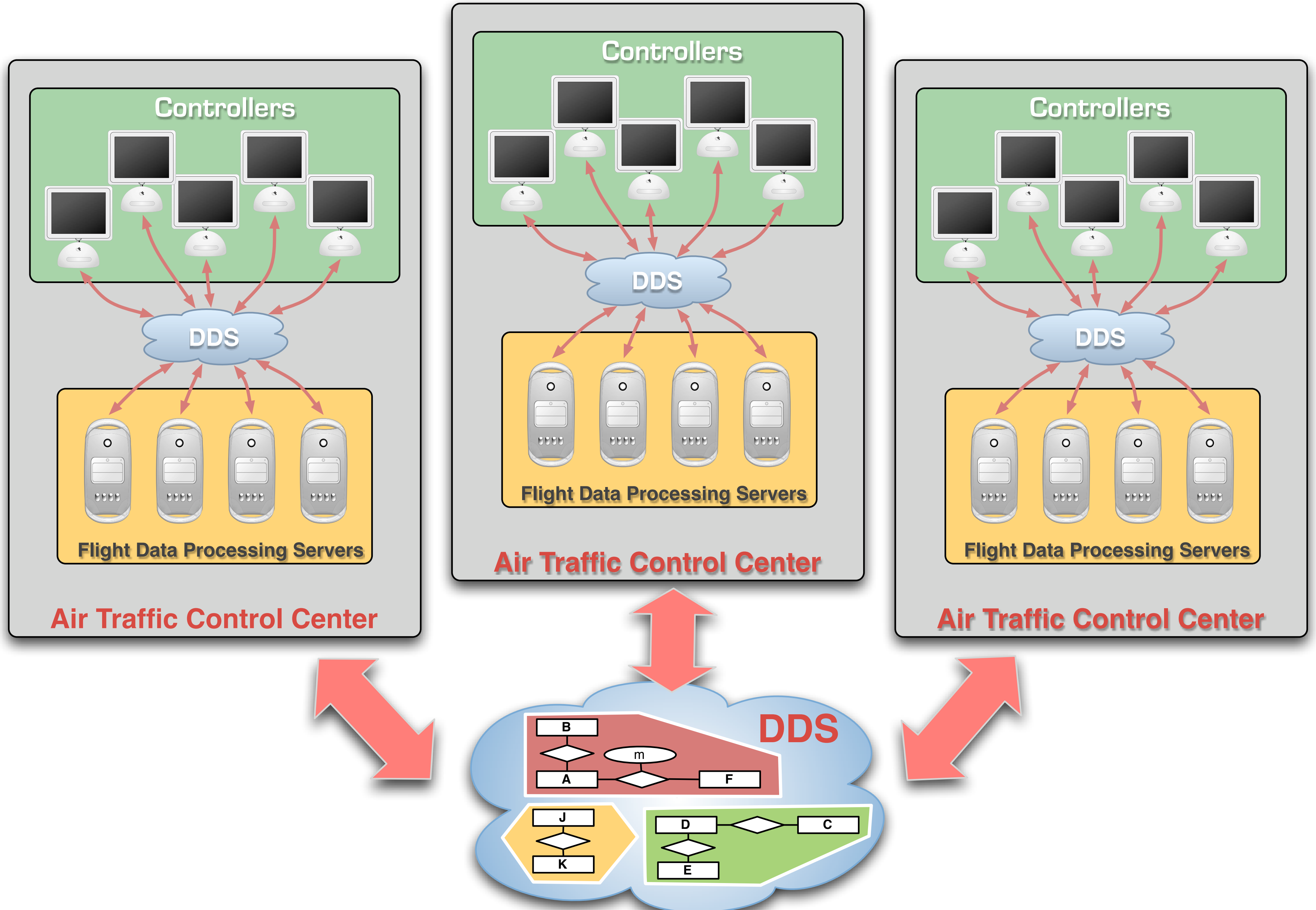
DDS in CoFlight -- CWP

- OpenSplice DDS is used within CoFlight to distribute the “external” Flight Data Plan to Controller Working Positions, to Control Towers, etc.



DDS in CoFlight -- IOP

- OpenSplice DDS is used to integrate CoFlight-based Centers
- OpenSplice DDS is used to provide interoperability with other Interoperable Centers (as per ICOG-2)


Controllers
DDS
Flight Data Processing Servers
Air Traffic Control Center
Controllers
DDS
Flight Data Processing Servers
Air Traffic Control Center
Controllers
DDS
Flight Data Processing Servers
Air Traffic Control Center
DDS
B
m
A
F
J
K
D
C
E

More Use Cases...

Defense & Aerospace

- Naval Combat Management Systems
- Submarines
- Vetronics
- Tactical Links
- Simulation
- Cybercrime
- Flycatcher Systems
- Data Fusion
- Battle Transformation Center

More Use Cases...

Transportation

- Drones
- Air Traffic Control & Management
- Metropolitan Transportation

Financial Services

- Automated Trading Firms
- Risk Management Firms

Some OpenSplice DDS Users

nexter
Rockwell Collins
MIT
Massachusetts Institute of Technology
L3
SIEMENS
NORTHROP GRUMMAN
LOCKHEED MARTIN
CISCO SYSTEMS
NAVCOM
A John Deere Company
NASA
GE
THALES
Raytheon
ITT Industries
Engineered for life
BOEING
Think Trade LLC
EMBRAER
RATP
BAE SYSTEMS
SELEX
GENERAL DYNAMICS
Indra
HARRIS
EADS
ProRail
Ultra ELECTRONICS

OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

SUMMiNG-UP

+/-*

In Summary

- The OMG DDS is the most advanced standard-based technology for real-time data distribution
- DDS provides a powerful set of abstractions to build distributed high-availability and high performance systems
- Implementation of the standard, such as OpenSplice DDS are deployed in several extremely challenging mission/business critical systems

REFERENCES

OpenSplice DDS

Delivering Performance, Openness, and Freedom

- http://www.opensplice.com/
- http://www.opensplice.org/
- emailto:openspliceds@prismtech.com

slideshare Present Yourself

- http://www.slideshare.net/angelo.corsaro

webex™

- http://bit.ly/1Sreg

twitter

- http://twitter.com/acorsaro/

YouTube

- http://www.youtube.com/OpenSpliceTube

Blogger

- http://opensplice.blogspot.com

DownLoad OpenSplice DDS at http://www.opensplice.org

OpenSplice|DDS

The Universal Data Bus

:: http://www.opensplice.org :: http://www.opensplice.com :: http://www.prismtech.com ::

THANK YOU!

P.S. Enjoyed the talk? Found it Bloody boring? Send me your comments at: angelo.corsaro@prismtech.com