The Middleware Experts

DDS Advanced Tutorial

Best-Practice Data-Centric Programming with DDS

OMG Tech Meeting, Berlin 2013

19/06/2013

Jaime Martin Losa
CTO eProsima
JaimeMartin@eProsima.com
+34 607 91 37 45

www.eProsima.com

In cooperation with RTI www.rti.com

Agenda

- DDS
  - Introduction
  - Architecture
  - Hands on
  - Common Use Cases
  - Best Practices
- Success Cases

The Middleware Experts

Introduction and Background

Introduction: Everything is distributed

- Enterprise Internet
- Internet of Things
- Cloud Computing
- Industry 4.0
- …

Next-generation systems needs:

- Scalability
- Integration & Evolution
- Robustness & Availability
- Performance
- Security

eProsima

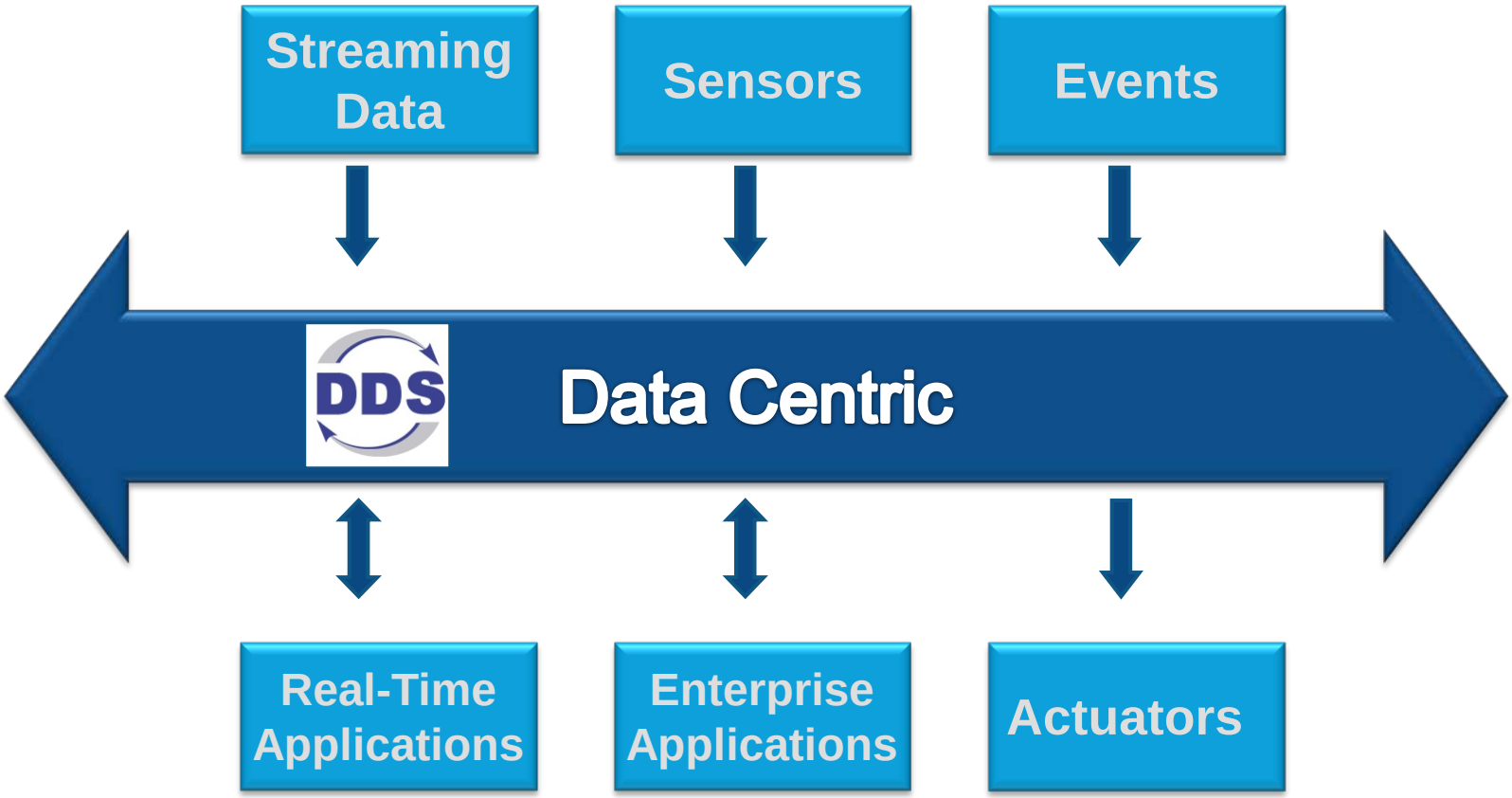
“Real World” Systems are integrated using a Data Model

- Grounded on the “physics” of the problem domain
  - Tied to the nature of the sensors and real objects in the system (vehicles, device types, …)
- Provides governance across disparate teams & organizations
  - The “N^2” integration problem is reduced to a “N” problem
- Increased decoupling from use-cases and components
  - Avoids over constraining applications
- Open, Evolvable, Platform-Independent
  - The use-cases, algorithms might change between missions or versions of the system

Challenge

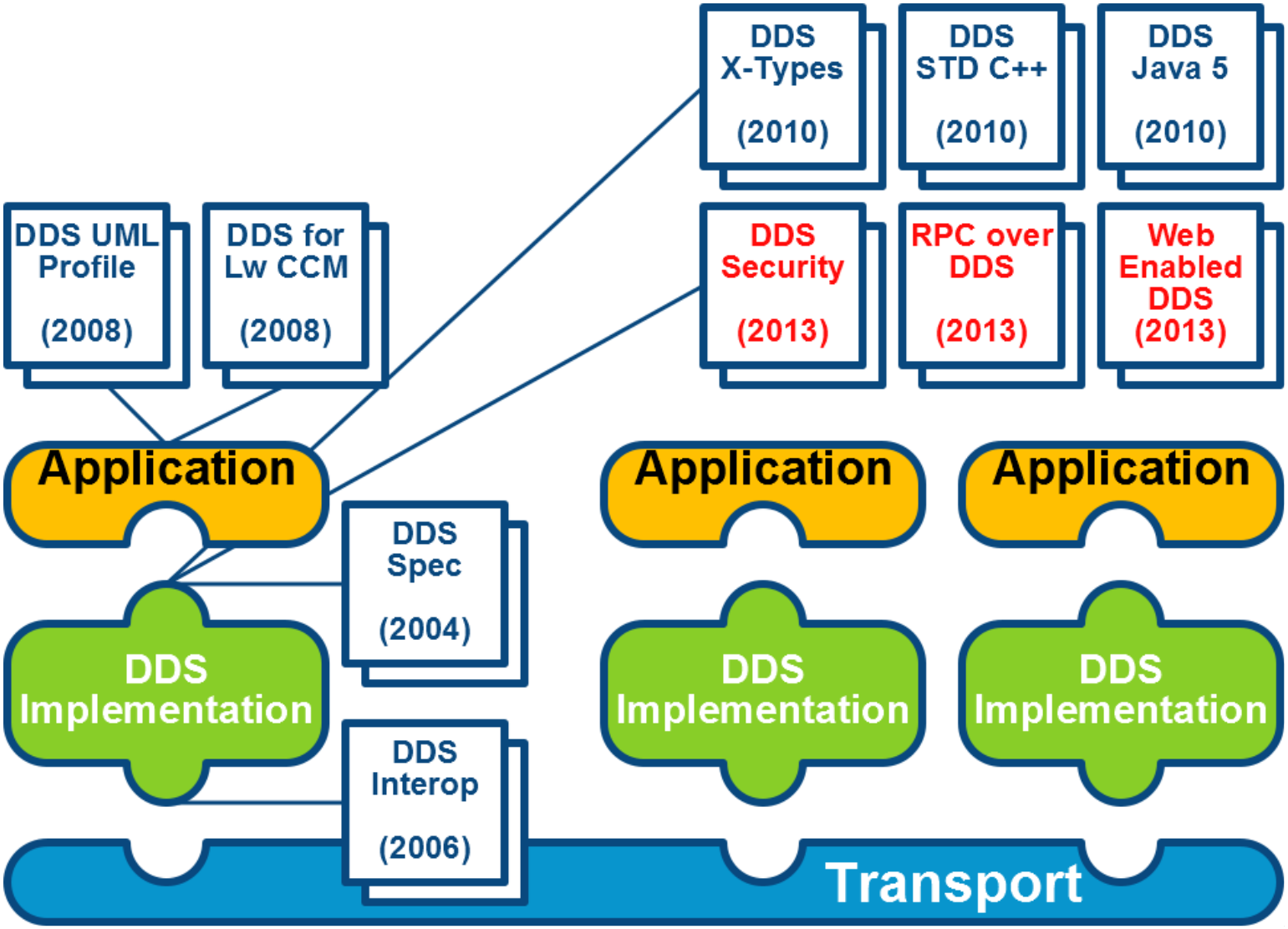
- Everything is connected, and we should enable communication between the different nodes.
- And this means:
  - Common protocols
  - Common Data Types
  - Known interfaces
  - Different QoS over different datalinks and performance requirements.
  - Different comunications patterns.
  - Broad platform and programming language support.
  - Good Data Models!
  - …

DDS: Standards-based Integration Infrastructure for Critical Applications



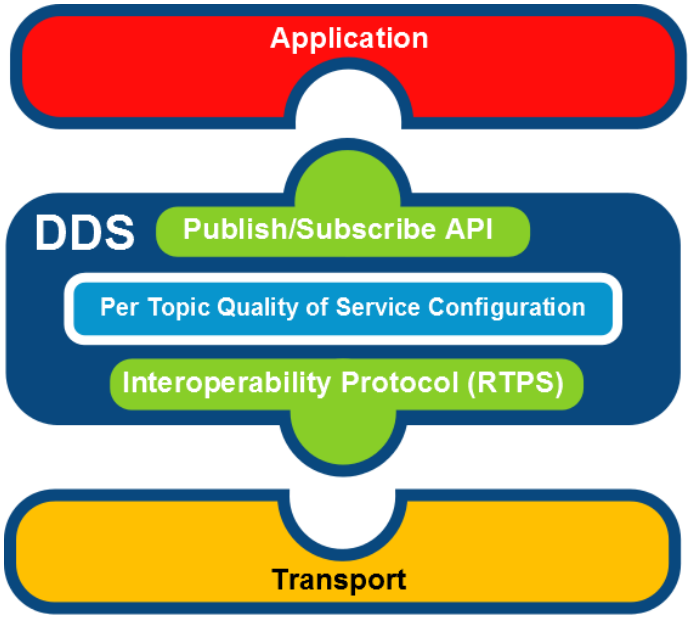
eProsima

DDS Family of Specifications



Broad Adoption

- Vendor independent
  - API for **portability**
  - Wire protocol for **interoperability**
- Multiple implementations
  - 10 of API
  - 8 support RTPS
- Heterogeneous
  - C, C++, Java, .NET (C#, C++/CLI)
  - Linux, Windows, VxWorks, other embedded & real time
- Loosely coupled



DDS adopted by key programs in Europe

- *European Air Traffic Control*
  - *DDS proposed for interoperate ATC centers*
- *Spanish Army*
  - *DDS is mandated for C2 Interoperability (ethernet, radio & satellite)*
- *UK Generic Vehicle Architecture*
  - *Mandates DDS for vehicle comm.*
  - *Mandates DDS-RTPS for interop.*

US-DoD mandates DDS for data-distribution

- DISR (formerly JTA)
  - DoD Information Technology Standards Registry
- US Navy Open Architecture
- Army, OSD
  - UCS, Unmanned Vehicle Control
- *SPAWAR NESI*
  - *Net-centric Enterprise Solutions for Interoperability*
  - *Mandates DDS for Pub-Sub SOA*

The Middleware Experts

DDS Architecture

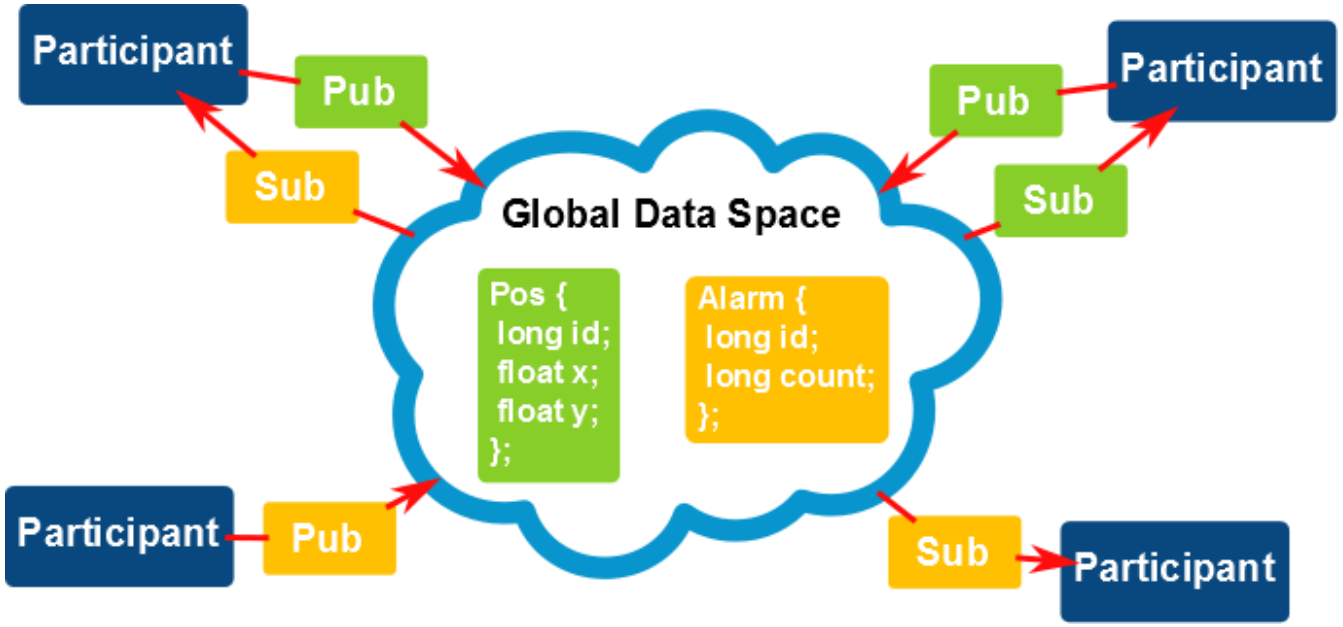
DDS

- DDS (Data Distribution Service for Real-Time Systems) is a OMG specification for a pub/sub data centric model (DCPS, Data Centric Publish/Subscribe) for Real-Time data comms in distributed systems.

- DDS is a networking middleware that:
  - Simplifies and Standardizes data flows in distributed real-time systems.
  - Provides robust comms (no single point of failure) and efficient (minimum latency)
  - Provides all kind of QoS to shape the data flows and deliver predictable results.

DDS

DDS uses the concept of **Global Data Space**. In this Space we define **topics** of data, and the **publishers** publish samples of these topics. DDS distributes these samples to all the **subscribers** of those topics. Any node can be a publisher or a subscriber.



Why DDS? Decoupled model

- **Space (location)**
  - Automatic Discovery ensures network topology independence
- **Redundancy:**
  - It is possible to configure redundant publishers and subscribers, primary/secundary and takeover schemas supported
- **Time:**
  - The reception of data does not need to be synchronous with the writing. A subscriber may, if so configured, receive data that was written even before the subscriber joined the network.
- **Platform:**
  - Applications do not have to worry about data representation, processor architecture, Operating System, or even programming language on the other side
- **Implementation:**
  - DDS Protocol is also an standard. Different implementations interoperate.

Why DDS? Fully configurable

Volatility

Infrastructure

Delivery

QoS Policy
DURABILITY
HISTORY
READER DATA LIFECYCLE
WRITER DATA LIFECYCLE
LIFESPAN

ENTITY FACTORY
RESOURCE LIMITS

RELIABILITY
TIME BASED FILTER
DEADLINE
CONTENT FILTERS

User QoS

Presentation

Redundancy

Transport

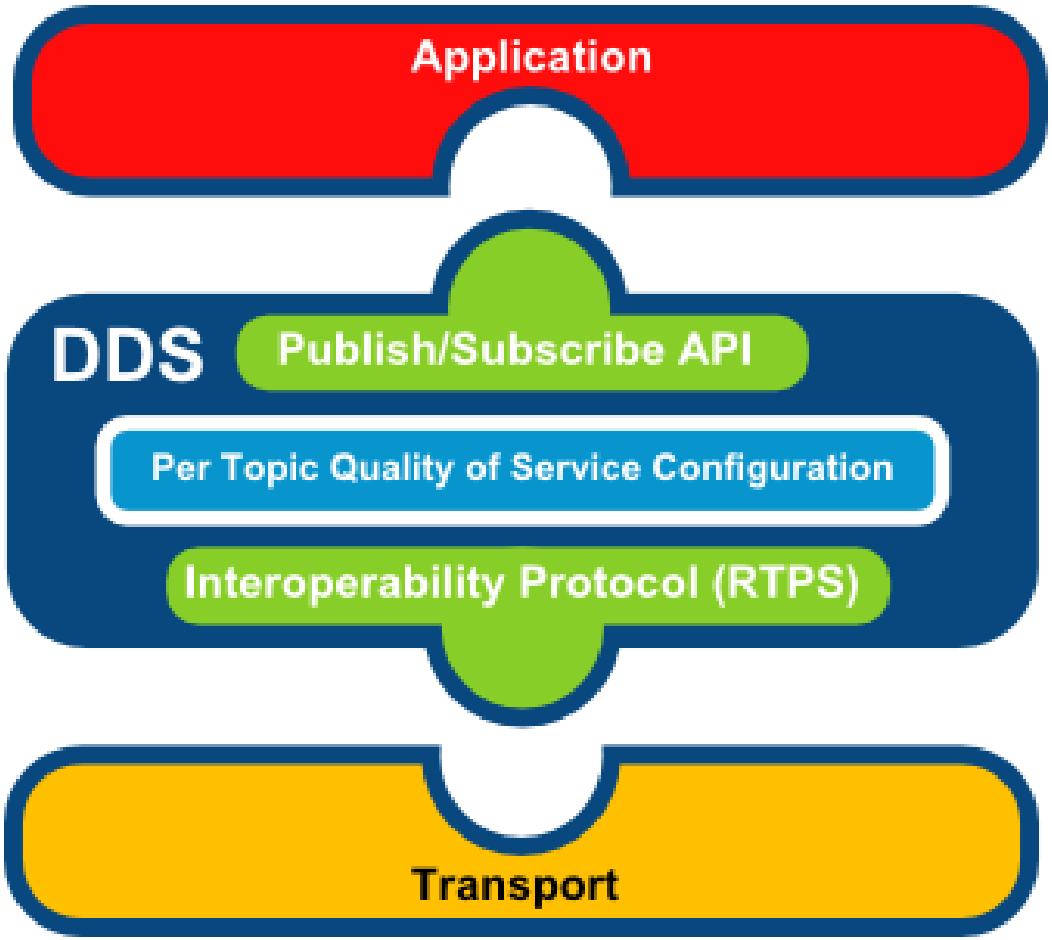
QoS Policy
USER DATA
TOPIC DATA
GROUP DATA

PARTITION
PRESENTATION
DESTINATION ORDER

OWNERSHIP
OWNERSHIP STRENGTH
LIVELINESS

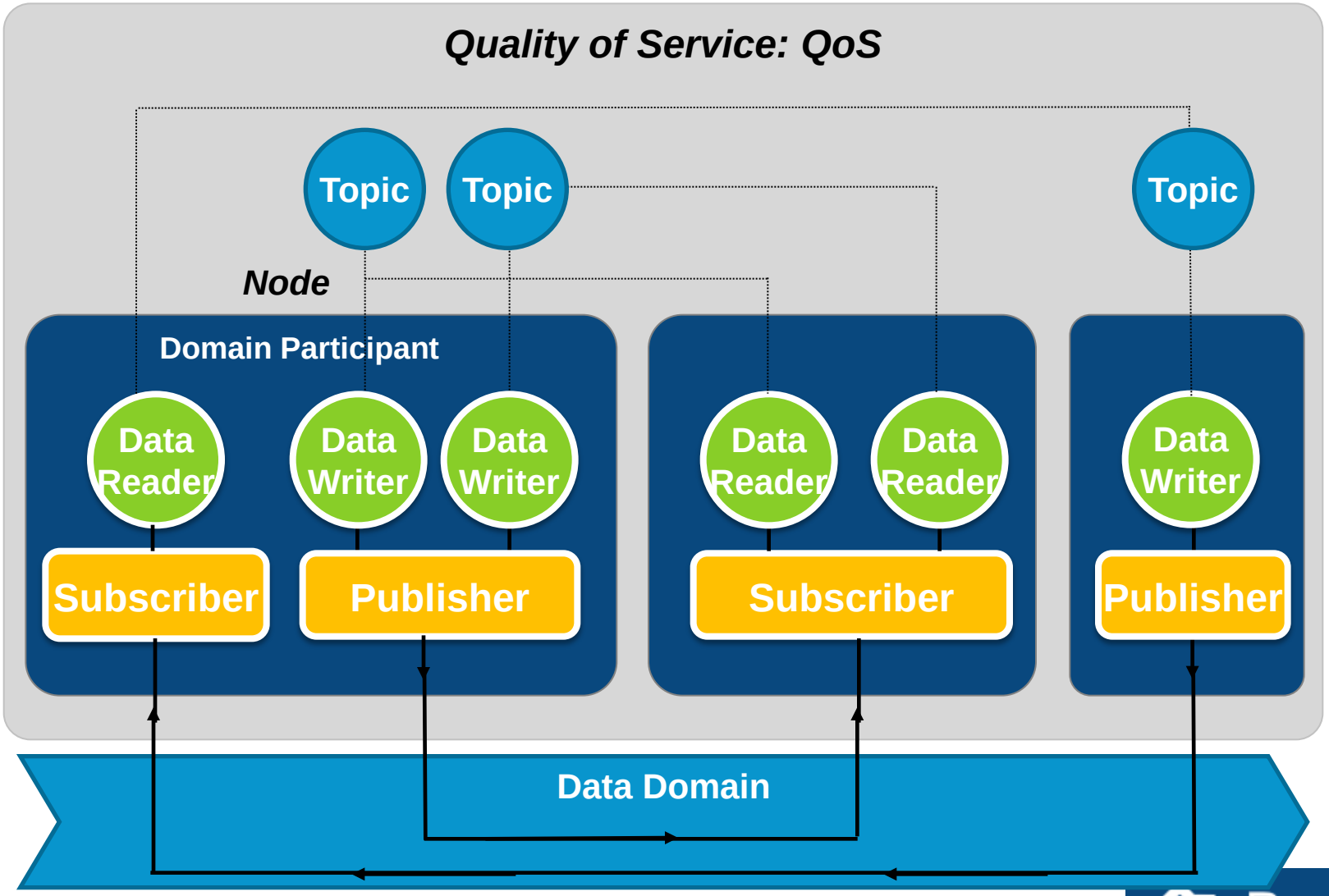
LATENCY BUDGET
TRANSPORT PRIORITY

DDS Infrastructure



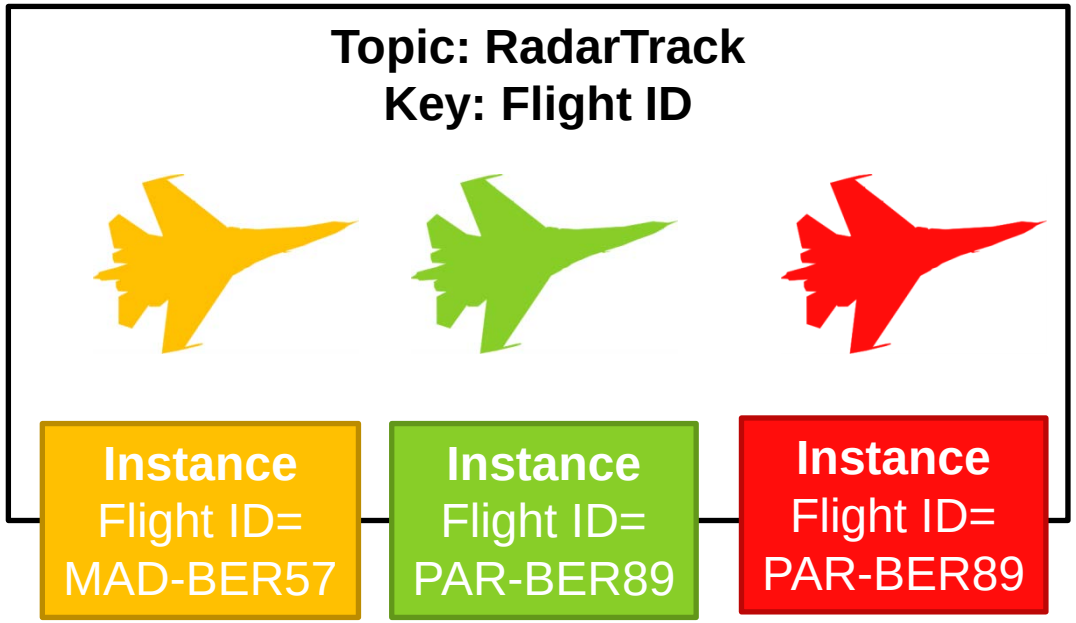
- Standard API for portability.
- RTPS can be implemented over any transport
- No central Broker/Service
- Different Comm channel per topic

The DDS Model



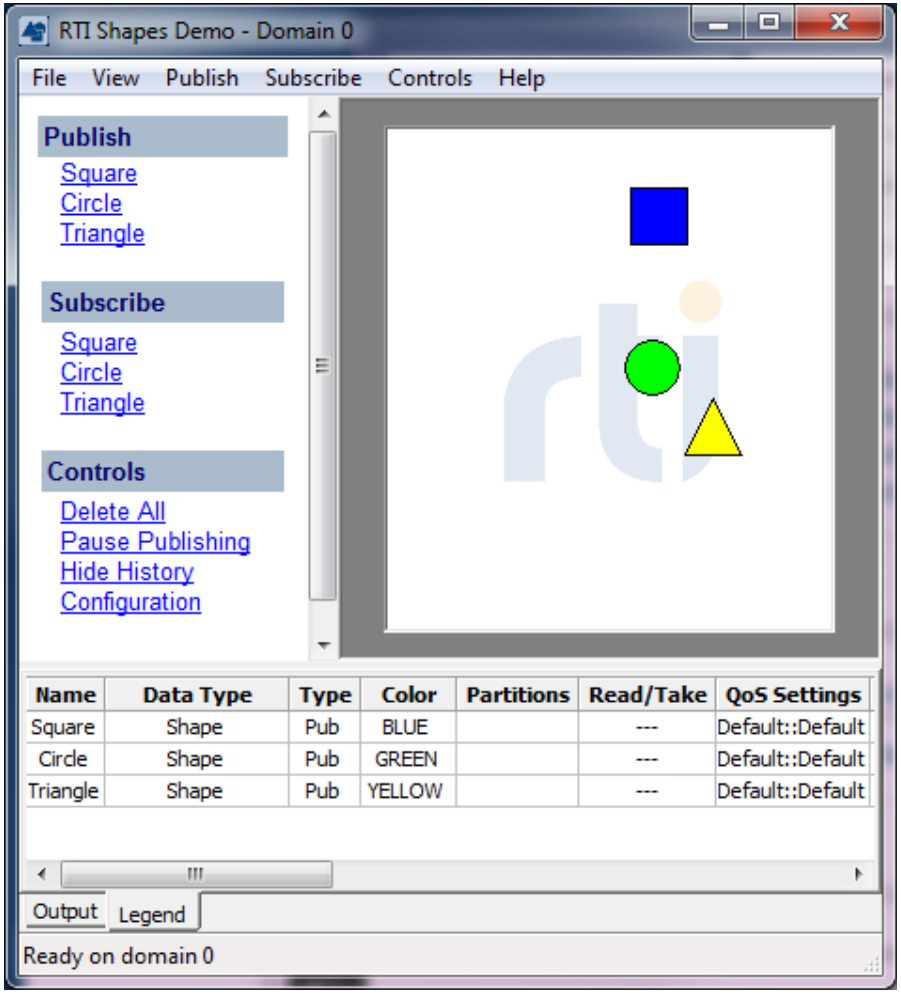
Topics, Instances and Keys

- **Topic**: A set of similar objects, sharing a common Data Type
- **Instance**: A particular object of the set
- **Key**: Fields of the Data Type to identify an object.



Qos Applied by Instance.

Demo




```
const long STR_LEN=24;
struct ShapeType {
  string<MSG_LEN> color; //@key
  long x;
  long y;
  long shapesize;
};
```

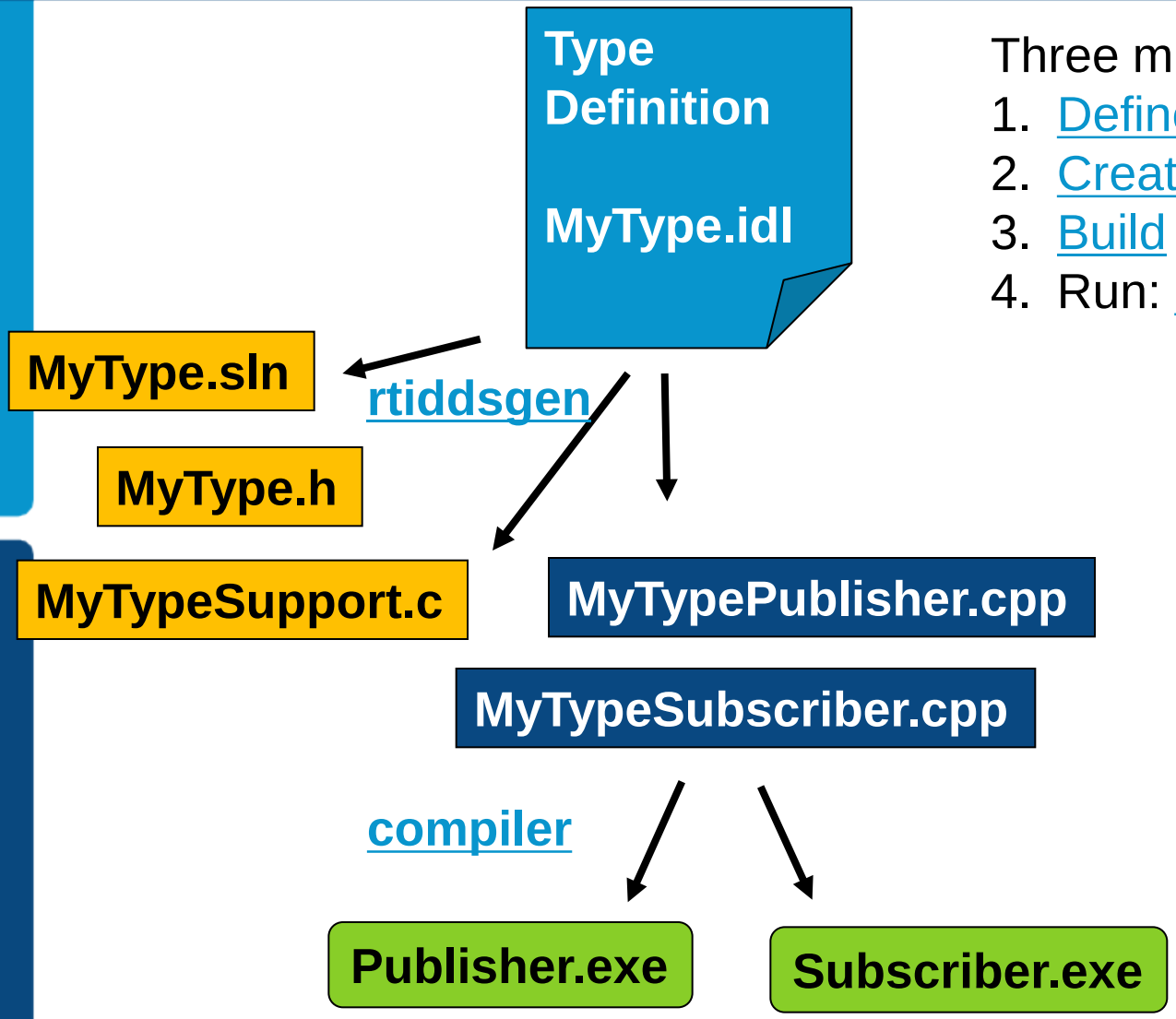
- 3 **Topics**:
  - Square, Circle, Triangle
- Color is the **KEY**

eProsima

The Middleware Experts

Hands On:
A Hello World

Hands-on Example (C++)



Three minutes to a running app!!

1. Define your data
2. Create your project
3. Build
4. Run: publisher subscriber

Aux:

File Browser

Console

Delete Files

rtiddsspy

Example #1 - Hello World

We will use this data-type :

```
const long MSG_LEN=256;
struct HelloMsg {
 string<MSG_LEN> user; //@key
 string<MSG_LEN> msg;
};
```

Side Note: IDL vs. XML

The same data-type can also be described in XML. This is part of the DDS X-Types specification

```
<const name="MSG_LEN" type="long" value="256"/>
<struct name="HelloMsg">
    <member name="user" key="true" type="string"
        stringMaxLength="MSG_LEN"/>
    <member name="msg" type="string"
        stringMaxLength="MSG_LEN" />
</struct>
```

Generate type support (for C++) [Windows]

```
rtiddsgen HelloMsg.idl -language C++ -example x64Win64VS2010\
                                        -replace -ppDisable
```

- Look at the directory you should see:
  - HelloMsg-64-vs2010.sln
  - And Several other files…
- Open the Solution:
  HelloMsgPublisher.cxx
- Compile from visual studio

Generate type support (for C++) [Linux]

rtiddsgen HelloMsg.idl -language C++ -example i86Linux2.6gcc4.4.3\
-replace -ppDisable

- Look at the directory you should see:
  - makefile_hello_i86Linux2.6gcc4.4.3
  - And Several other files…
- Open the source files:
  HelloMsgPublisher.cxx
  HelloMsgSubscriber.cxx
- Compile:
  make –f makefile_hello_i86Linux2.6gcc4.4.3

Generate type support (for Java - Linux)

rtiddsgen HelloMsg.idl -language Java -example i86Linux2.6gcc4.4.3jdk\
-replace -ppDisable

- Look at the directory you should see:
  - makefile_hello_i86Linux2.6gcc4.4.3jdk
  - And Several other files…
    - Look at HelloMsgPublisher.java
    - Look at HelloMsgSubscriber.java
- You can use the makefile to build and the Java programs:

  gmake –f makefile_hello_i86Win32jdk

Execute the program [Windows]

- C++:
  - On one window run:
    - objs\i86Win32VS2005\HelloMsgPublisher.exe
  - On another window run:
    - objs\i86Win32VS2005\HelloMsgSubscriber.exe
- Java
  - On one window run:
    - gmake –f makefile_hello_i86Win32jdk HelloMsgPublisher
  - On another window run:
    - gmake –f makefile_hello_i86Win32jdk HelloMsgSubscriber

- You should see the subscribers getting an empty string…

Execute the program [Linux]

- C++:
  - On one window run:
    - objs/i86Linux2.6gcc4.4.3/HelloMsgPublisher.exe
  - On another window run:
    - objs/i86Linux2.6gcc4.4.3/HelloMsgSubscriber.exe
- Java
  - On one window run:
    - gmake –f makefile_hello_i86Linux2.6gcc4.4.3jdk HelloMsgPublisher
  - On another window run:
    - gmake –f makefile_hello_i86Linux2.6gcc4.4.3jdk HelloMsgSubscriber

- You should see the subscribers getting an empty string…

Writting some data

- Modify HelloMsg_publisher.cxx:

```
/* Main loop */
for (count=0; (sample_count == 0) || (count < sample_count); ++count) {

        printf("Writing HelloMsg, count %d\n", count);

        /* Modify the data to be sent here */
sprintf(instance->user,"%s","eProsima");
sprintf(instance->msg,"Writing HelloMsg, user eProsima, count %d",count);
retcode = HelloMsg_writer->write(*instance, instance_handle);
```

Writting some data (performance tip)

- Modify HelloMsg_publisher.cxx:

```
 /* For a data type that has a key, if the same instance is going to be
       written multiple times, initialize the key here
       and register the keyed instance prior to writing */
sprintf(instance->user,"%s","eProsima");

instance_handle = HelloMsg_writer->register_instance(*instance);


/* Main loop */
for (count=0; (sample_count == 0) || (count < sample_count); ++count) {

        printf("Writing HelloMsg, count %d\n", count);

        /* Modify the data to be sent here */
sprintf(instance->msg,"Writing HelloMsg, user eProsima, count %d",count);
retcode = HelloMsg_writer->write(*instance, instance_handle);
```

Example: Publication

```
// Entities creation
DomainParticipant participant =
    TheParticipantFactory->create_participant(
        domain_id, participant_qos, participant_listener);

Publisher publisher = domain->create_publisher(
        publisher_qos, publisher_listener);

Topic topic = domain->create_topic(
        "MyTopic", "MyType", topic_qos, topic_listener);

DataWriter writer = publisher->create_datawriter(
        topic, writer_qos, writer_listener);

MyTypeDataWriter twriter = MyTypeDataWriter::narrow(writer);

MyType my_instance;
twriter->write(my_instance);
```

Example: Subscription

```
// Entities creation
Subscriber  subscriber = domain->create_subscriber(
        subscriber_qos, subscriber_listener);

Topic topic = domain->create_topic(
        "MyTopic", "MyType",
        topic_qos, topic_listener);

DataReader reader = subscriber->create_datareader(
        topic, reader_qos, reader_listener);

// Use listener-based or wait-based access
```

How to Get Data? (Listener-Based)

```
// Listener code
MyListener::on_data_available( DataReader reader )
{
    MyTypeSeq received_data;
    SampleInfoSeq sample_info;
    MyTypeDataReader treader = TextDataReader::narrow(reader);

    treader->take( &received_data, &sample_info, …)
    // Use received_data
    printf("Got: %s\n", received_data[0]->contents);
}
```

How to Get Data? (WaitSet-Based)

```
// Creation of condition and attachement
Condition  foo_condition =
   treader->create_readcondition(…);
waitset->add_condition(foo_condition);

// Wait
ConditionSeq active_conditions;
waitset->wait(&active_conditions, timeout);

// Wait returns when there is data (or timeout)
MyTypeSeq received_data;
SampleInfoSeq sample_info;

treader->take_w_condition
(&received_data,
       &sample_info,
 foo_condition);

// Use received_data
printf("Got: %s\n", received_data[0]->contents);
```

Listeners, Conditions & WaitSets

Middleware must notify user application of relevant events:

- Arrival of data
- But also:
  - QoS violations
  - Discovery of relevant entities
- These events may be detected asynchronously by the middleware

  … Same issue arises with POSIX signals

DDS allows the application to choice:

- Either to get notified asynchronously using a **Listener**
- Or to wait synchronously using a **WaitSet**

Both approaches are unified using STATUS changes

Status Changes

DDS defines

- A set of enumerated STATUS
- The statuses relevant to each kind of DDS Entity

DDS entities maintain a value for each STATUS

STATUS	Entity
INCONSISTENT_TOPIC	Topic
DATA_ON_READERS	Subscriber
LIVELINESS_CHANGED	DataReader
REQUESTED_DEADLINE_MISSED	DataReader
RUQESTED_INCOMPATIBLE_QOS	DataReader
DATA_AVAILABLE	DataReader
SAMPLE_LOST	DataReader
SUBSCRIPTION_MATCH	DataReader
LIVELINESS_LOST	DataWriter
OFFERED_INCOMPATIBLE_QOS	DataWriter
PUBLICATION_MATCH	DataWriter

```
struct LivelinessChangedStatus
{
    long active_count;
    long inactive_count;
    long active_count_change;
    long inactive_count_change;
}
```

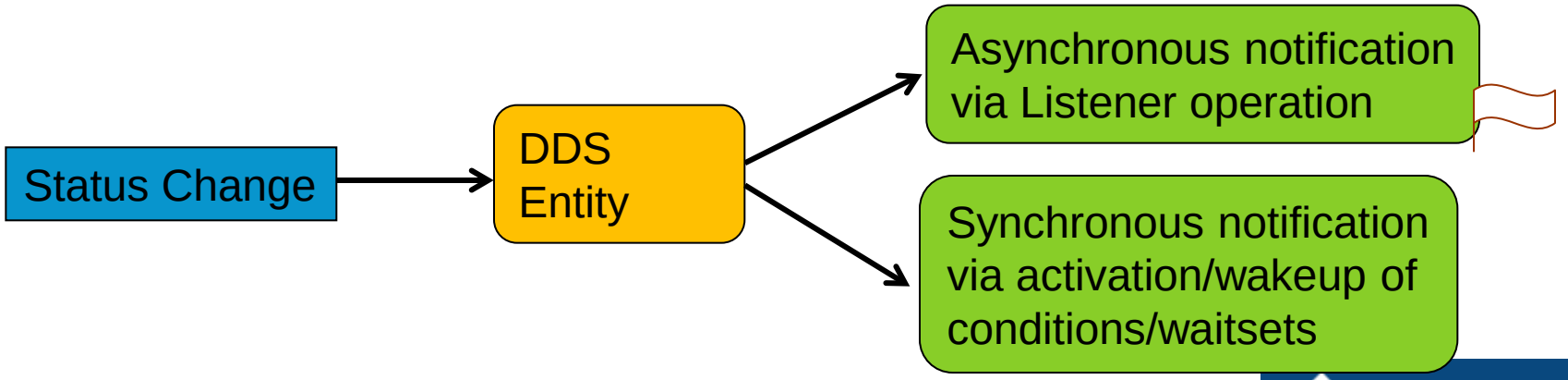
Listeners, Conditions and Statuses

- A DDS Entity is associated with:
  - A listener of the proper kind (if attached)
  - A StatusCondition (if activated)
- The Listener for an Entity has a separate operation for each of the relevant statuses

STATUS	Entity	Listener operation
INCONSISTENT_TOPIC	Topic	on_inconsistent_topic
DATA_ON_READERS	Subscriber	on_data_on_readers
LIVELINESS_CHANGED	DataReader	on_liveliness_changed
REQUESTED_DEADLINE_MISSED	DataReader	on_requested_deadline_missed
RUQESTED_INCOMPATIBLE_QOS	DataReader	on_requested_incompatible_qos
DATA_AVAILABLE	DataReader	on_data_available
SAMPLE_LOST	DataReader	on_sample_lost
SUBSCRIPTION_MATCH	DataReader	on_subscription_match
LIVELINESS_LOST	DataWriter	on_liveliness_lost
OFFERED_INCOMPATIBLE_QOS	DataWriter	on_offered_incompatible_qos
PUBLICATION_MATCH	DataWriter	on_publication_match

Listeners & Condition duality

- A StatusCondition can be selectively activated to respond to any subset of the statuses
- An application can wait changes in sets of StatusConditions using a WaitSet
- Each time the value of a STATUS changes DDS
  - Calls the corresponding Listener operation
  - Wakes up any threads waiting on a related status change



Example #2 - Command-Line Shapes

We will use this data-type :

```
const long STR_LEN=24;
struct ShapeType {
  string<MSG_LEN> color; //@key
  long x;
  long y;
  long shapesize;
};
```

Example #2 - Command-Line Shapes

- Edit the publisher and subscriber
  - Change the TopicName to “Square”

  (or “Circle” or “Triangle”)

- Change the publisher to do something interesting
  - Use colors such as “GREEN” “RED” “YELLOW”
  - Keep the ‘x’ and ‘y’ between 0 and 260
  - Keep the ‘shapesize’ between 0 and 80

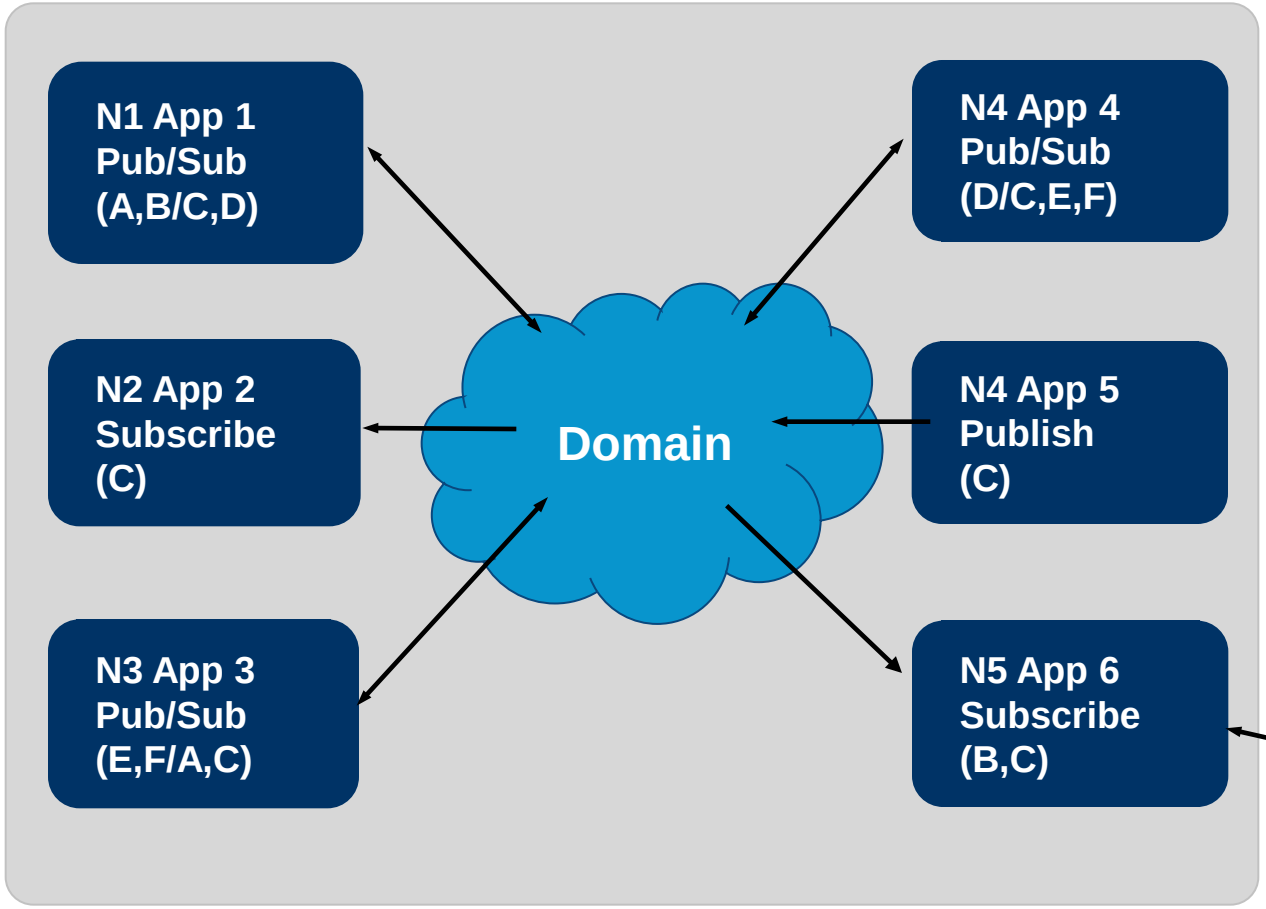
The Middleware Experts

Using DDS:
Common Use Cases

Common use cases

1. Isolating Subsystems
2. Detecting presence of applications
3. Discovering who is publishing/subscribing what
4. Publishing data that outlives its source
5. Keeping a “last-value” cache of objects
6. Monitoring and detecting the health of application elements
7. Building a highly-available system
8. Limiting data-rates
9. Controlling data received by interest set

1. Isolating Subsystems: Domain and Domain Participants



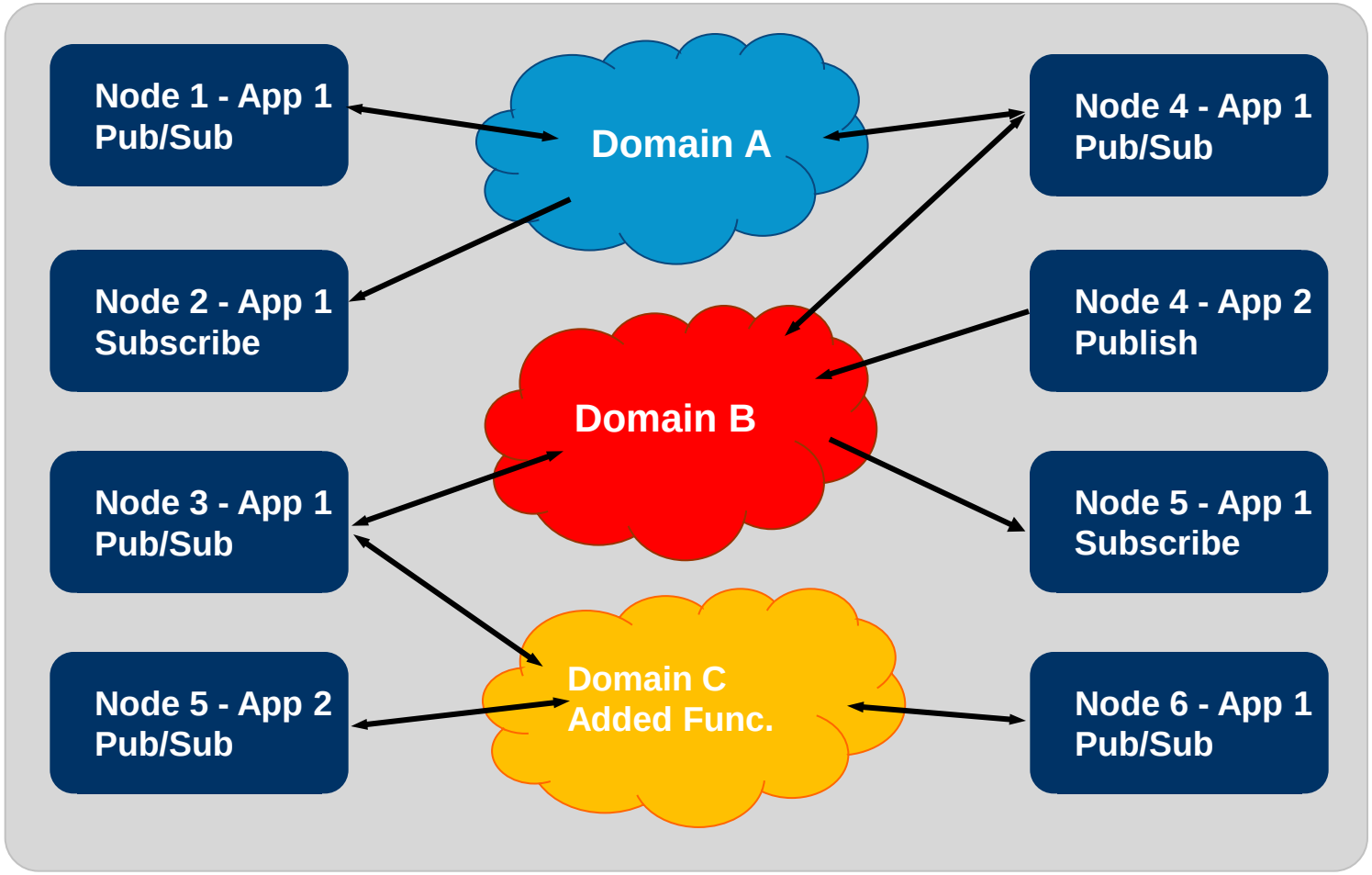
- Container for applications that want to communicate
- Applications can join or leave a domain in any order
- New Applications are “Auto-Discovered”
- An application that has joined a domain is also called a “Domain Participant”

Single ‘Domain’ System

eProsima

1. Isolating Subsystems: Domain and Domain Participants

Using Multiple domains for Scalability, Modularity & Isolation



demo_domain_0 **Multiple Domain System** demo_domain_1

2. Detecting presence of applications DDS builtin Discovery Service

- DDS provides the means for an application to discover the presence of other participants on the Domain
  - The Topic “DCPSParticipants” can be read as a regular Topic to see when DomainParticipants join and leave the network
- Applications can also include meta-data that is sent along by DDS discovery

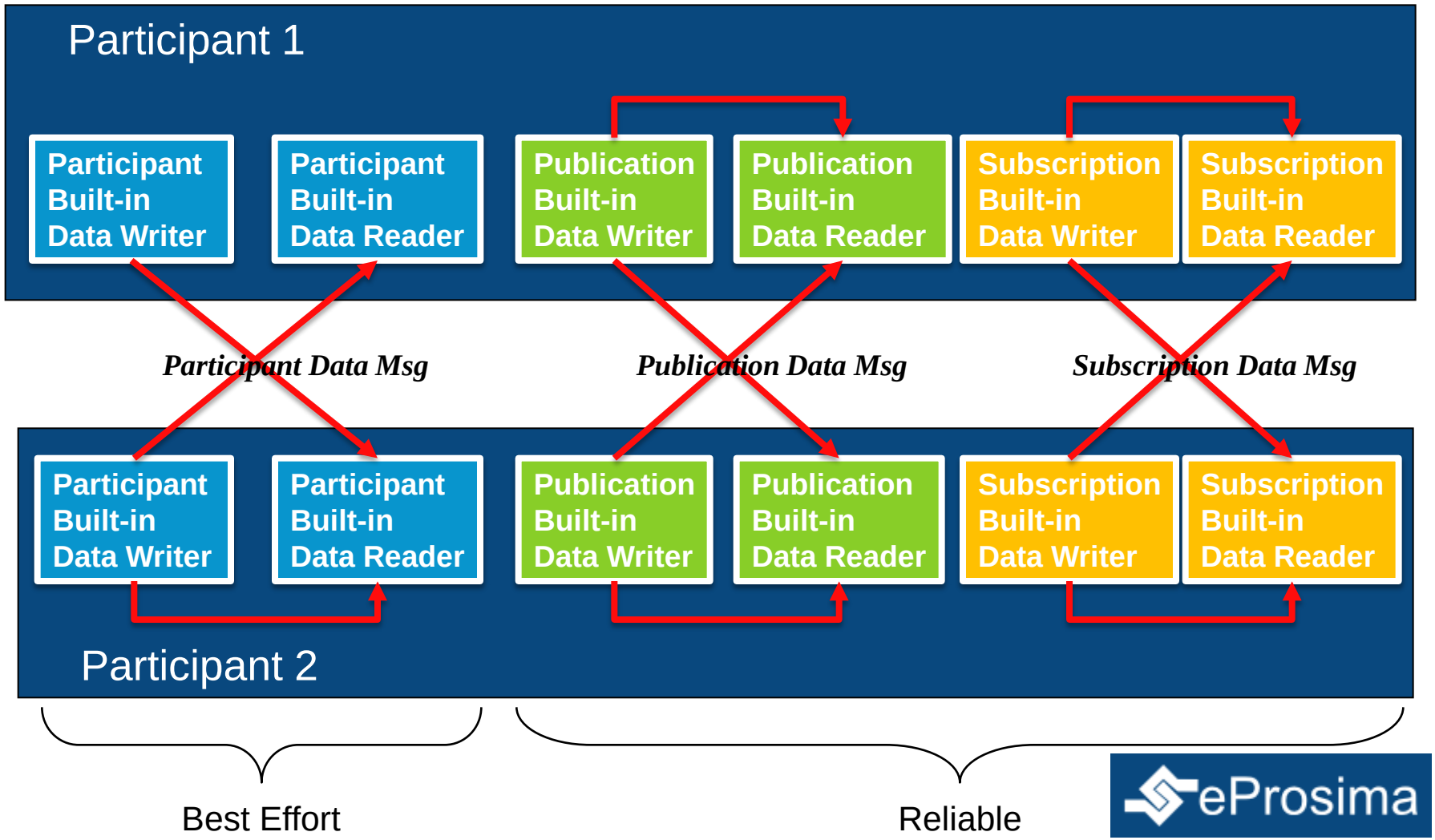
shapes_demo

discovery_in_excel

2. Discovery: How it works

- DDSI Spec Standarized the default mechanism
- Two phase-Discovery
  - 1. Simple Participant Discovery Protocol (SPDP)
    - Best Effort
    - Participant Presence Announcements
  - 2. Simple Endpoint Discovery Protocol (SEDP)
    - Reliable
    - EndPoint information (Publications, Subscriptions, Topics) sent between the different participants

2. Discovery: How it works



3. Discovering who is sublishing/subscribing DDS builtin Discovery Service

- DDS provides the means for an application to discover all the other DDS Entities in the Domain
  - The Topics “DCPSPublications”, “DCPSSubscriptions”, “DCPSTopics”, and “DCPSParticipants” be read to observe the other entities in the domain

shapes_demo **Analizer_**

Example: Accessing discovery information

```
reader = participant
      ->get_builtin_subscriber()
      ->lookup_datareader("DCPSSubscription");

reader_listener = new DiscoveryListener();
reader->set_listener(reader_listener);
```

Example: Displaying discovery information

```
DDS_SubscriptionBuiltinTopicData* subscriptionData =
        DDSSubscriptionBuiltinTopicDataTypeSupport::create_data();
DDS_SampleInfo *info = new DDS_SampleInfo();

…

do {
        retcode = subscriptionReader
                ->take_next_sample( *subscriptionData, *info);

        DDSSubscriptionBuiltinTopicDataTypeSupport
                ::print_data (subscriptionData);

}  while ( retcode != DDS_RETCODE_NO_DATA );
```

shapes_demo

Discovery Example

4. Publishing data that outlives its source: DDS DURABILITY QoS

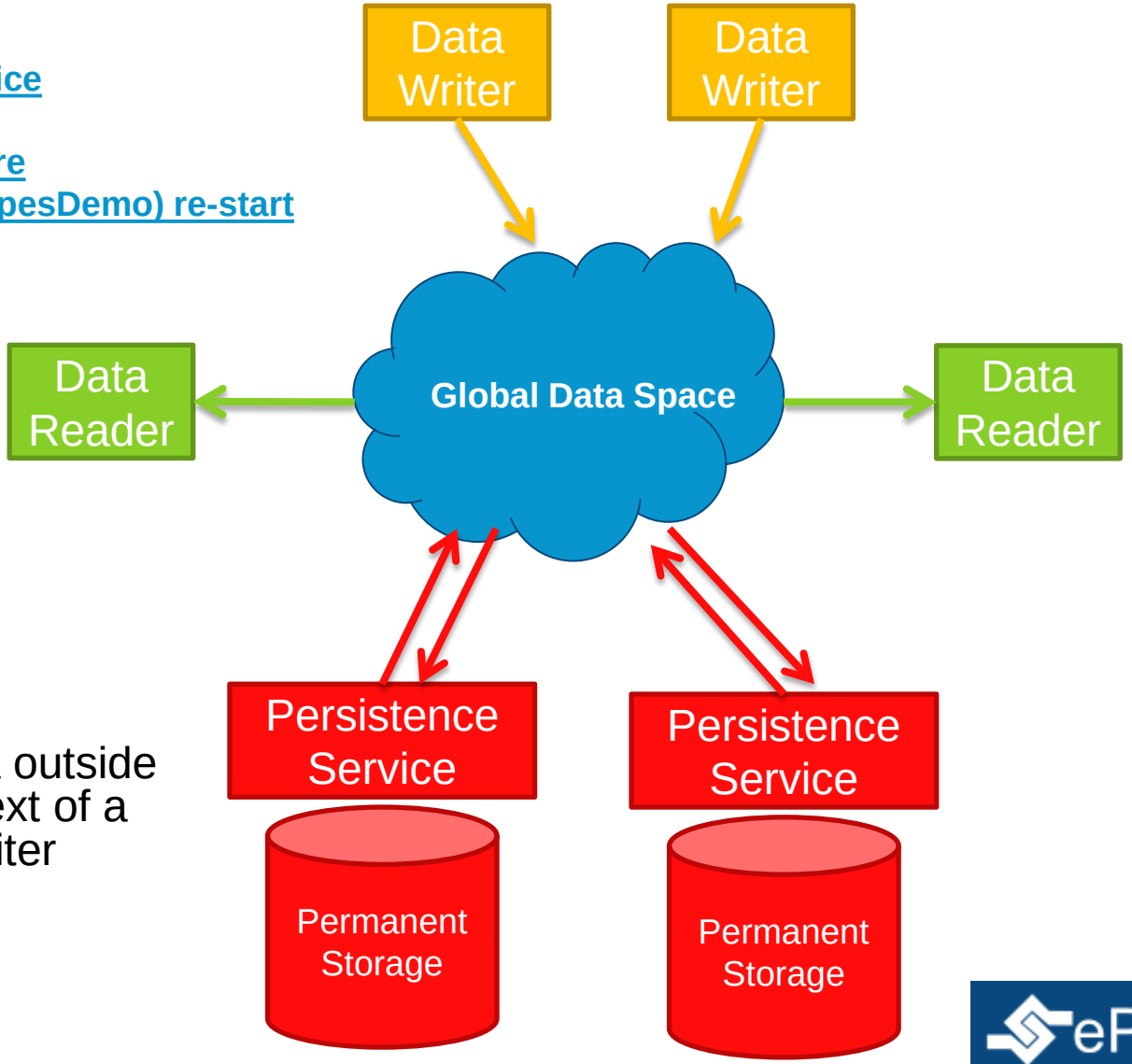
DURABILITY QoS can be set to:

- VOLATILE -- No durability (default)
- TRANSIENT_LOCAL
  - Durability provided by the DataWriter
  - Late joiners will get data as long as writer is still present
- TRANSIENT
  - Durability provided by external “persistence” service
  - Late joiners will get data as long as persistence is still present
- PERSISTENT
  - Durability provided by external “persistence” service
  - Persistence service must store/sync state to permanent storage
  - Persistence service recover state on re-start
  - Late joiners will get data even if persistence service crashes and re-starts

4. Publishing data that outlives its source: Persistence Service

Demo:

1. PersistenceService
2. ShapesDemo
3. Application failure
4. Application (ShapesDemo) re-start



Persists data outside of the context of a DataWriter

Persistence Demo

- Run persistence service:
  - Config: defaultDisk -- from RTI_PERSISTENCE_SERVICE
- Run Shapes demo
  - Persistence Qos profile
  - Publish
  - Kill with process explorer (kill process)
- Run Shapes demo
  - Persistence Qos profile
  - Subscribe

5. Keeping a “Last value” cache

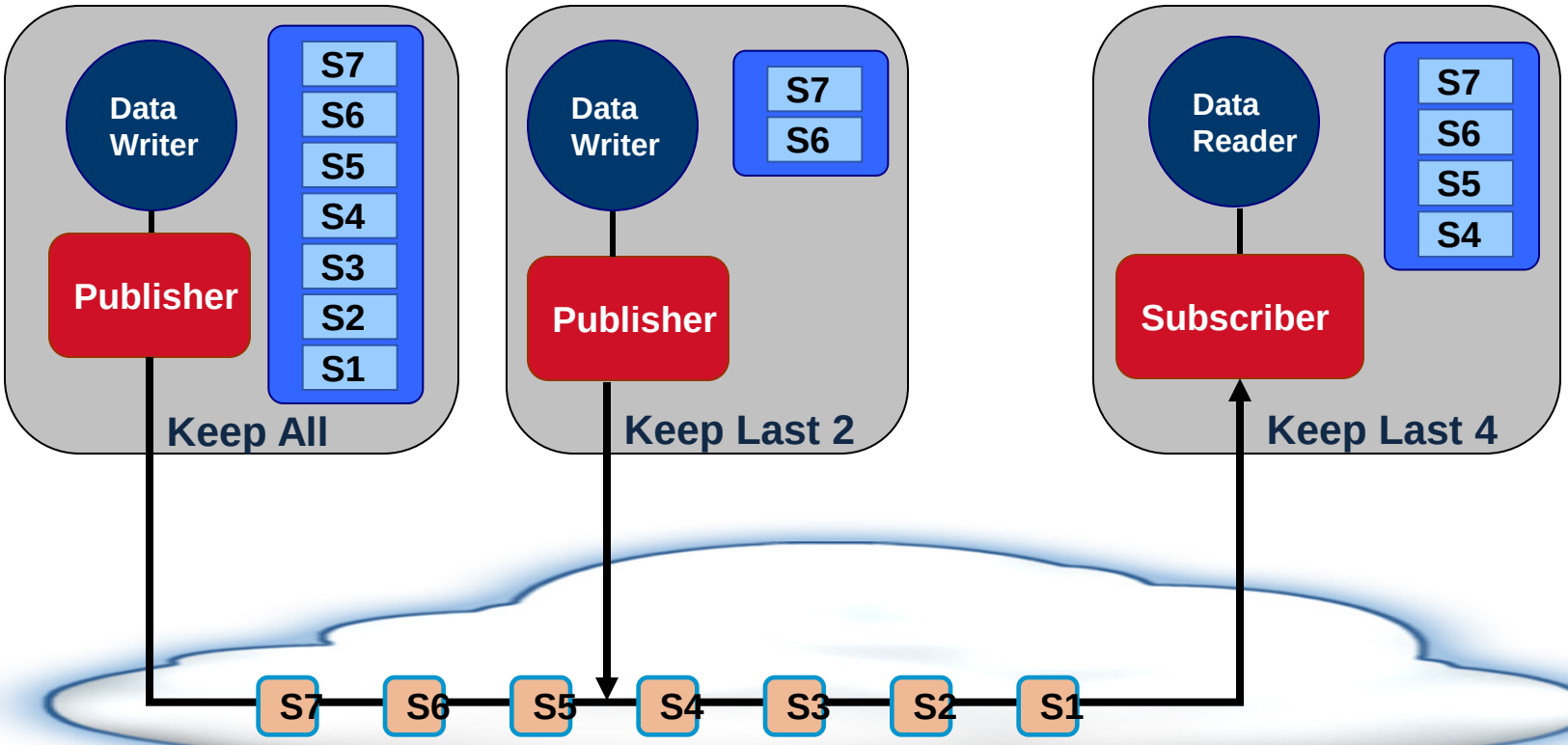
- A last-value cache is already built-in into every Writer in the system
  - Can used in combination with a Durable Writer
- A late joiner will automatically initialize to the last value
- Last value cache can be configure with history depth greater than 1
- The Persistence Service can be used to provide a last value cache for durable data

QoS: History – Last x or All

KEEP_ALL:
Publisher: keep all until delivered
Subscriber: keep each sample until the application processes that instance

KEEP_LAST: "depth" integer for the number of samples to keep at any one time

demo_history



5. Monitoring the health of applications: Liveliness QoS – Classic watchdog/deadman switch

- DDS can monitor the presence, health and activity of DDS Entities (Participant, Reader, Writer)
- Use Liveliness QoS with settings
  - AUTOMATIC
  - MANUAL_BY_PARTICIPANT
  - MANUAL_BY_TOPIC
- This is a request-offered QoS
- Answers the question: “Is no news good news?”

QoS: Liveliness: Type and Duration

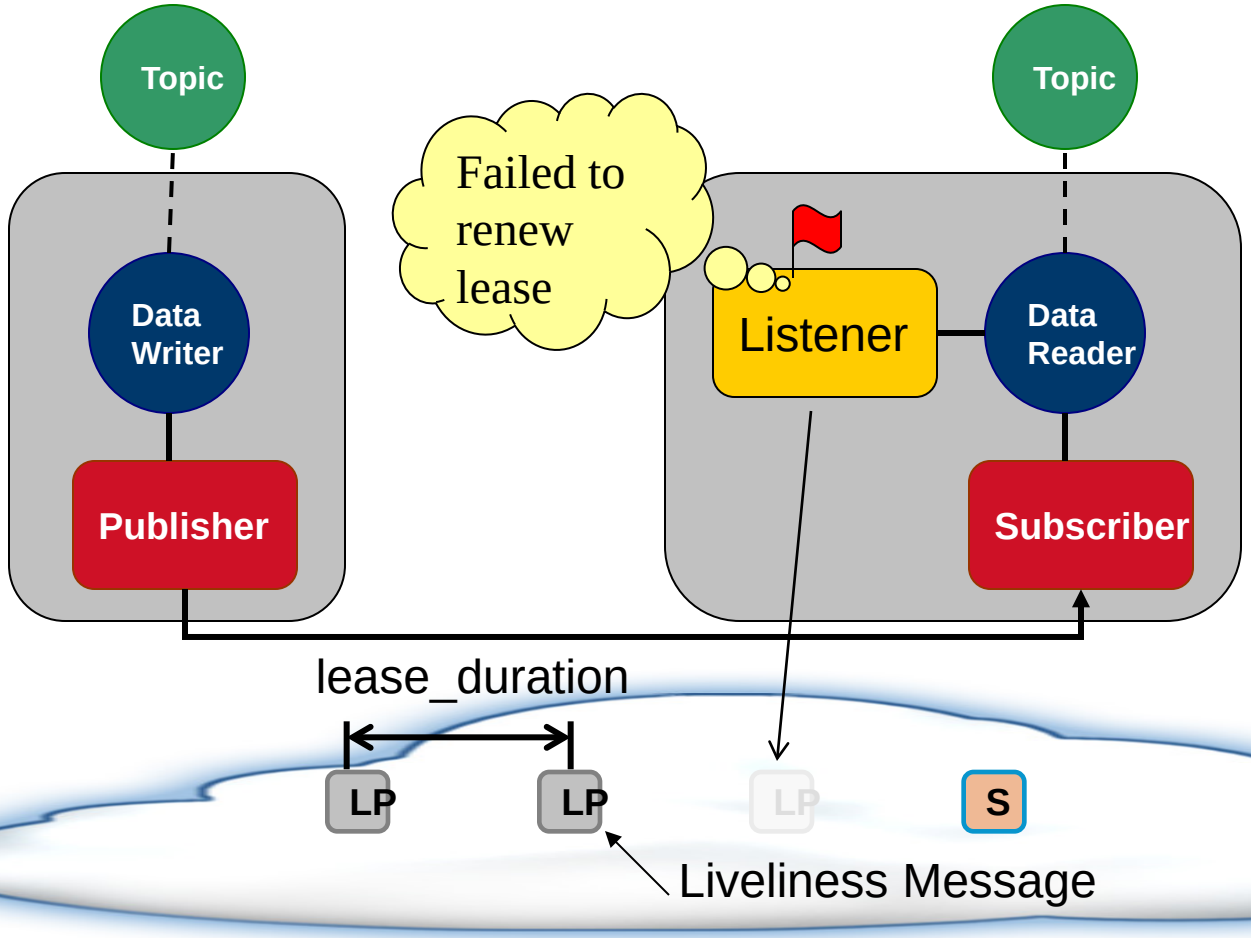
liveliness_example

Type: Controls who is responsible for issues of 'liveliness packets'

AUTOMATIC = Infrastructure Managed

MANUAL = Application Managed

kill_apps

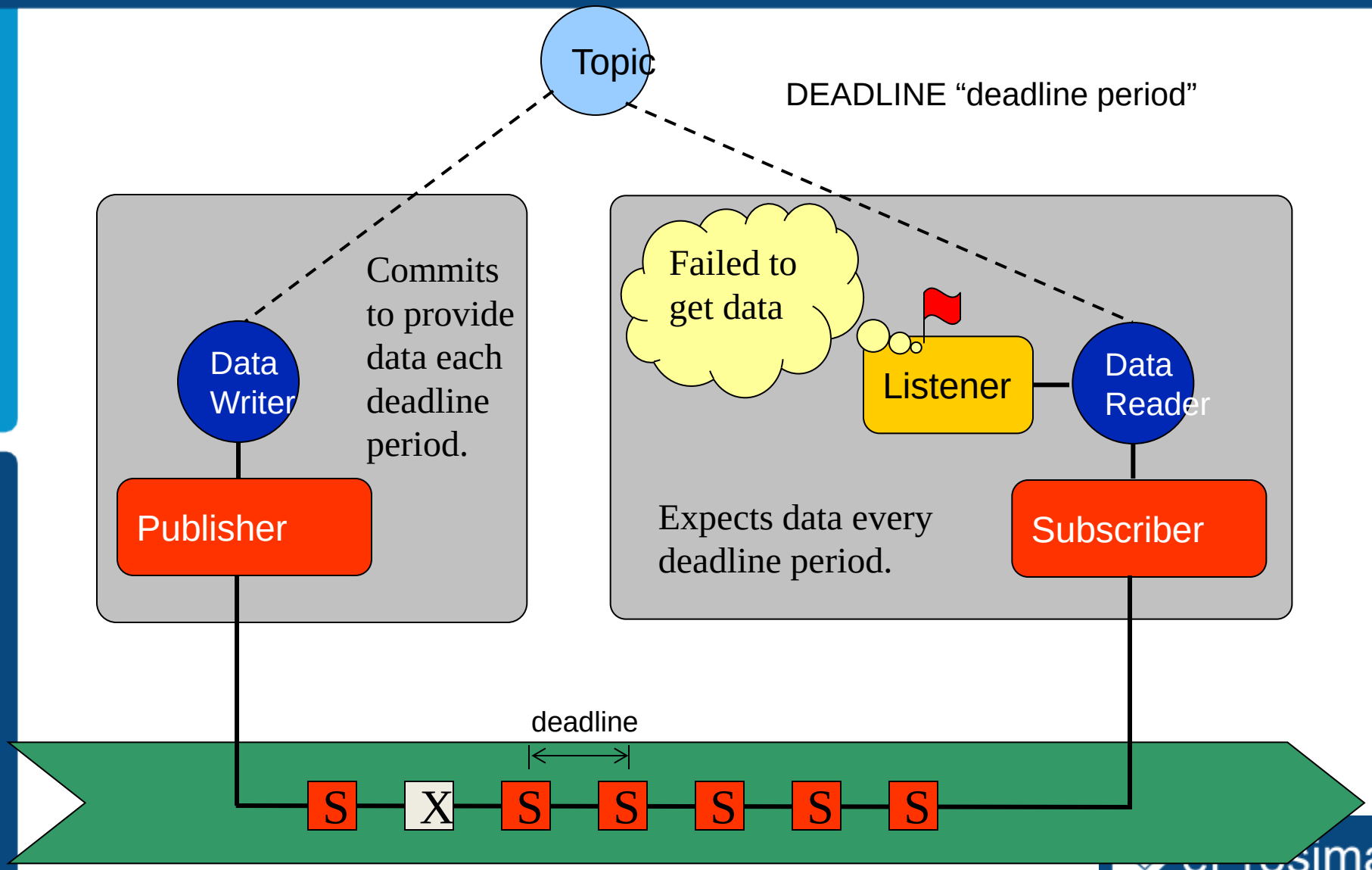


5. Monitoring the health of data-objects: Deadline QoS

- DDS can monitor activity of each individual data-instance in the system
- This is a request-offered QoS
- If an instance is not updated according to the contract the application is notified.
- Failover is automatically tied to this QoS

QoS: Deadline

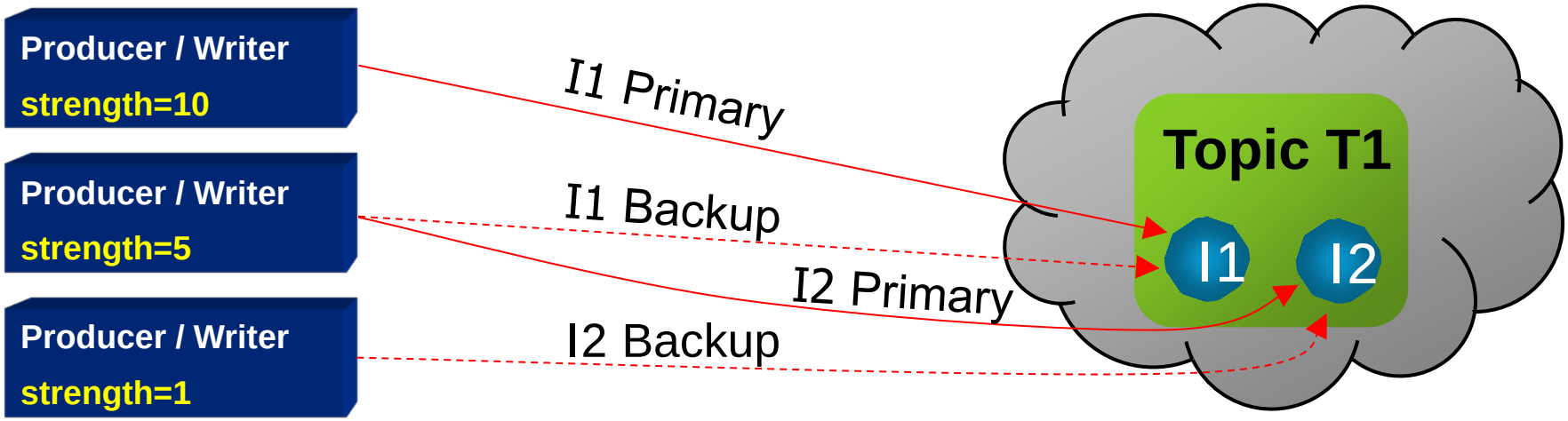
deadline_example



5. Building a highly-available system

- HA systems require combining multiple patters, many directly supported by DDS:
  - Detection of presence -> DDS Discovery
  - Detection of Health and activity -> DDS LIVELINESS
    -> DDS DEADLINE
  - Making data survive application & system failures
    -> DDS DURABILITY
  - Handling redundant data sources and failover
    -> DDS OWNERSHIP

Ownership and High Availability



- Owner determined per subject
- Only extant writer with highest strength can publish a subject (or topic for non-keyed topics)
- Automatic failover when highest strength writer:
  - Loses liveliness
  - Misses a deadline
  - Stops writing the subject

Start demo

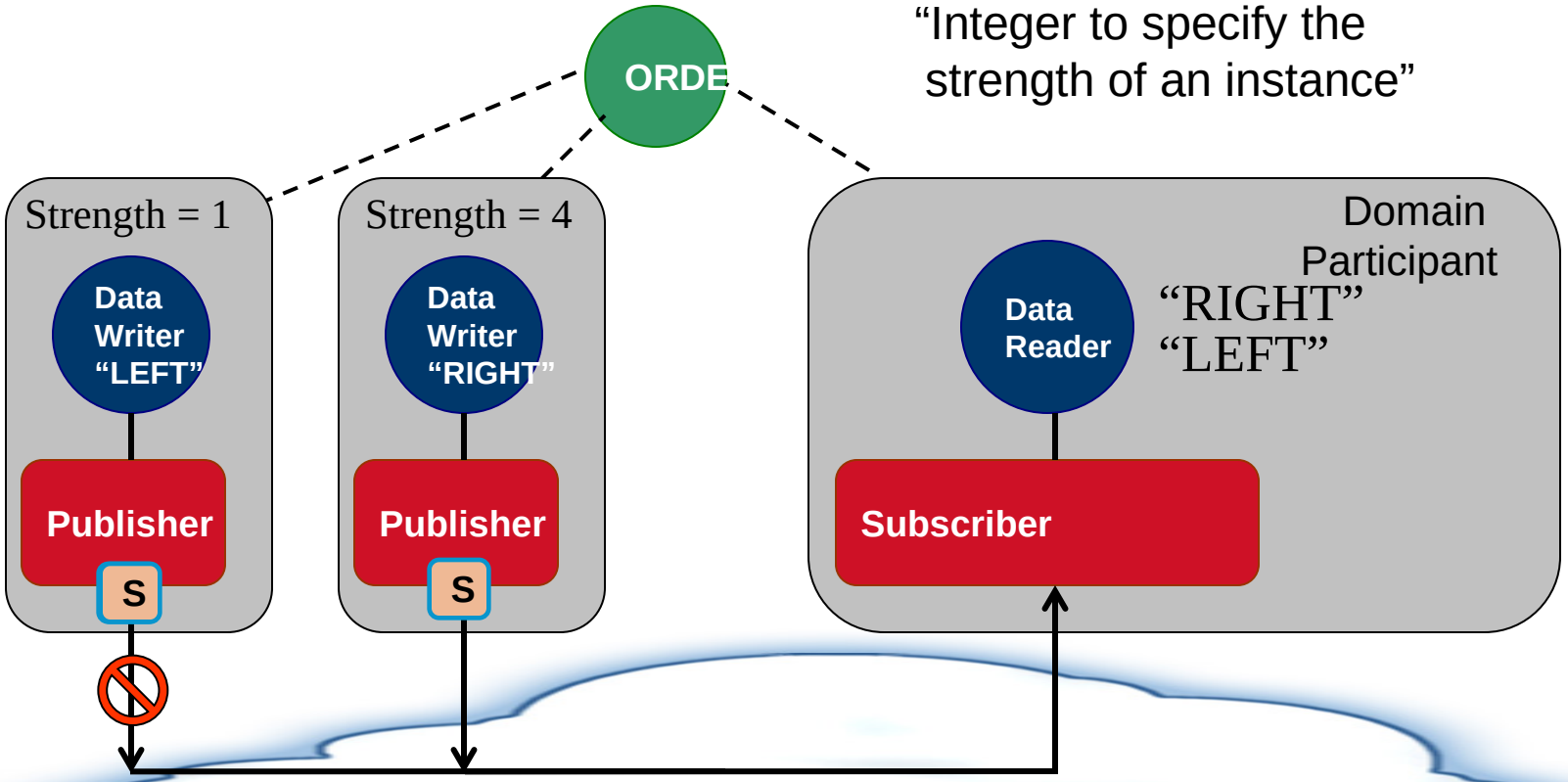
- Shared Ownership allows any writer to update the subject

demo_ownership

QoS: Ownership Strength

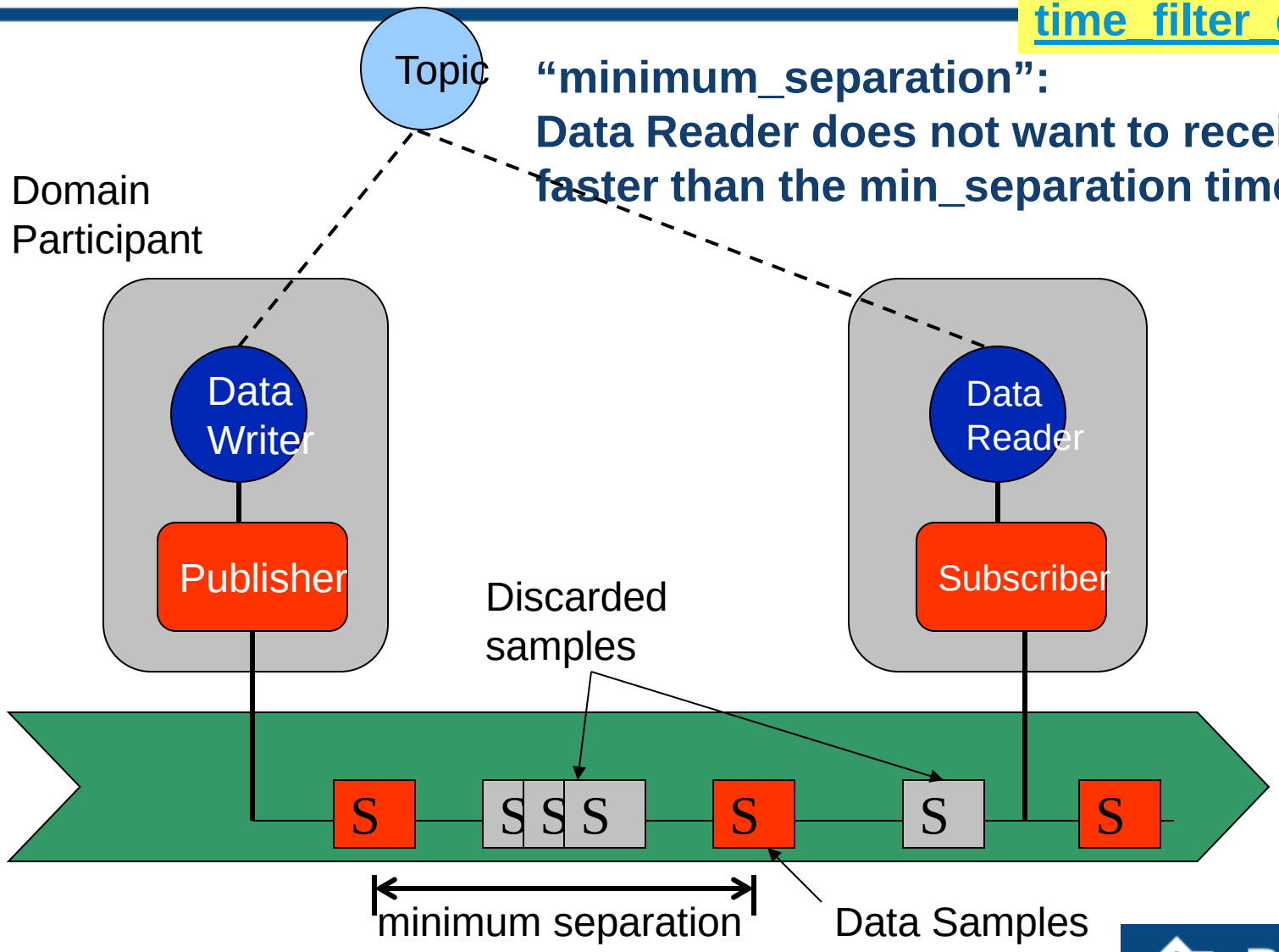
Specifies which DataWriter is allowed to update the values of data-objects

OWNERSHIP_STRENGTH
"Integer to specify the strength of an instance"



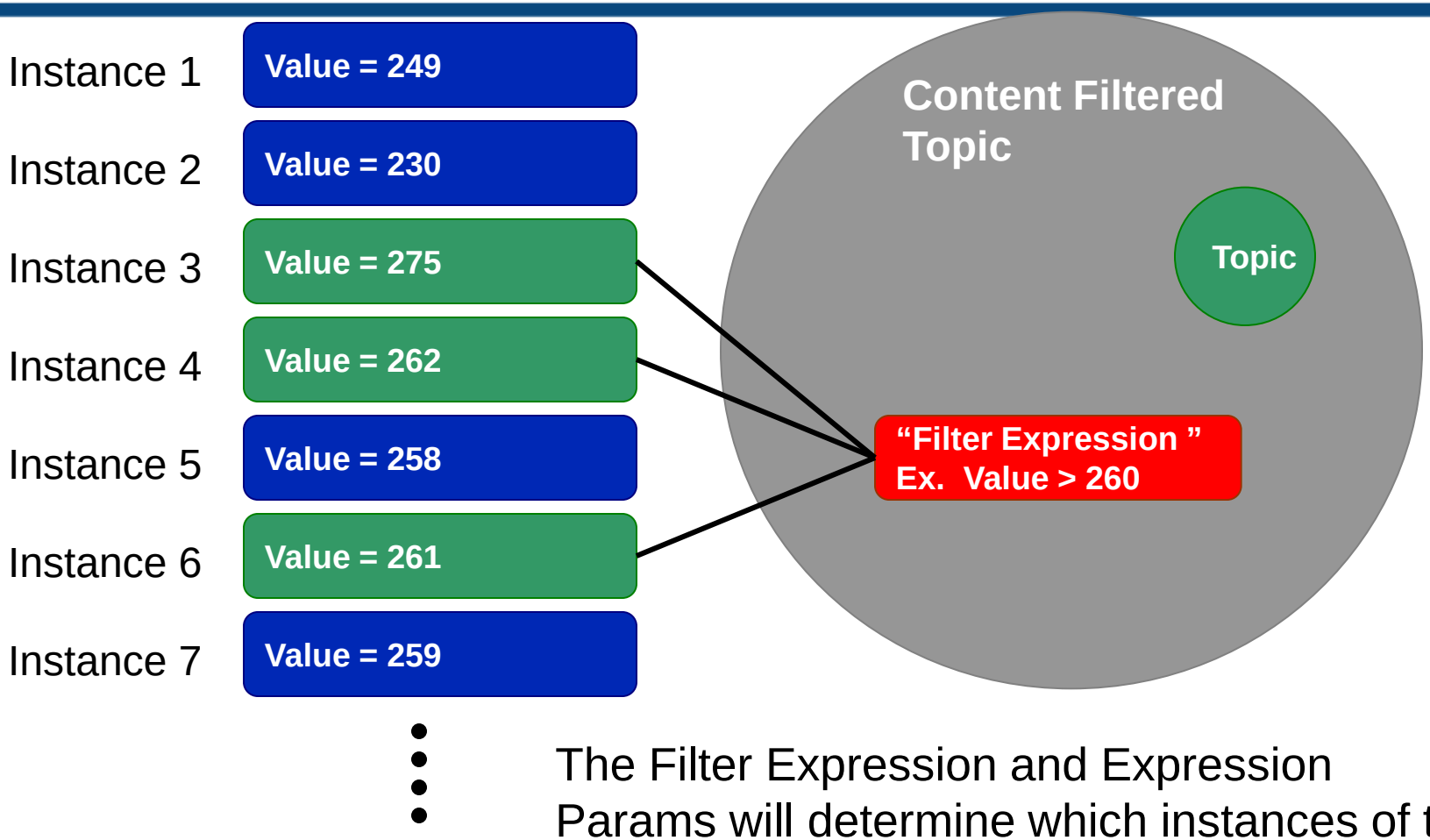
8. Limiting data-rates: QoS: TIME_BASED_FILTER

time_filter_example



eProsima

9. Controlling data received by interest set Content-Based Filtering



The Filter Expression and Expression Params will determine which instances of the Topic will be received by the subscriber.

content_filter_example

The Middleware Experts

Using DDS: Best Practices

Jaime Martin Losa
CTO eProsima
JaimeMartin@eProsima.com
+34 607 91 37 45

www.eProsima.com

Best Practices Summary

1. Start by defining a data model, then map the data-model to DDS domains, data types and Topics.
2. Fully define your DDS Types; do not rely on opaque bytes or other custom encapsulations.
3. Isolate subsystems into DDS Domains.
4. Use keyed Topics. For each data type, indicate the fields that uniquely identify the data object.
5. Large teams should create a targeted application platform with system-wide QoS settings.

See http://www.rti.com/docs/DDS_Best_Practices_WP.pdf

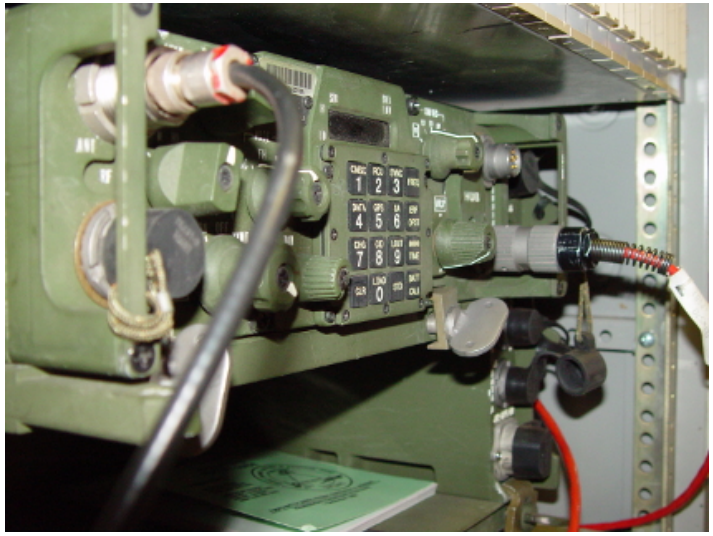
Why defining the proper keys for your data types is important

Many advanced features in DDS depend on the use of keys

- History cache.
- Ensuring regular data-object updates.
- Ownership arbitration and failover management.
- Integration with other data-centric technologies (e.g. relational databases)
- Integration with visualization tools (e.g. Excel)
- Smart management of slow consumers and applications that become temporarily disconnected.
- Achieving consistency among observers of the Global Data Space

Some Success Cases

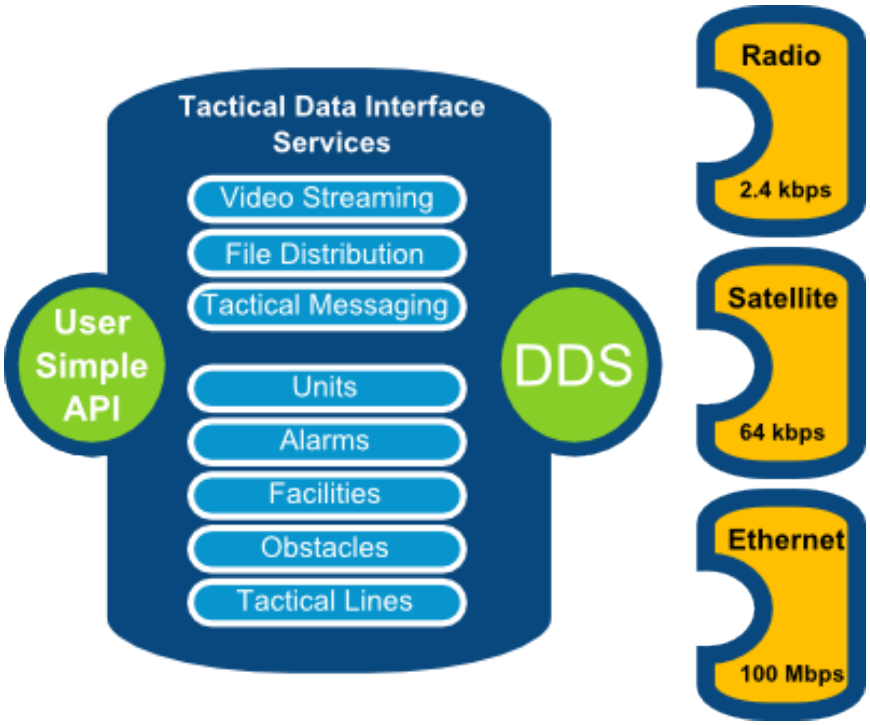
RTI DDS DIL Plugins: Disconnected and Intermitent Links



- eProsima developed the plugins for the Spanish Army Tactical Radios, and later were adquired by RTI.
- Allow the use of DDS in very low bandwidth links, such as **Tactical Radios** and **Satellite**.
  - Tested from 2400 bps

eProsima

Tactical Data Interface: Spanish Army



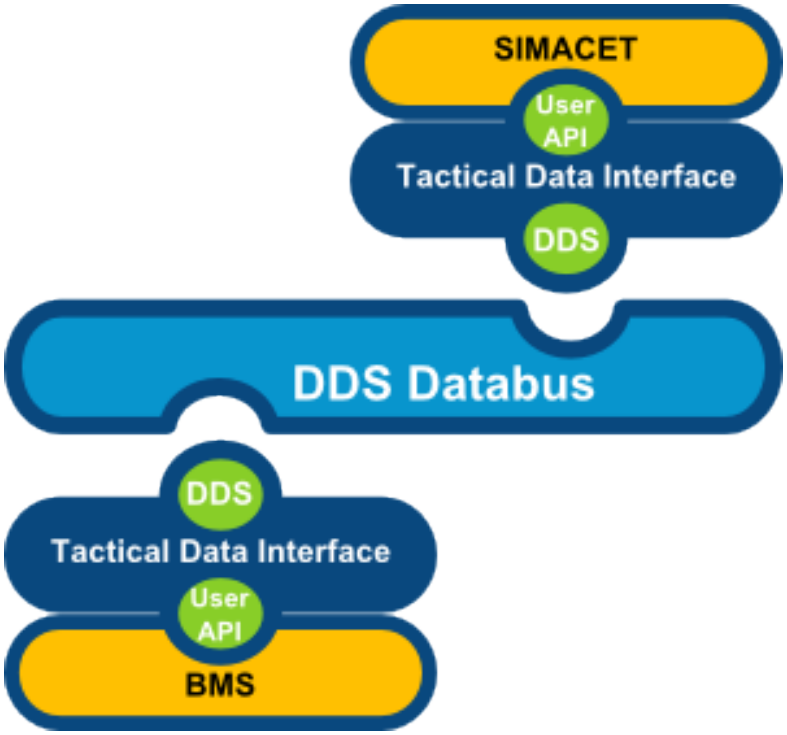
- C2 Interoperability Comm layer:
  - Tactical Radios
    - From 2400bps
  - Satellite

- Mandated for all the Spanish Army C2 systems.
  - Already implemented in the their main C2 systems

eProsima developed the army C2 comm layer using RTI Connext DDS optimized for low bandwidth enviroments. The project included the design of the Data Model and QoS requisites for the Army.

eProsima

C2 Systems: INDRA & Amper



- eProsima Provides a DDS based comm layer for **INDRA and Amper C2 Systems**.

eProsima implemented the mandated Spanish Army Tactical Data Interface for Simacet (Main Spanish Army C2 System, Amper) and BMS (Tanks C2 System, INDRA & Amper)

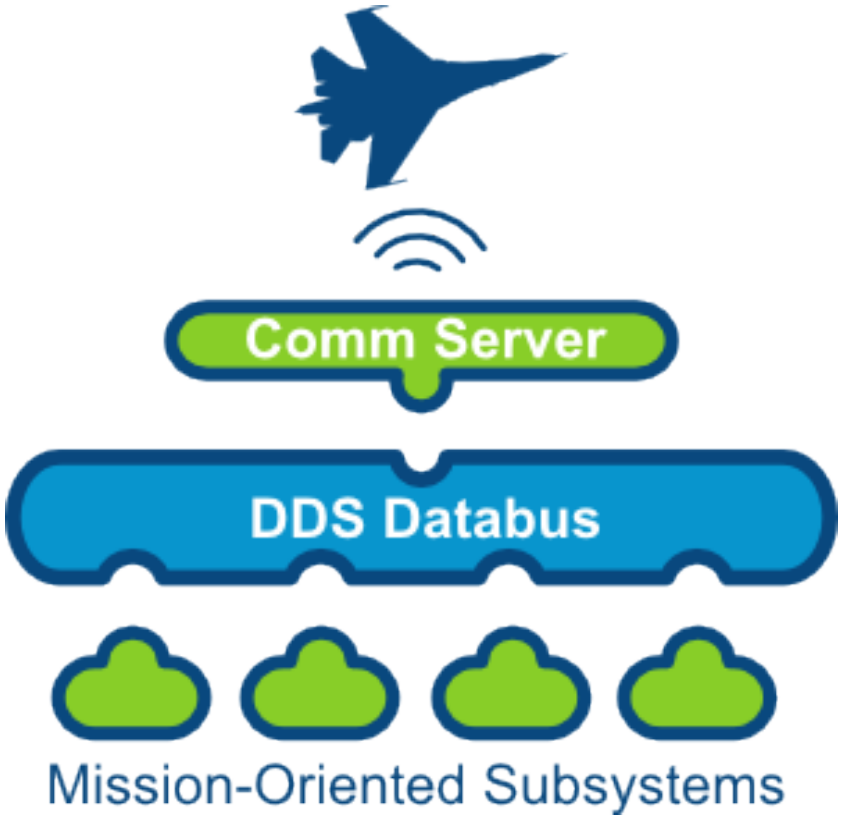
eProsima

SESAR - INDRA ATC

- eProsima provides middleware research and prototyping for ATC Interoperability
- Among the different middleware technologies studied, DDS and WS are the SESAR proposed technologies for ATC interoperability.

eProsima

Cassidian: nEURon and Atlante GS

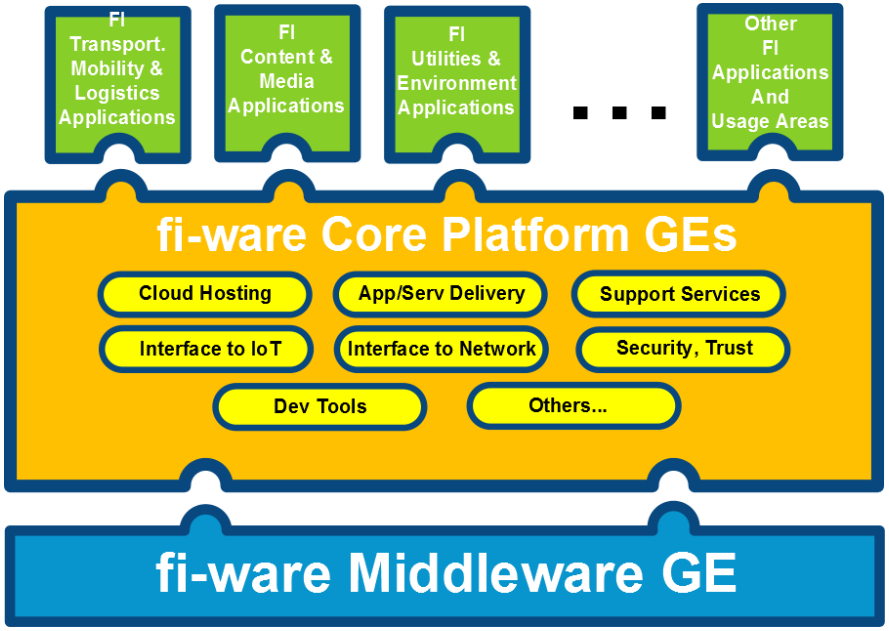


- eProsima provides the comm layer for the ground station comm server.

eProsima Non-Intrusive Recorder is used to record the communications for later analisys.

eProsima

FI-WARE Middleware



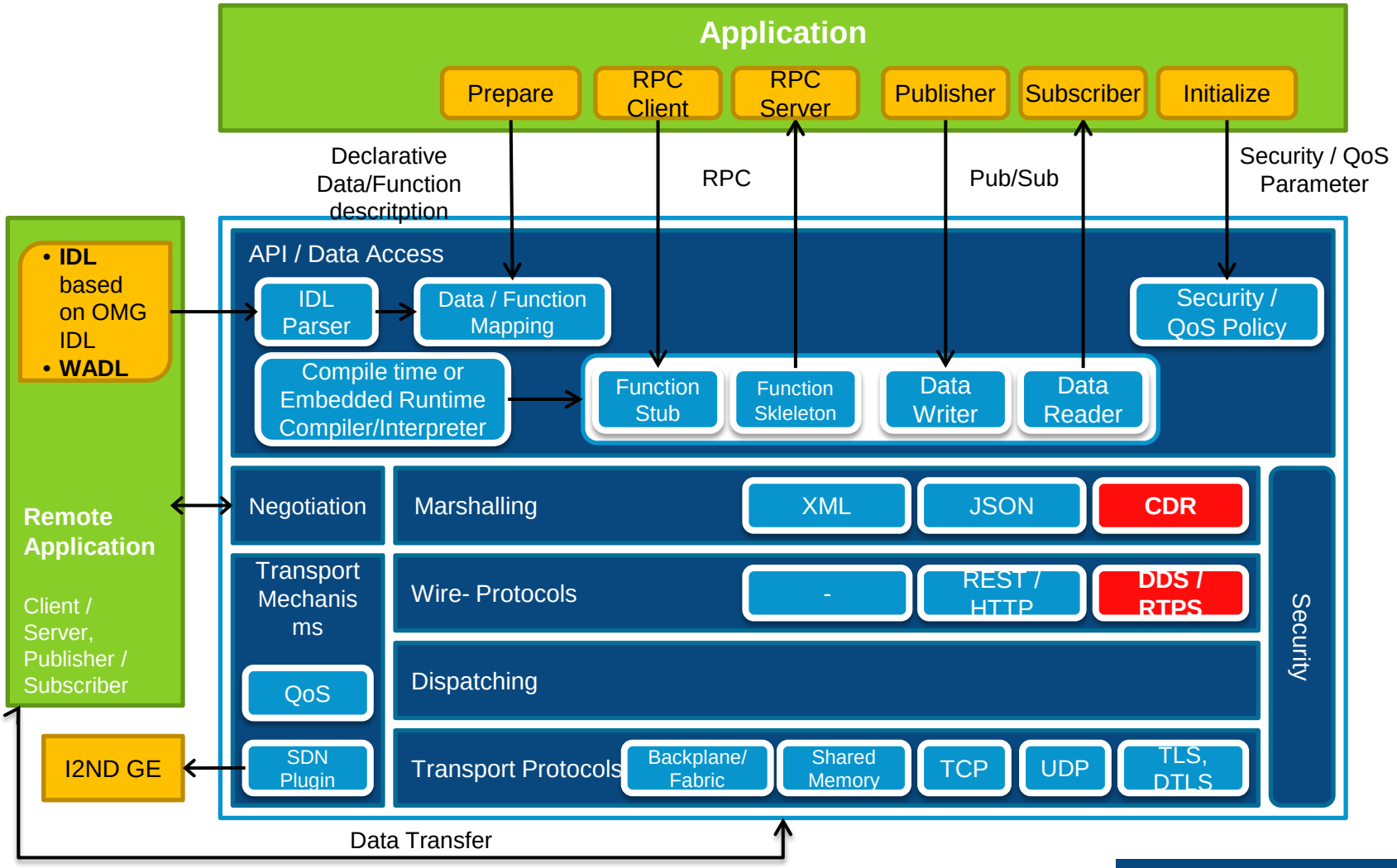
- eProsima has been selected to develop **Future Internet Middleware** in the FI-WARE programme.
- **DDS** will be the core technology

Fi-WARE is a consortium of over more than 30 companies and universities including Telefonica, Siemens, SAP, Atos…

eProsima will partner in this project with the German Universities DKFI and CISPA and the Swiss ZHAW.

eProsima

FI-WARE Middleware: DDS Based



eProsima

The Middleware Experts

Thank you!

Jaime Martin Losa
CTO eProsima
JaimeMartin@eProsima.com
+34 607 91 37 45

www.eProsima.com