The Data Distribution Service for Real-Time Systems

The Real-Time Even-Driven Middleware

OpenSplice DDS

Copyright is held by the author/owner(s).

Angelo Corsaro, Ph.D.
angelo.corsaro@prismtech.com
Product Marketing Manager | PrismTech
OMG RTESS & DDS SIG Co-Chair

Douglas C. Schimidt, Ph.D.
doug.schmidt@prismtech.com
Chief Technology Officer | PrismTech
Table of Contents

- Challenges
- OMG DDS
- DDS Patterns
- Use Cases
- Concluding Remarks
Instructor Biographies

Angelo Corsaro, Ph.D.
Product Strategy & Marketing Manager
PrismTech
Tel: +33 1 69 01 53 54 | Cell: +33 6 43 30 75 65

Dr. Angelo Corsaro is the OpenSplice DDS Product Strategy & Marketing Manager at PrismTech, a market leader in the field of high performance middleware. His responsibilities include product strategy, competitive positioning, technology evolution, & technology evangelism. He is also responsible for strategic standardization at the Object Management Group (OMG), where is co-chair of the Data Distribution Service (DDS) Special Interest Group, & the Real-Time Embedded & Specialized Services Task Force.

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

Douglas C. Schmidt, Ph.D.
Professor of Computer Science at Vanderbilt University

Dr. Schmidt has published over 400 technical papers & 9 books & has given over 400 invited talks & tutorials that cover a range of research topics, including patterns, optimization techniques, & empirical analyses of software frameworks & domain-specific modeling environments that facilitate the development of DRE middleware & applications running over high-speed networks & embedded system interconnects.

In addition to his academic research & government service, Dr. Schmidt has over fifteen years of experience researching & developing ACE, TAO, CIAO, & CoSMIC, which are widely used, open-source middleware frameworks & model-driven development tools that contain a rich set of components & domain-specific languages that implement patterns & product-line architectures for mission-critical enterprise DRE systems.
Objective of the Course

- Familiarize with the problem domains that the OMG Data Distribution Service Standard (DDS) has been designed to address.
- Develop an in depth understanding of the key features of the OMG Data Distributions Service, including:
  - Data-Centricity & Relational Information Modeling
  - Distributed Object/Relational Mapping
  - QoS
- Learn about the Key Patterns recurrently used to build DDS-based High Performance Mission- & Business-Critical Systems.
## Table of Contents

- **Challenges**
  - OMG DDS
  - DDS Patterns
  - Use Cases
  - Concluding Remarks
Defense Systems
The Need for Data Distribution

Real-Timeliness

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 & civil system of systems
- Efficient, scalable & QoS-enabled data dissemination is an enabling technology for Network Centric Systems

The Data Distribution Service, has emerged, & is evolving as the Standard Dissemination Mechanism for Network Centric Systems

The Right Information => To the Right People => At the Right Time
Traffic Growth

Adapted from: “NextGen JPDO Overview”
Airlines Profitability

- Airlines are the industry with the lowest average ROIC (5.9%)
- Operational costs are steadily increasing, e.g., cost of fuel
- Strong competition from low-costs airlines
- Growing Competition from High-Speed Trains for medium distance trips (Europe)

[ROIC = Return On Invested Capital]

Aircraft emissions are a major issue, especially considering traffic growth predictions.

More fuel-efficient aircrafts are needed.

Optimal gate-to-gate route-planning can seriously improve the fuel-efficiency, thus reducing pollution.
Challenges

**Capacity**
- Avoid delays to Travelers
- With traffic growing fast the future air traffic management system needs a three-fold increase in capacity to avoid traffic congestion.

**Efficiency**
- Those who fly often bear the cost of air navigation services.
- Efficiency gains are obtained by improving flight operations & better coordinating airlines, military users, airports & air traffic management bodies.

**Environment**
- Air traffic management professionals reduce this impact by designing fuel-efficient routes, better distributing traffic flows within the available airspace, & optimizing the calculation of departure times.
Challenges

Safety

- When traffic doubles, the risk is squared
- Efforts to maintain safety must be intensified to ensure that the number of incidents does not increase.

Security

- The current security level of the Air Traffic Management (ATM) system has to increase to match threats, protect ATM operations & safeguard citizens’ safe mobility.
Where is the Hurdle?

- Tightly coupled, brittle architectures are pervasive
- It is hard to extend scale, keep up with throughput, add new operational features
- Interoperability is limited and/or requires major human intervention
- Lack of a global vision which prevents optimization & causes fragmented ATC management
Trends & Challenges - Financial Firms

Trends

› Direct connection to exchanges is increasingly chosen over data aggregators to reduce trading latency
› Market data is growing steadily
› Milliseconds speedup over competitors can be worth millions of dollars

Challenges

› Process several hundreds-of-thousands of messages, quickly approaching millions messages, per second
› Minimize latency & jitter, while maintaining the system stability & reliability

Deployed middleware solutions were not designed to address these challenges!
Trends

- More & more firms connecting directly
- Increasing number of sources providing the same quotes
- Competitive advantage obtained by reducing latency on data feeds

Challenges

- Ensuring fairness of data distribution
- Minimizing latency
- Scaling with the number of destinations

Deployed middleware solutions were not designed to address these challenges!
**OPRA Traffic Predictions**

**Trends**
- One order of magnitude growth on market data from 2005 to 2008
- OPRA message rates predicted to reach ~2M msgs/sec by early 2009
- Traffic bursts predicted to further increase by 20%

**Challenges**
- Can your messaging infrastructure keep-up with this trend?
### Predictions & Adjustments

#### Trends
- Market data adjustments of 480K msgs/sec for July 2008
- July 2008 adjustment is higher than the market data rates of Jan 2007

#### Challenges
- Traffic peaks might be might higher than you were used to, can your infrastructure cope with it?
Trading with Pareto

- Usually 20% of the trades provides 80% of the gains
- Under overload condition how can you ensure you keep trading at least 20% of your target?
- How can you ensure that you keep trading the **right 20%** of data?

Vilfredo Pareto, 1848-1923

![Graph showing System Message Rate and Incoming Message Rate](image-url)
Power Grid

Trends/Challenges

- Deregulation in Power Production changed the operational landscape
- Communication is fundamental to control generation, transmission, & distribution
- Currently deployed monitoring & control systems are not capable of providing the right level of QoS
- Secure communication
August 14, 2003 Blackout in US & Canada

NOAA Satellite Perspective...
Utilities

Trends/Challenges

- Growing systems size, larger scale, higher data volumes
- Increase need to have real-time situation awareness
- QoS
- Control over communication resources
- Necessity to easily integrate with Enterprise
- Security
SCADA / DCS

**Trends/Challenges**

- Increasingly moving toward COTS IT technologies for networking, e.g., ethernet, industrial ethernet
- Leveraging COTS IT communication middleware
- Maintaining Real-Time behavior
- Smoothly Integrating with the higher functional layer
Resulting Technical Challenges

Power is nothing without control!

We need a Publish/Subscribe Middleware that provides...

Most Hyped

☑ Low latency & predictable data dissemination
☑ Support for several 100K messages/sec, up to Millions messages/sec
Resulting Technical Challenges

We need a Publish/Subscribe Middleware that provides...

Most Hyped
- Low latency & predictable data dissemination
- Support for several 100K messages/sec, up to Millions messages/sec

Not so Hyped, yet Fundamentally Important
- Rich QoS Support
- Stability under overload condition
- Scalability
- Control over latency/throughput tradeoffs
- Traffic Engineering
  - HW Filtering
  - Traffic Shaping
  - Priority driven delivery
- High Performance Persistence
- Fairness
- Event Processing
- Productivity

*Power is nothing without control!*
Table of Contents

- Challenges
- Overview of DDS
  - DDS Patterns
  - Use Cases
  - Concluding Remarks
Defining Publish/Subscribe
What is Publish/Subscribe?

- **Coordination Model** based on the idea of an *asynchronous ensemble* which communicates by writing & reading (publishing & subscribing) information in a “Global Information Space”
- The coordination model is intended to promote loose coupling
Client/Server vs. Publish/Subscribe

The “4Ws” of Client/Server

- **Who+Where**: Space Coupling
- **What**: Structural Coupling
- **When**: Time Coupling

Examples: CORBA, COM+, Java RMI, .Net Remoting, Web SVCs

A Single “W” for Publish/Subscribe

- **What**: Structural Coupling

Examples: OpenSplice DDS, TIBCO Rendezvous, JMS
Client Server vs. Publish Subscribe

Client/Server
- Complex Deployment
- Tight Coupling
- Fragile to Fault
- Inherently One-to-One

Pub/Sub
- Plug & Play
- Loosely Coupled
- Fault Resilient
- Inherently Many-to-Many
Different realizations of the publish/subscribe model provide different support with respect to:

- **Subscription Model.** How is the “What” specified? i.e., how information to be written/read (published/subscribed) is specified
- **Degree of Time Decoupling.** To which extent “When” does not matter
- **Support for QoS**
Data-Centric Publish Subscribe

- Relies on a Relational Information Model to specify the information which belongs to the Global Data Space
- Subscriptions can be specified by means of:
  - Topic & their associated Types, as well as
  - Content, often by means of SQL expressions
OpenSplice DDS

The future of Publish/Subscribe has come!

A High Performance Real-Time Data-Centric Publish/Subscribe Middleware

- The right data, at the right place, at the right time
- Fully distributed, multicast-enabled, high performance, highly scalable, & high availability, hot-swap/hot-hot architecture

Perfect Blend of Data-Centric & Real-Time Publish/Subscribe Technologies

- Object/Relational Mapping, Content-based subscriptions, queries, filters, & windows
- Fine-grained tuning of resource usage & data delivery & availability QoS
- Optimal networking & computing resources usage

Loosely coupled

- Plug & Play Architecture with Dynamic Discovery
- Time & Space Decoupling

Open Standard

- API Standard (DDS v1.2)
- Wire Protocol Standard (DDSI v2.1)
The OMG Data Distribution Service

Data Distribution Service for Real-Time Systems

- Leap forward in Topic-based Publish/Subscribe Middleware state of the art
- Carefully specified to allow very high performance, scalable, predictable & high-availability implementations
- Language Independent, OS & HW architecture independent

Fully Standardized Solution

- Data Distribution Service (DDS) for Real-Time Systems v1.2
- RTPS, DDS Interoperability Wire Protocol

The OMG Data Distribution Service satisfies the most challenging information dissemination requirements across a wide set of application domains, ranging from multi-board systems to system-of-systems.
DDS: Foundational Abstractions

- **Information Model.** Defines the structure, relations, & QoS, of the information exchanged by the applications, & supports Flat, Relational, & Object Oriented Modeling

- **Typed Global Data Space.** A logical data space in which applications read & write data **anonymously** & **asynchronously**, decoupled in space & time

- **Publisher/Subscriber.** Produce/Consume information into/from the Global Data Space

- **QoS.** Regulates the non-functional properties of information in the Global Data Space, e.g., reliability, availability, & timeliness, etc.
Topics & Data-Centric Pub/Sub

- **Topics.** Represent the information exchanged between Publisher & Subscribers. Topics are an association between a **data type** & a set of **QoS** & identified by a **unique name**

- **Data Types.** The data type associated to a Topic must be a structured type expressed in OMG IDL

- **Topic Instances.** Key values in a datatype uniquely identify a Topic Instance (like rows in table)

- **Content Awareness.** SQL Expressions can be used to do content-aware subscriptions, queries, joins, & correlate topic instances

---

```
struct TempSensor {
    int tID;
    float temp;
    float humidity;
};
#pragma keylist TempSensor tID
```

---

<table>
<thead>
<tr>
<th>TempSensor</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>t2</td>
</tr>
<tr>
<td>tID</td>
<td>temp</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>21.5</td>
</tr>
</tbody>
</table>

---

---
Relational Information Modeling

- Topic Keys can be used to identify instances as well as relationships
- Relationships can be navigated by relying on a subset of SQL 92
- One-to-many relationships can be captured using foreign keys
- Many-to-many relationships need to be modeled using topics
- Keys can be represented by an arbitrary number of Topic fields
Data Centric Publish/Subscribe

```c
struct TempSensor {
    int tID;
    float temp;
    float humidity;
};
#pragma keylist TempSensor tID
```

### Publishers

<table>
<thead>
<tr>
<th>tID</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td>60</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>71</td>
</tr>
</tbody>
</table>

### Subscribers

```sql
SELECT * FROM TempSensor t
WHERE s.temp > 20
```

<table>
<thead>
<tr>
<th>tID</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td>60</td>
</tr>
</tbody>
</table>

**Fully Distributed Global Data Space**
Data Centric Publish/Subscribe

Publishers

TempSensor

<table>
<thead>
<tr>
<th>tID</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>62</td>
</tr>
</tbody>
</table>

TempSensor

<table>
<thead>
<tr>
<th>tID</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>18</td>
<td>75</td>
</tr>
</tbody>
</table>

TempSensor

<table>
<thead>
<tr>
<th>tID</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>21</td>
<td>71</td>
</tr>
</tbody>
</table>

Subscribers

TempSensor

<table>
<thead>
<tr>
<th>tID</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>62</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>21</td>
<td>71</td>
</tr>
</tbody>
</table>

struct TempSensor {
    int tID;
    float temp;
    float humidity;
};

#pragma keylist TempSensor tID

SELECT * FROM TempSensor t
WHERE s.temp > 20

s.tID == 1
Data Centric Publish/Subscribe

Publishers

```
struct TempSensor {
  int tID;
  float temp;
  float humidity;
};
#pragma keylist TempSensor tID
```

Subscribers

```
SELECT * FROM TempSensor t
WHERE s.temp > 20
```

Fully Distributed Global Data Space
Processing Content & Structure

- Content-based Subscriptions (Continuous Queries)
- Topics Joins & Projections
- Queries
- Correlation across topics (via joins)
- Event windows (via History QoS)

OpenSplice DDS some of the most useful features found in Complex Event Processing platforms!

---

<table>
<thead>
<tr>
<th>tID</th>
<th>temp</th>
<th>humidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>21.5</td>
<td>72.3</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>78</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>62</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
S_1 & \{\text{tID} = 1\} \\
S_2 & \{\text{temp} > 21 \land \text{humidity} > 75\} \\
S_3 & \{\text{temp} < 16\}
\end{align*}
\]
Processing Content & Structure

- Content-based Subscriptions (Continuous Queries)
- Topics Joins & Projections
- Queries
- Correlation across topics (via joins)
- Event windows (via History QoS)

OpenSplice DDS some of the most useful features found in Complex Event Processing platforms!

<table>
<thead>
<tr>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>{iID = 1}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tID</td>
<td>temp</td>
<td>humidity</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>62</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>78</td>
</tr>
<tr>
<td>3</td>
<td>21.5</td>
<td>72.3</td>
</tr>
</tbody>
</table>

\{ temp > 21 | humidity > 75\}
OpenSplice DDS supports **true Object Oriented Distributed Information Modeling** providing:

- **Reduced Complexity & Improved Productivity**
  - Focus on the architecture & business logic, while hiding away the details involved with the diffusion of shared objects state

- **Encapsulation**
  - Attributes are only accessible through dedicated getter/setter operations, i.e., don’t need to the messaging middleware or the application to have privileged access to business objects representation

- **Local Operations**
  - Besides getters/setters, all other kind of manipulations can be done using custom operations

- **Inheritance**
  - Single inheritance supported for DLRL Objects

- **Navigable Relationships**
  - Single Relationships
  - Multi Relationships (Set, Map, List)
**Object/Relational Mapping**

Unleashing the power of Objects...

**OO ➔ Relational**

- Middleware can automatically manage the generation & association between the Object-Oriented Model & the Relational Model

**Relational ➔ OO**

- The Relational Model can be mapped to an Object Oriented model
- The mapping is under control of the architect
Multiple Mappings

\(\text{‣ Different Object/Relational Mappings (ORM) can be used for different subsystems} \)

\(\text{‣ The state is shared across all the local reconstruction (ORMs)} \)
An Example

High-End 3D Visualization

Primary Radar

Flight Data Processor

OpenSplice|DDS

OOPSLA Tutorial

Angelo CORSARO
An Example

High-End 3D Visualization

Primary Radar

Flight Data Processor
An Example

High-End 3D Visualization

Primary Radar

Secondary Radar

Flight Data Processor

Automatically Manage Relationships

OpenSplice DDS

OOPSLA Tutorial

Angelo CORSARO
Custom Filter Example

High-End 3D Visualization

Primary Radar

Secondary Radar

Filter based on Region-of-interest

- SQL Filter
- Custom Filter

Automatically Manage Relationships

Flight Data Processor
DDS: The Big Picture

Object-Oriented Reconstruction

- `aClass`
  - `attr1`
  - `attr2`
  - `op1()`
  - `op2()`

- `aContainer`
  - `op1()`
  - `op2()`

- `anotherClass`
  - `attrx`
  - `attry`
  - `opX()`
  - `opY()`

Relational Reconstruction

- SQL92 Filters
- Filter-Objects

- SQL92 Filters
- SQL92 Filters

OpenSplice DDS

OOPSLA Tutorial

Angelo CORSARO
DDS & CEP

- Present “Events” in a programmer-friendly manner
- In/Out Events
- Push Filtering some of the Distribution Middleware
- Predictable, High Performance
- Smart Distribution vs. Brute Force
OpenSplice DDS: Foundational Abstractions

- **Information Model.** Defines the structure, relations, & QoS, of the information exchanged by the applications, & supports *Simple, Relational, & Object Oriented Modeling*

- **Typed Global Data Space.** A logical data space in which applications *read* & *write* data *anonymously* & *asynchronously*, decoupled in space & time

- **Publisher/Subscriber.** Produce/Consume information into/from the Global Data Space

- **QoS.** Regulates the non-functional properties of information in the Global Data Space, e.g., reliability, availability, & timeliness, etc.
The Global Data Space can be organized into domains which in turn can have partitions.

Each partition is mapped to an IP Multicast Address.

QoS controls the availability & consistency of data.
OpenSplice DDS: Foundational Abstractions

- **Information Model.** Defines the structure, relations, & QoS, of the information exchanged by the applications, & supports **Simple, Relational, & Object Oriented Modeling**
- **Typed Global Data Space.** A logical data space in which applications read & write data **anonymously & asynchronously**, decoupled in space & time
- **Publisher/Subscriber.** Produce/Consume information into/from the Global Data Space
- **QoS.** Regulates the non-functional properties of information in the Global Data Space, e.g., reliability, availability, & timeliness, etc.
Publisher/DataWriter & Subscriber/DataReader

Publisher/DataWriter

- Publishers are responsible for managing the dissemination of publications
- The dissemination is driven by the QoS associated with the DataWriter, the Publisher, & the Topic
- A DataWriter is associated with only one Publisher & one Topic. It embeds the knowledge of dealing with the Topic’s Data Type

Subscriber/DataReader

- Subscribers are responsible for managing the reception of data resulting from subscriptions
- The presentation of data is driven by the QoS associated with the DataReader, the Publisher & the Topic
- A DataReader is associated with only one Subscriber & one Topic. It embeds the knowledge of dealing with the Topic’s Data Type

Network
Example: Create Domain Participant

- `DomainParticipant` object acts as factory for `Publisher`, `Subscriber`, `Topic` and `MultiTopic` entity objects

```cpp
// used to identify the participant
DomainId_t domain_id = ...;

// get the singleton factory instance
DomainParticipantFactory_var dpf =
    DomainParticipantFactory::get_instance();

// create domain participant from factory
DomainParticipant_var dp =
    dpf->create_participant (domain_id,
                             PARTICIPANT_QOS_DEFAULT,
                             NULL);
```
Example: Create Topic

......

// register the data type associated with the topic
FooDataType foo_dt;
foo_dt.register_type (dp,
        "Foo");

// create a topic
Topic_var foo_topic =
    dp->create_topic ("Foo_topic", //topic name
        "Foo", // type name
        TOPIC_QOS_DEFAULT, // Qos policy
        NULL); // topic listener
Example: Create Subscriber and DataReader

......

// create a subscriber from the domain participant
SubscriberQos sQos;
dp->get_default_subscriber_qos (sQos);
Subscriber_var s =
    dp->create_subscriber (sQos,
        NULL);

// create a data reader from the subscriber
// and associate it with the created topic
DataReader_var reader =
    s->create_datareader (foo_topic.in (),
        DATAREADER_QOS_DEFAULT,
        NULL);
Example: Create Publisher and DataWriter

......

// create a publisher from the domain participant
PublisherQos pQos;

dp->get_default_publisher_qos (pQos);
Publisher_var p =
    dp->create_publisher (pQos, NULL);

// create a data writer from the publisher
// and associate it with the created topic
DataWriter_var writer =
    p->create_datawriter (foo_topic.in (), DATAWRITER_QOS_DEFAULT, NULL);

// narrow down to specific data writer
FooDataWriter_var foo_writer =
    FooDataWriter::_narrow (writer);

// publish user-defined data
Foo foo_data;
foo_writer->write (foo_data);
Listener_var subscriber_listener = new MyListener();
foo_reader->set_listener(subscriber_listener);

MyListener::on_data_available(DataReader reader)
{
    FooSeq_var received_data;
    SampleInfoSeq_var sample_info;

    FooDataReader_var foo_reader =
        FooDataReader::_narrow (reader.in () );

    foo_reader->take(received_data.out (),
                        sample_info.out ());
    // Use received_data
    ......}

How to Get Data (Sync Wait-based)

```c
Condition_var foo_condition =
    reader->create_readcondition(ANY_SAMPLE_STATE,
                                ANY_LIFECYCLE_STATE);
WaitSet waitset;
waitset->attach_condition(foo_condition);

ConditionSeq_var active_conditions;
Duration_t timeout = {3,0};
waitset->wait(active_conditions.out (), timeout);
...
FooSeq_var received_data;
SampleInfoSeq_var sample_info;
reader->take_w_condition(received_data.out (),
                         sample_info.out (),
                         foo_condition);
// Use received_data
```
OpenSplice DDS: Foundational Abstractions

- **Information Model.** Defines the structure, relations, & QoS, of the information exchanged by the applications, & supports **Simple, Relational, & Object Oriented Modeling**

- **Typed Global Data Space.** A logical data space in which applications read & write data *anonymously* & *asynchronously*, decoupled in space & time

- **Publisher/Subscriber.** Produce/Consume information into/from the Global Data Space

- **QoS.** Regulates the non-functional properties of information in the Global Data Space, *e.g.*, reliability, availability, & timeliness, etc.
QoS Model

How is QoS matched in the System?

- QoS can be associated with all relevant OpenSplice DDS entities.
- Some QoS are matched based on a Request vs. Offered Model.
- Publications & Subscriptions match only if the declared & requested QoS are compatible.
  - e.g., it is not possible to match a publisher that delivers data unreliably with a subscriber that requires reliability.
### QoS Policy Summary

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Applicability</th>
<th>RxO</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DURABILITY</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>DURABILITY SERVICE</td>
<td>T, DW</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LIFESPAN</td>
<td>T, DW</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>HISTORY</td>
<td>T, DR, DW</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>PRESENTATION</td>
<td>P, S</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>RELIABILITY</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>PARTITION</td>
<td>P, S</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DESTINATION ORDER</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>OWNERSHIP</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>OWNERSHIP STRENGTH</td>
<td>DW</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>LIVELINESS</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>DEADLINE</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>LATENCY BUDGET</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>TRANSPORT PRIORITY</td>
<td>T, DW</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>TIME BASED FILTER</td>
<td>DR</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>RESOURCE LIMITS</td>
<td>T, DR, DW</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>USER_DATA</td>
<td>DP, DR, DW</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>TOPIC_DATA</td>
<td>T</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>GROUP_DATA</td>
<td>P, S</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

#### Data Availability

- Rich set of QoS allow to configure several different aspects of data availability, delivery & timeliness
- QoS can be used to control & optimize network as well as computing resource
Mapping QoS

Which properties does QoS controls?

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

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

- 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
Data Timeliness

### QoS Policy Table

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Applicability</th>
<th>RxO</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEADLINE</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>LATENCY BUDGET</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>TRANSPORT PRIORITY</td>
<td>T, DW</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>
The **DEADLINE** QoS policy can be used to define the **maximum inter-arrival time** between data samples

- 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
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.

- 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.
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.

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Applicability</th>
<th>RxO</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSPORT PRIORITY</td>
<td>T, DW</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>

VIP Data, stay clear!
OpenSplice DDS Data Availability

![Diagram showing the relationship between History, Lifespan, Ownership, Ownership Strength, Durability, and Data Availability]

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Applicability</th>
<th>RxO</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DURABILITY</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>DURABILITY SERVICE</td>
<td>T, DW</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>LIFESPAN</td>
<td>T, DW</td>
<td>-</td>
<td>Y</td>
</tr>
<tr>
<td>HISTORY</td>
<td>T, DR, DW</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

Data Availability
The **DURABILITY** QoS controls the data availability w.r.t. late joiners, specifically 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

The **DURABILITY_SERVICE** QoS provide control over configuration of the service that implements the transient & persistent durability features.
The **LIFESPAN** QoS policy allows to control what happens to stale data.

- It specifies the validity interval for data written by the DataWriter.
- The default validity interval is infinite.

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Applicability</th>
<th>RxO</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIFESPAN</td>
<td>T, DW</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>
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. 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.
- **On the DataWriter Samples** are kept until delivered to all known subscribers.
- **On the DataReader side** samples are kept until the application “takes” them.
Data Availability @ Work

- DDS’ publisher & subscriber dynamically discover each other
- Time & Space decoupling is highly configurable by relying on DDS’ rich set of QoS, such as Durability, Lifespan, etc.
Data Availability @ Work

- DDS’ publisher & subscriber dynamically discover each other
- Time & Space decoupling is highly configurable by relying on DDS’ rich set of QoS, such as Durability, Lifespan, etc.
### Data Delivery

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Applicability</th>
<th>RxO</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRESENTATION</td>
<td>P, S</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>RELIABILITY</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>PARTITION</td>
<td>P, S</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>DESTINATION ORDER</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>OWNERSHIP</td>
<td>T, DR, DW</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>OWNERSHIP STRENGTH</td>
<td>DW</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>
The **RELIABILITY** QoS indicate the level of guarantee offered by the DDS in delivering data to subscribers. 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.
Ownership

The OWNERSHIP QoS specifies whether it is allowed for multiple DataWriters to write the same instance of the data & if so, how these modifications should be arbitrated. Possible choices are:

- **Shared.** Multiple writers are allowed to update the same instance & all the updates are made available to the reader.

- **Exclusive.** Indicates that each instance can only be owned by one DataWriter, but the owner of an instance can change dynamically -- due to liveliness changes.

- The selection of the owner is controlled by the setting of the OWNERSHIP_STRENGTH QoS policy.
Ownership Strength

The **OWNERSHIP_STRENGTH** specifies the value of the “strength” used to arbitrate among DataWriters that attempt to modify the same data instance:

- Data instance are identified by the couple (Topic, Key)
- The policy applies only if the OWNERSHIP is EXCLUSIVE

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Applicability</th>
<th>RxO</th>
<th>Modifiable</th>
</tr>
</thead>
<tbody>
<tr>
<td>OWNERSHIP_STRENGTH</td>
<td>DW</td>
<td>-</td>
<td>Y</td>
</tr>
</tbody>
</table>

How strong are you? 83
Example: A DLRL Object Model

ObjectModel.idl

```idl
valuetype TrackMap; // IntMap<Track>
valuetype stringList; // List<string>

valuetype Track : DLRL_ObjectRoot {
    float x;
    float y;
    stringList comments;
    long w; // Local attribute
    Radar a_radar; // associated
};

valuetype Track3D : Track {
    float z;
};

valuetype Radar {
    TrackMap tracks; // associated
};
```
DRLR-DCPS Mapping
Example Mapped to DCPS

<table>
<thead>
<tr>
<th>T3D_TOPIC</th>
<th>RADAR_TOPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Oid</strong></td>
<td><strong>Z</strong></td>
</tr>
<tr>
<td>2</td>
<td>300</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TRACK_TOPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
</tr>
<tr>
<td>Track</td>
</tr>
<tr>
<td>Track3D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COMMENTS_TOPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Class</strong></td>
</tr>
<tr>
<td>Track</td>
</tr>
<tr>
<td>Track</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RADAR-TRACKS_TOPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R_Oid</strong></td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>11</td>
</tr>
</tbody>
</table>
Default Mapping file

Mapping.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Dlrl SYSTEM "dlrl.dtd">
<DLrl name="Example">
  <templateDef name="stringList" pattern="List" itemType="string"/>
  <templateDef name="TrackMap" pattern="IntMap" itemType="Track"/>
  <classMapping name="Track">
    <local name="w"/>
  </classMapping>
  <associationDef>
    <relation class="Track" attribute="a_radar"/>
    <relation class="Radar" attribute="tracks"/>
  </associationDef>
</DLrl>
```

ObjectModel.idl

```c
valuetype TrackMap;    // IntMap<Track>
valuetype stringList;  // List<string>

valuetype Track : DLRL_ObjectRoot {
  float x;
  float y;
  stringList comments;
  long w;       // Local attribute
  Radar a_radar; // associated
};

valuetype Radar {
  TrackMap tracks; // associated
};
```

TopicModel.idl

```c
struct COMMENTS_TOPIC {
  string Class;   // key
  DLRLOid Oid;    // key
  long Index;     // key
  string Comments; }

struct RADAR-TRACKS_TOPIC {
  DLRLOid R_Oid;  // key
  long Index;     // key
  string T_Class;
  DLRLOid T_Oid;  }
```

mapping.xml

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Dlrl SYSTEM "dlrl.dtd">
<DLrl name="Example">
  <templateDef name="stringList" pattern="List" itemType="string"/>
  <templateDef name="TrackMap" pattern="IntMap" itemType="Track"/>
  <classMapping name="Track">
    <local name="w"/>
  </classMapping>
  <associationDef>
    <relation class="Track" attribute="a_radar"/>
    <relation class="Radar" attribute="tracks"/>
  </associationDef>
</DLrl>
```
<classMapping name="Track">
  <mainTopic name="TRACK_TOPIC">
    <keyDescription content="FullOid">
      <keyField>Class</keyField>
      <keyField>Oid</keyField>
    </keyDescription>
  </mainTopic>
  <monoAttribute name="x">
    <valueField>X</valueField>
  </monoAttribute>
  <monoAttribute name="y">
    <valueField>Y</valueField>
  </monoAttribute>
  <multiAttribute name="comments">
    <multiPlaceTopic name="COMMENTS_TOPIC" indexField="Index">
      <keyDescription content="FullOid">
        <keyField>CLASS</keyField>
        <keyField>OID</keyField>
      </keyDescription>
    </multiPlaceTopic>
    <valueField>COMMENT</valueField>
  </multiAttribute>
  <monoRelation name="a_radar">
    <keyDescription content="SimpleOid">
      <keyField>Radar</keyField>
    </keyDescription>
  </monoRelation>
</classMapping>
DLRL: How does it Work?

Concepts

The mechanism at the foundation is a managed Object Cache:

- An Object Cache can be populated by different types (classes) of Objects
- Each object class has its own manager called an ObjectHome
  - They can inform the application about object creation/modification/deletion
- Classes may contain navigable relationships to other classes
- Each Object class may inherit from 1 other Object class
DLRL: How does it Work?

- ‘vanilla DDS’: updates arrive as separate samples at separate times.
- OpenSplice DDS Object Technology: updates are processed in ‘update rounds’:
  - ObjectHomes read all available samples from the DDS information backbone & update their corresponding objects in the Cache accordingly.
  - Objects are allocated once & their state is ‘overwritten’ on subsequent updates.
  - Therefore an Object always contains the latest available state.
  - Push mode: update rounds start when new data arrives. The application gets notified by Listeners.
  - Pull mode: the application can determine the start of each update round manually.
Object Caches offer two ways to notify an application of incoming information:

- Listeners can be triggered for each modification of an object’s state
  - Listeners registered to the Cache indicate the start & end of each update round
  - Listeners registered to the ObjectHome pass each modification back as a callback argument
  - With a simple mechanism that can be translated into callbacks for Listeners on individual objects
- It is possible to get a separate list of all objects that have been created/modified/deleted in the current update round

```plaintext
on_object_modified()
on_object_created()
attach_listener(on_begin_updates())
attach_listener(on_end_updates())
get_modified_objects()
```
CacheAccess: Examining objects in isolation

Some applications want to be able to store temporal ‘snapshots’:

- A CacheAccess can be used to contain a temporal graph of objects
- Objects must physically be cloned from Cache to CacheAccess
- A CacheAccesses is not automatically kept in sync with the main Cache
- A ‘refresh’ operation can be used to resync the contents of CacheAccess with the contents of the main Cache
Some applications want to be able to modify or create certain objects:

- An initial set of Objects may be cloned into a writeable CacheAccess
- Available objects may then be modified locally
- New objects can be created in the CacheAccess as well
- The ‘write’ operation instructs the ObjectHomes to write any modifications into the system
Using Selections to Manage Subsets

A Selection mechanism can keep track of subsets of information:

- Selections are created & managed by the ObjectHomes
- A Criterion plugged into a Selection determines the boundaries of a subset:
  - A QueryCriterion determines boundaries based on an SQL statement
  - A FilterCriterion determines boundaries based on user-defined callback filters
- Selections can notify the application when objects enter & leave it
Your First DLRL Application
Example: DDS based Chatroom

Chatroom information backbone

ChatMessage {
    int userID;
    string Message;
}

NameService {
    int userID;
    string Name
}

Key: userID
QoS: Reliable,
Volatile

Key: userID
QoS: Reliable,
Transient

SHOWS ALL MESSAGES

Message Board

Chatter
name = Marge
userID = 1

Chatter
name = Lisa
userID = 2

Topic-model

Angelo CORSARO

OOPSLA Tutorial
Example: Object Model for the chatroom

- ChatMessage & User Objects reflect the state of the corresponding ChatMessage & NameService Topics
- Related keyfields (i.e. the userID fields) can be mapped onto navigable Relationships
- Objects may add local operations to manipulate/display their state
Example: Using a WhiteList

Chatroom information backbone

- **Message Board**
  - Shows all messages

- **WhiteListed MessageBoard**
  - Only shows messages from users on your whitelist

- **NameService**
  - Key: userID
  - QoS: Reliable, Volatile

- **WhiteList**
  - Key: name
  - QoS: Reliable, Transient

**ChatMessage**
- int userID;
- string Message;

**Chatroom**
- Marge (name = Marge, userID = 1)
- Lisa (name = Lisa, userID = 2)
Example: WhiteListed MessageBoard

A WhiteList object contains a set of users whose messages should not be blocked by the WhiteListedMessageBoard.

Multiple WhiteList objects may be created for different purposes.

A WhiteList object is identified by name, e.g., “family”, “friends”, “colleagues”, etc.
### Object Model

- **ObjectRoot**
  - **ChatMessage**:
    - id: long
    - message: String
    - display()
  - **User**:
    - id: long
    - name: String
  - **WhiteList**:
    - name: String
    - addUser()
    - displayAllUsers()
  - **UserSet**:
    - members

### Topic Model

- **ChatMessageTopic**:
  - id: long
  - message: String
- **UserNameTopic**:
  - name: String
  - userID: long
- **WhiteListTopic**:
  - name: String
  - userID: long
  - addUser()
Example: using a WhiteList – sequence of events

- WhiteListEditor
  - Create a WhiteList object with a unique name
  - Add a number of users to its members set
  - Write the WhiteListObject into DDS

- WhiteListedMessageBoard
  - Select a WhiteListObject by name
  - Get all New & all Modified ChatMessage Objects
  - Execute the following algorithm:

```java
for (int i = 0; i < chatMessages.length(); i++) {
    if (whiteList.friends.contains(ChatMessage[i].sender())) {
        ChatMessage[i].display();
    } else {
        // Message blocked !!
    }
}
```
Lingua Franca

Problem

- Designing large-scale interoperable distributed systems, & system of systems, is a very complex engineering endeavor
- Functional & Object-Oriented decomposition have proven to be powerful methodologies, but yet, often lead to tightly coupled systems (see 4W)
- The key challenge lies in the inherent fragility of interfaces which tend to change often throughout the lifetime of the system

Context

- Most of the current practice in designing distributed system is based on a functional or OO decomposition whose goal is that of identifying the key Interfaces
- Different component of the system cooperate agreeing on interfaces, & invoking methods over these interfaces
- Examples are distributed systems based on CORBA, .NET, J2EE, Java RMI
- However these systems are fragile with respect to extensibility as well as integration with other technologies
**Lingua Franca**

**Solution**

- Focus on identifying the information model, *i.e.*, **data & relationships**, underlying the distributed system, the **Lingua Franca**
- Information exchanged within & across a system is much more stable than functional interfaces
- The Lingua Franca provides the fabric that keeps together the system along with the QoS invariant capturing the non-functional requirements

**Related Pattern**

- **Global Data Space.** The **Lingua Franca** is often used along with the Global Data Space Pattern

**Known Uses**

- DDS
- DBMSs
Replication Patterns
Hot-Swap

Problem

- Business- & Mission-Critical Systems cannot usually afford to fail in delivering their services in case of HW or SW failure
- Failure need to be properly masked, & gracefully tolerated by the system

Context

- In order to meet challenging throughput requirements many of these systems want to minimize the performance implication induced by providing high availability
- Moreover, for latency-sensitive system the *switch-over* has to happen in a short, & predictable time interval

Forces

- Switch over overhead should be minimized
- Networking resources should be used parsimoniously
Hot-Swap

Solution

- Replicate service in your system so to have only one of them actively produces data-updates
- The other replicas are also *hot* in the sense that they operate mirroring the primary behavior but not having impact on the system status

Hot-Swap in DDS

- This behavior can be obtained by using in combination the OWNERSHIP & OWNERSHIP_STRENGTH QoS Policies
- Exclusively owned Topics can be modified by a single writer
- Writer strength is used to coordinate replicated writers
Hot-Swap Example

P
STRENGTH=3

P'
STRENGTH=2

P"
STRENGTH=1

S1
{symbol = “AAPL”}

S2
{quote > 500}

S3
{ name =”Microsoft Coorp.”}
Hot-Swap Example

STRENGTH=1

S1
{symbol = “AAPL”}

S2
{quote > 500}

S3
{name = “Microsoft Coorp.”}
Problem

‣ Business- & Mission-Critical Systems cannot usually afford to fail in delivering their services in case of HW or SW failure
‣ Failure need to be properly masked, & gracefully tolerated by the system

Context

‣ In order to meet challenging throughput requirements many of these systems want to minimize the performance implication induced by providing high availability
‣ Moreover, for latency-sensitive system the switch-over has to happen in a short, & predictable interval of time

Forces

‣ Latency minimization is the foremost goal
‣ Network bandwidth is not necessarily an issue
Hot-Hot

Solution

- Use Hot-Hot replicas could be used to ensure that always, & only, the *fastest sample* is delivered
- Hot-Hot replicas send data which is delivered to subscribers, filtering is achieved on the receiving side

Hot-Hot in DDS

- This can be achieved in a few different manners, the most effective being a combination of DESTINATION_ORDER QoS & write_with_timestamp()
Hot-Hot Example

T1 Data Feed

Satellite Data Feed

Another Data Feed

{symbol = "AAPL"}

{quote > 500}

{ name = "Microsoft Corp."}
Traffic Engineering Patterns
Singleton Communication

Problem

‣ Complete control over resources is a must for all mission- & business-critical applications
‣ For distributed applications the ability to control networking resources is fundamental to ensure predictable, scalable, & dependable behaviour

Context

‣ It often happens that distributed applications start misbehaving when there are temporary overload conditions
‣ In the lucky cases, the system ends up providing degraded QoS, but in some unfortunate, but yet not so rare cases, the system might miss some fundamental QoS, e.g., not delivering an alarm within the allocated time
‣ These overload conditions are often to control at an application level because often there is no coordinated way of agreeing what should be done
Singleton Communication

Solution

- Rely on a single service per node for dealing with the network traffic management.
- This solution ensures that global properties can be enforced, resources “fire walled”, & QoS properly enforced.

Example: Singleton Communication in OpenSplice DDS
Problem

- Real-Time distributed systems need ensure that priority is enforced end-to-end
- Specifically, priority has to be ensured & enforced on data transmission, to avoid unbounded priority inversion

Context

- Priority is inherent to many applications. This fact is not always acknowledge, but there are some event or information that are inherently more important then other, e.g., alarms, high rate data, etc.
- Sometimes, data priorities is hard-coded in the application, but this hinders reuse & extensibility
VIP Data

Solution

- Associate a priority with data & use it to express a importance

VIP Data in DDS

- The priority of data is expressed by means of the TRANSPORT_PRIORITY QoS Policy
- This QoS is used by DDS to control the importance of data & ensure priority preemptive behaviour
VIP Data

- Transport Priority can be associated with the Topic
- All Data Writer Created for this Topic will use the same priority

```c
TopicQos tqos = TOPIC_QOS_DEFAULT;
tqos.transport_priority.value = 75;

// -- Create Topic
Topic_var pingMsgTopic =
    participant->create_topic("PingMsg",
                         pingMsgTN.c_str(),
                         tqos,
                         NULL,
                         ANY_STATUS);
```

- Transport Priority can be overridden per Writer
- Transport Priority can also be changed once the Data Writer is created

```c
// // Write at the priority defined for the topic (45)
// pingMsgWriter->write(msg, HANDLE_NIL);
// ...

// // If needed, priority can be changed with set_qos
// dwqos.transport_priority.value = 90;
pingMsgWriter->set_qos(dwqos);

// Write at the new priority (90)
pingMsgWriter->write(msg, HANDLE_NIL);
```
Priority Lanes

Problem

- Real-Time distributed systems need ensure that priority is enforced end-to-end
- Specifically, priority has to be ensured & enforced on data transmission, to avoid unbounded priority inversion

Context

- Most of the Distributed Real-Time Systems which are being designed & deployed today run on non-real-time networks, such as TCP-UDP/IP & Ethernet
- How can transport priorities be enforced on these transports so that the system maintains control over priority inversion?
Priority Lanes

Solution

‣ Ensure that communication throughout the systems happens via a configurable set of **Priority Lanes** so to ensure that high priority data, always find its way preemting lower priority data

‣ With **Priority Lanes**, priority inversion is completely under control of the Engineer that design/deploys the system

Related Patterns

‣ **VIP Data.** Priority Lanes are often used as the mechanism to implement VIP Data
Problem

- Distributed systems have to co-exist with inherent trade-off between latency & throughput.
- However, most of currently available middleware do not provide any way of controlling this trade-off while preserving certain time properties associated with data.

Context

- The inability to control the latency-throughput trade off often leads to poor network utilization & excessive CPU usage.
- Application characterized by small messages produced at high rate, can easily get trapped into
Solution

- Use a time budget to express the amount of time that can be used in order to exploit buffering on the sending side.
- As a result bundling can take place in order to reduce CPU utilization & improve network utilization.

Budget in DDS

- The LATENCY_BUDGET QoS is used to express the latency budget available for exploiting buffering.
- This QoS allows to control the latency without data’s violating its temporal properties.
Traffic Engineering Patterns Relationship

- Priority Inversion/Resources Tradeoff
- Expediting Data
- Latency/Throughput Tradeoff
- Scheduling
- Traffic Partitioning

VIP Data → Priority Lanes

Priority Lanes → Singleton Network

Singleton Network → Divide et Impera

Divide et Impera → Budget

Budget → Priority Inversion/Resources Tradeoff
Putting it all Together

Example: A DDS Real-Time Networking Service
Table of Contents

- Challenges
- OMG DDS
- DDS Patterns
- Use Cases
- Concluding Remarks
DDS Recommendations

Mandated/Recommended by:

- **US Navy**: Open Architecture
- **DISA**: Net-centric Systems
- **EuroControl**: Air Traffic Control Center Operational Interoperability
Use Cases

Defense
- Combat Management Systems
- Flycatcher system
- Tactical Links
- Radar Processing
- Submarine Systems
- Future Combat Systems

SCADA/Utilities
- Industrial Automation
- Power Grids
- Rocket Launch Systems
Use Cases

Transportation
- Air Traffic Management/Control
- Metropolitan Traffic Management
- Underground Metropolitan Transportation

Financial Services
- Automated Trading Firms
- Compliance Systems
- Market Data Platforms
The DDS is the only technology that spans across the board -- It guarantees exceptional real-time behavior, while providing unparalleled level of throughput!
Table of Contents

- Challenges
- OMG DDS
- DDS Patterns
- Use Cases
- Concluding Remarks
Why DDS?

Most Hyped

- Low latency & predictable data dissemination
- Support for several 100K messages/sec, up to Millions messages/sec

Not so Hyped, yet Fundamentally Important

- Stability under overload condition
- Scalability
- Fairness
- Control over latency/throughput tradeoffs
- Traffic Engineering
  - HW Filtering
  - Traffic Shaping
  - Priority driven delivery
- High Performance Persistence
- Event Processing

Other

- Security
- DBMS/Web Services Connectivity

Optimally addresses the requirements of next generation Mission/Business-Critical Applications
Extensible & Dynamic Topic Types

- Extending the DDS Type System to natively support Topic Types extensibility without jeopardizing performance/efficiency
- Dynamic APIs for DDS

Native C++ PSM for DDS

- Avoid IDL2C++ mapping traps/pitfalls
- Provide a DDS API that leverages fully leverages C++

Native Java PSM for DDS

- Avoid IDL2Java mapping traps/pitfalls
- Provide a DDS API that leverages fully leverages & integrates with Java technologies, e.g., annotations, hibernate for ORM, JMS, etc.
Resource Center

- http://www.prismtech.com/opensplice-dds/

Archived Webcasts


OMG DDS Information

- http://www.dds-forum.org
- http://portals.omg.org/dds/
- http://dds4u.blogspot.com/

Thank You!
Suggested Reading

Data Distribution Service


Patterns


- Frank Buschman, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture*


- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns—Elements of Reusable OO Software*