Data Distribution Service (DDS) Tutorial

dds/2006-04-05
Gerardo Pardo-Castellote - RTI
Virginie Watine - THALES
26 April 2006
The DDS Standard

Data Distribution Service for Real-Time Systems

- Adopted in June 2003
- Finalized in June 2004
- Revised June 2005
- Joint submission (RTI, THALES, OIS)
- Specification of API required to facilitate the Data-Centric Publish-Subscribe communication environment for real-time distributed systems.
DDS Layers

- DDS made of two layers
  - DCPS = Data Centric Publish/Subscribe
    - Purpose: distribute the data
    - Close to Relational model
  - DLRL = Data Local Reconstruction Layer
    - Purpose: provide an object-based model to access data ‘as if’ it was local

Diagram:
- Application
  - DLRL
  - DCPS
Data Distribution Service - DCPS
Outline

- Background
  - Middleware information models
  - Publish / Subscribe
  - Topic-based Publish / Subscribe

- Focus on Topics
  - Topic definition, keys
  - ContentFilteredTopic, MultiTopic

- Publication & Subscription
  - Related DDS Entities
  - DDS Publication
  - DDS Subscription
  - Dual mechanism to access incoming information
    - Listeners
    - WaitSets and Conditions

- Quality of Service
Outline

- Background
  - Middleware information models
  - Publish / Subscribe
  - Topic-based Publish / Subscribe

- Focus on Topics
  - Topic definition, keys
  - ContentFilteredTopic, MultiTopic

- Publication & Subscription
  - Related DDS Entities
  - DDS Publication
  - DDS Suscription
  - Dual mechanism to access incoming information
    - Listeners
    - WaitSets and Conditions

- Quality of Service
Middleware Information Models

**Point-to-Point**
- Telephone, TCP
- Simple, high-bandwidth
- Leads to stove-pipe systems

**Client-Server**
- File systems, Database, RPC, CORBA, DCOM
- Good if information is naturally centralized
- Single point failure, performance bottlenecks

**Publish/Subscribe Messaging**
- Magazines, Newspaper, TV
- Excels at *many-to-many communication*
- Excels at distributing *time-critical information*

**Replicated Data**
- Libraries, Distributed databases
- Excels at data-mining and analysis

© 2006 RTI and THALES - All rights Reserved
Publish Subscribe Model

Efficient mechanism for data communications

Reporter does not need to know where subscribers live.

Subscribers do not need to know where reporter lives.

Data Producer

Middleware

Consumers

© 2006 RTI and THALES - All rights Reserved
Topic-Based Publish-Subscribe

Track

Command

Publishers

Subscribers

© 2006 RTI and THALES - All rights Reserved
Topic-Based Publish-Subscribe

- Publish-subscribe allows infrastructure to prepare itself...
- ... Such that when the data is written it is directly sent to the subscribers
DDS/DCPS

Provides a “Global Data Space” that is accessible to all interested applications.

- Data objects addressed by Topic and Key
- Subscriptions are decoupled from Publications
- Contracts established by means of QoS
- Automatic discovery and configuration
Outline

- Background
  - Middleware information models
  - Publish / Subscribe
  - Topic-based Publish / Subscribe

- Focus on Topics
  - Topic definition, keys
  - ContentFilteredTopic, MultiTopic

- Publication & Subscription
  - Related DDS Entities
  - DDS Publication
  - DDS Subscription
  - Dual mechanism to access incoming information
    - Listeners
    - WaitSets and Conditions

- Quality of Service
Topics

**Central DDS Entity**

- Topics gather all 'instances' of a given data type related to a given purpose

- **Topic** *Keys*
  - Needed to model dynamic objects (e.g. tracks)
  - Can dramatically decrease system size
  - Used for reliable many-to-one (i.e. ALARM topic)

- **Topic** *QoS*
  - Convenient way to describe information model

- Data types specified in *IDL* and can be reused
  - eg a data structure received by DDS can be used as is in a CORBA call

- ContentFilteredTopic and MultiTopic control subscription scope
Topic Definition

Good topic definition leads to:

- Better interfaces
- Easier integration
- Improved scalability
- Decreased system size
- Faster startup and discovery times

Choosing the proper Topics is the central design decision

- By Sender “Role” - MixerTank3Data
- By Receiver “Role” - AirTrackCorrelator
- By Message ID - Filter23ToGUI12
- By Data "Role" - AAWTracks
- By Data Type - CommandString
Example without Keys

When **not** using **keys**:

- Each Topic corresponds to a single data instance.
- A DataWriter associated with a Topic can write to the instance corresponding to That topic.
- Multiple DataWriters may write to the same instance.
- A DataReader specifies the Topic (instance) it wants to receive updates from.
Example with Keys

Address in Global Data Space = (Topic, Key)
- Each Topic corresponds to multiple data instances
- Each DataWriter can write to multiple instances of a single Topic
- Multiple DataWriters may write to the same instance
- Each DataReader can receive updates from multiple instances of a single Topic
- Multiple DataReaders may read from the same instances
Data Instances Addressing: Keys

- Address in Global Data Space = (Topic, Key) => multiple instances of the same Topic
- Used to sort specific instances
- Do not need a separate Topic for each data-object instance
- Topic key can be any field within the Topic.

Example:

```c
struct LocationInfo {
    int LocID; //key GPSPos pos;
};
```
DDS Subscription (ContentFilteredTopic)

The Filter Expression and Expression Params will determine which instances of the Topic will be received by the subscriber.
** DDS Subscription (MultiTopic) **

** Listeners Wait-Set or conditions available **

** MultiTopics can combine, filter and rearrange data from multiple topics **
Outline

- Background
  - Middleware information models
  - Publish / Subscribe
  - Topic-based Publish / Subscribe

- Focus on Topics
  - Topic definition, keys
  - ContentFilteredTopic, MultiTopic

- Publication & Subscription
  - Related DDS Entities
  - DDS Publication
  - DDS Suscription
  - Dual mechanism to access incoming information
    - Listeners
    - WaitSets and Conditions

- Quality of Service
DDS Communication Model

- Publisher declares information it has by specifying the Topic...
  - ...and the offered QoS contract
  - ...and an associated listener to be alerted of any significant status changes
- Subscriber declares information it wants by specifying the Topic...
  - ...and the requested QoS contract
  - ...and an associated listener to be alerted of any significant status changes
- DDS automatically discovers publishers and subscribers
  DDS ensures QoS matching and alerts of inconsistencies
**DCPS Entities**

- **Publisher**: Represents participation of the application in the communication collective.
- **DataWriter**: Accessor to write typed data on a particular Topic.
- **DomainParticipant**: Aggregation of DataWriter objects.
  Responsible for disseminating information.
- **Subscriber**: Accessor to read typed data regarding a specific Topic.
- **DataReader**: Aggregation of DataReader objects.
  Responsible for receiving information.
Domains and Participants

- Domain
- DomainParticipant
- Node

Node 1 connects to DomainParticipant 1, Domain 2, and Node 2.
Node 2 connects to DomainParticipant 1, Domain 1, and Node 3.
Node 3 connects to DomainParticipant 1, Domain 1, and Node 2.
Domain Partitioning

DomainParticipant

Node

Domain1

 DW 1
 DW 2
 DW 3
 DW 4
 DW 5

DR 1
 DR 2
 DR 3

Instance

Domain2

 DW 1
 DW 2
 DW 3
 DW 4
 DW 5

DR 4
 DR 5
 DR 6

Instance

Instance

Topic “green”  Topic “orange”
User Application:
- Creates related DDS entities
  - Publisher
  - Topic
  - DataWriter
- Configures entities' QoS then
- Provides data to DataWriter
// Entities creation
Publisher publisher = domain->create_publisher(
    publisher_qos,
    publisher_listener);

Topic topic = domain->create_topic(
    "Track", "TrackStruct",
    topic_qos, topic_listener);

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

TrackStructDataWriter twriter =
    TrackStructDataWriter::narrow(writer);

TrackStruct my_track;
// (Repeat each time data needs to be written)
twriter->write(&my_track);
**User Application:**

- Creates related DDS entities
  - Subscriber
  - Topic
  - DataReader
- Configures entities' QoS and attach listeners
- Receives Data from DataReader through attached listeners
**User Application:**
- Creates related DDS entities
  - Subscriber
  - Topic
  - DataReader
- Configures entities' QoS
- Creates a Condition and attaches it to a WaitSet
- Waits on the WaitSet until data arrive, then picks it on the DataReader
Example: Subscription

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

Topic topic = domain->create_topic(
    "Track", "TrackStruct",
    topic_qos, topic_listener);

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

// Use listener-based or wait-based access
// Listener creation and attachment
Listener listener = new MyListener();
reader->set_listener(listener);

// Listener code
MyListener::on_data_available( DataReader reader )
{
    TrackStructSeq received_data;
    SampleInfoSeq sample_info;
    TrackStructDataReader treader =
        TrackStructDataReader::narrow(reader);

    treader->take( &received_data,
                   &sample_info, ...)
    // Use received_data
}
How to Get Data? (WaitSet-Based)

// Creation of condition and attachment
Condition foo_condition =
    treader->create_readcondition(...);
waitset->add_condition(foo_condition);

// Wait
ConditionSeq active_conditions;
waitset->wait(&active_conditions, timeout);
// active_conditions[0] == foo_condition
// => data is there, ready to be picked
FooSeq received_data;
SampleInfoSeq sample_info;

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

<table>
<thead>
<tr>
<th>STATUS</th>
<th>Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCONSISTENT_TOPIC</td>
<td>Topic</td>
</tr>
<tr>
<td>DATA_ON_READERS</td>
<td>Subscriber</td>
</tr>
<tr>
<td>LIVELINESS_CHANGED</td>
<td>DataReader</td>
</tr>
<tr>
<td>REQUESTED_DEADLINE_MISSED</td>
<td>DataReader</td>
</tr>
<tr>
<td>REQUESTED_INCOMPATIBLE_QOS</td>
<td>DataReader</td>
</tr>
<tr>
<td>DATA_AVAILABLE</td>
<td>DataReader</td>
</tr>
<tr>
<td>SAMPLE_LOST</td>
<td>DataReader</td>
</tr>
<tr>
<td>SUBSCRIPTION_MATCH</td>
<td>DataReader</td>
</tr>
<tr>
<td>LIVELINESS_LOST</td>
<td>DataWriter</td>
</tr>
<tr>
<td>OFFERED_INCOMPATIBLE_QOS</td>
<td>DataWriter</td>
</tr>
<tr>
<td>PUBLICATION_MATCH</td>
<td>DataWriter</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>STATUS</th>
<th>Entity</th>
<th>Listener operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCONSISTENT_TOPIC</td>
<td>Topic</td>
<td>on_inconsistent_topic</td>
</tr>
<tr>
<td>DATA_ON_READERS</td>
<td>Subscriber</td>
<td>on_data_on_readers</td>
</tr>
<tr>
<td>LIVELINESS_CHANGED</td>
<td>DataReader</td>
<td>on_liveliness_changed</td>
</tr>
<tr>
<td>REQUESTED_DEADLINE_MISSED</td>
<td>DataReader</td>
<td>on_requested_deadline_missed</td>
</tr>
<tr>
<td>REQUESTED_INCOMPATIBLE_QOS</td>
<td>DataReader</td>
<td>on_requested_incompatible_qos</td>
</tr>
<tr>
<td>DATA_AVAILABLE</td>
<td>DataReader</td>
<td>on_data_available</td>
</tr>
<tr>
<td>SAMPLE_LOST</td>
<td>DataReader</td>
<td>on_sample_lost</td>
</tr>
<tr>
<td>SUBSCRIPTION_MATCH</td>
<td>DataReader</td>
<td>on_subscription_match</td>
</tr>
<tr>
<td>LIVELINESS_LOST</td>
<td>DataWriter</td>
<td>on_liveliness_lost</td>
</tr>
<tr>
<td>OFFERED_INCOMPATIBLE_QOS</td>
<td>DataWriter</td>
<td>on_offered_incompatible_qos</td>
</tr>
<tr>
<td>PUBLICATION_MATCH</td>
<td>DataWriter</td>
<td>on_publication_match</td>
</tr>
</tbody>
</table>
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
Outline

- Background
  - Middleware information models
  - Publish / Subscribe
  - Topic-based Publish / Subscribe
- Focus on Topics
  - Topic definition, keys
  - ContentFilteredTopic, MultiTopic
- Publication & Subscription
  - Related DDS Entities
  - DDS Publication
  - DDS Subscription
  - Dual mechanism to access incoming information
    - Listeners
    - WaitSets and Conditions
- Quality of Service
QoS Contract “Request / Offered”

QoS Request / Offered: Ensure that compatible QoS parameters are set.

QoS: Durability
QoS: Presentation
QoS: Deadline
QoS: Latency_Budget
QoS: Ownership
QoS: Liveliness
QoS: Reliability

Communication not established

QoS not compatible
QoS: RELIABILITY

BEST EFFORT
Sample delivery is not guaranteed

RELIABLE
Sample delivery is guaranteed

Data Writer R
Publisher

Data Writer BE
Publisher

Data Reader BE
Subscriber

Missed samples

S1 S2 S3 S4 S5 S6 S7

S1 S2 S3 S4 S5 S6 S7

© 2006 RTI and THALES - All rights Reserved
**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
State Propagation

- System state
  - Information needed to describe future behavior of the system
    - System evolution defined by state and future inputs.
  - Minimalist representation of past inputs to the system

- State variables
  - Set of data-objects whose value codifies the state of the system

- Relationship with DDS
  - DDS well suited to propagate and replicate state
  - Topic+key can be used to represent state variables
  - KEEP_LAST history QoS exactly matches semantics of state-variable propagation

Present in many RT applications
Key ingredient for fault-tolerance
QoS: DEADLINE

DEADLINE
“deadline period”

Data Writer
Commits to provide data each deadline period.
Publisher

Failed to get data
Listener

Data Reader
Subscriber

Expects data every deadline period.

S X S S S S S S
QoS: LIVELINESS – Type, Duration

"type"
• AUTOMATIC = Infrastructure Managed
• MANUAL = Application Managed

Domain Participant

Topic

Data Writer
Publisher

Data Reader
Subscriber

Listener

Failed to renew lease

lease_duration

Liveliness Message
QoS: TIME_BASED_FILTER

“minimum_separation”: Data Reader does not want to receive data faster than the min_separation time

Data Writer
Publisher

Topic
Data Reader
Subscriber

Domain Participant

Discarded samples

Data Samples
minimum separation
QoS: OWNERSHIP_STRENGTH

OWNERSHIP_STRENGTH:
Specifies which writer is allowed to update the values of data-objects

Note: Only applies to Topics with OWNERSHIP=Exclusive
QoS: LATENCY_BUDGET

- Intended to provide time-critical information to the publisher for framework tuning where possible.
- Will not prevent data transmission and/or receipt.

Latency = t1 + t2 + t3
QoS: RESOURCE_LIMITS

- Specifies the resources that the Service can consume to meet requested QoS

**max_samples_per_instance:** max # data samples per instance

**max_instances:** max # instances for a single DW or DR

**max_samples:** max # data samples for a single DW or DR, across all instances
QoS: USER_DATA

- User-defined portion of Topic metadata
- Example of use: Security Authentication

USER_DATA can be used to authenticate an origination entity.

Note: USER_DATA is contained within the DDS metadata.
QoS: PARTITION

Logical “namespace” for topics

Domain Participant

Topic A  Topic B  Topic C

Data Writer  Data Writer  Data Writer

Publisher  Partition U,W

Data Reader  Data Reader  Data Reader

Subscriber  Partition U,Z

Subscriber  Partition X,Y

** Partition string names must match between publisher and subscriber
QoS: DURABILITY

Determines if/how instances of a topic are saved.

Durability Kind:
- VOLATILE – No Instance History Saved
- TRANSIENT – History Saved in Local Memory
- PERSISTENT – History Saved in Permanent storage

# saved in Transient affected by QoS: History and QoS: Resource_Limits
QoS: PRESENTATION

Governs how related data-instance changes are presented to the subscribing application.

Type: Coherent Access and Ordered Access

- Coherent access: All changes (as defined by the Scope) are presented together.
- Ordered access: All changes (as defined by the Scope) are presented in the same order in which they occurred.

Scope: Instance, Topic, or Group

- Instance: The scope is a single data instance change. Changes to one instance are not affected by changes to other instances or topics.
- Topic: The scope is all instances by a single Data Writer.
- Group: The scope is all instances by Data Writers in the same Subscriber.
## QoS: Quality of Service (1/2)

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Concerns</th>
<th>RxO</th>
<th>Changeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEADLINE</td>
<td>T,DR,DW</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>LATENCY BUDGET</td>
<td>T,DR,DW</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>READER DATA LIFECYCLE</td>
<td>DR</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>WRITER DATA LIFECYCLE</td>
<td>DW</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>TRANSPORT PRIORITY</td>
<td>T,DW</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>LIFESPAN</td>
<td>T,DW</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>LIVELINESS</td>
<td>T,DR,DW</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>TIME BASED FILTER</td>
<td>DR</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>RELIABILITY</td>
<td>T,DR,DW</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>DESTINATION ORDER</td>
<td>T,DR</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
## QoS: Quality of Service (2/2)

<table>
<thead>
<tr>
<th>QoS Policy</th>
<th>Concerns</th>
<th>RxO</th>
<th>Changeable</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER DATA</td>
<td>DP, DR, DW</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>TOPIC DATA</td>
<td>T</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>GROUP DATA</td>
<td>P, S</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>ENTITY FACTORY</td>
<td>DP, P, S</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>PRESENTATION</td>
<td>P, S</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>OWNERSHIP</td>
<td>T</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>OWNERSHIP STRENGTH</td>
<td>DW</td>
<td>N/A</td>
<td>YES</td>
</tr>
<tr>
<td>PARTITION</td>
<td>P, S</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>DURABILITY</td>
<td>T, DR, DW</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>HISTORY</td>
<td>T, DR, DW</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>RESOURCE LIMITS</td>
<td>T, DR, DW</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
DDS-DCPS Summary

- DDS targets applications that need to distribute data in a real-time environment
- DDS is highly configurable by QoS settings
- DDS provides a shared “global data space”
  - Any application can publish data it has
  - Any application can subscribe to data it needs
  - Automatic discovery
  - Facilities for fault tolerance
  - Heterogeneous systems easily accommodated
Data Distribution Service - DLRL
What is DLRL?
- Purpose of DLRL
- DLRL objects characteristics
- Mapping DLRL <-> DCPS

Main DLRL entities
- Cache, CacheAccess, ObjectRoot, ObjectHome Collections, Relations
- How does that work?
  - Initialisation
  - Reading objects
  - Writing objects

Other DLRL Entities - helpers
- Selections, Criterions, Listener
Outline

- What is DLRL?
  - Purpose of DLRL
  - DLRL objects characteristics
  - Mapping DLRL <-> DCPS

- Main DLRL entities
  - Cache, CacheAccess, ObjectRoot, ObjectHome Collections, Relations
  - How does that work?
    - Initialisation
    - Reading objects
    - Writing objects

- Other DLRL Entities - helpers
  - Selections, Criterions, Listener
What is DLRL? (1/2)

DLRL = Data Local Reconstruction Layer

- **Purpose** = provide a more 'natural' access to data
  - Object-oriented - using native language constructs
  - Management of graphs of objects
- Can be built on top of DCPS

- **Object model can be local** (no need for a global one)

Topics - need to be shared

DLRL classes are only local
What is DLRL? (2/2)

- With DLRL, the application developer can:
  - Describe **classes of objects** with their methods, data fields, relations
  - Get some of those **data members attached to DCPS entities**
  - Manipulate these objects (i.e., create, modify, delete) using the **native language constructs** that will, under the scene, activate accordingly the attached DCPS entities
  - Have those objects **refreshed transparently**
    - Means are provided to manage potential conflicts between incoming updates and application-made modifications
  - Relies on a **mapping** between DLRL objects and DCPS entities
    - **Generated** from a set of tags that annotate the DLRL model
How is the Mapping Indicated?

- Model Tags → DLRL Generator
- Model description
- Native model description
- Dedicated DLRL entities
- DCPS description
How is the Mapping Indicated (IDL PSM)?

- Model Description (IDL valuetypes)
- DLRL Generator
- Enhanced Model Description (+ implied IDL)
- Dedicated DLRL Entities (IDL)
- DCPS description
- Language mappings
- Native constructs
- Native constructs
DLRL Objects

- DLRL objects are plain objects with:
  - Data members
  - Methods

- Only the data members are relevant to data distribution; they can be:
  - Attribute -> value
  - Relation -> DLRL object
  - Plain local data members (i.e., not involved in data distribution) are also supported

- Attributes and relations can be mono/multi valued
  - List, Set or Map as collection types

- Two relations can form an association (inverse)

- DLRL classes can be organised by single inheritance
Example

Track

- x : real
- y : real
- comments [*] : string
- w : integer

Track3D

- z : real

Radar

tracks

* 0..1

a_radar
The structural mapping describes the links between DLRL objects and DCPS data

- E.g., in which 'topic' a given DLRL attribute will be put...
- Close to Object to Relational mapping
- Can handle existing DCPS models

Set on an attribute basis

- Needed for multi-valued attributes
  - collections as separate 'topics'
- Needed also:
  - to handle inheritance nicely (extension tables)
  - for better flexibility (e.g., savings in memory and partial updates)
- 1 DLRL class -> several DCPS topics

Default rules exist
Example of a DLRL Object Model

```
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 : DLRL::ObjectRoot {
  TrackMap tracks;       // associated
};
```

Example Mapped to DCPS

<table>
<thead>
<tr>
<th>Track</th>
<th>x : real</th>
<th>y : real</th>
<th>comments [*] : string</th>
<th>w : integer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Track3D</th>
<th>z : real</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Radar</th>
<th>Oid</th>
<th>a_radar</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>0..1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TRACK_TOPIC</th>
<th>Class</th>
<th>Oid</th>
<th>X</th>
<th>Y</th>
<th>Radar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track</td>
<td>1</td>
<td>100</td>
<td>200</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Track3D</td>
<td>2</td>
<td>102</td>
<td>201</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COMMENTS_TOPIC</th>
<th>Class</th>
<th>Oid</th>
<th>Index</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Track</td>
<td>1</td>
<td>0</td>
<td></td>
<td>a comment</td>
</tr>
<tr>
<td>Track</td>
<td>1</td>
<td>1</td>
<td></td>
<td>another comment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RADAR-TRACKS_TOPIC</th>
<th>R_Oid</th>
<th>index</th>
<th>T_Class</th>
<th>T_Oid</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>0</td>
<td>Track</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>Track3D</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
**Corresponding Topic Model**

### RADAR_TOPIC
- **Oid**: 11

### T3D_TOPIC
- **Oid**: 2
  - **Z**: 300

### TRACK_TOPIC
- **Class**: Track
  - **Oid**: 1
    - **X**: 100
    - **Y**: 200
    - **Radar**: 11

- **Class**: Track3D
  - **Oid**: 2
    - **X**: 102
    - **Y**: 201
    - **Radar**: 11

### COMMENTS_TOPIC
- **Class**: Track
  - **Oid**: 1
    - **Index**: 0
      - **Comments**: a comment

- **Class**: Track
  - **Oid**: 1
    - **Index**: 1
      - **Comments**: another comment

### RADAR-TRACKS_TOPIC
- **R_Oid**: 11
  - **Index**: 0
    - **T_Class**: Track
    - **T_Oid**: 1

- **R_Oid**: 11
  - **Index**: 1
    - **T_Class**: Track3D
    - **T_Oid**: 2

---

**TopicModel.idl**

```idl
struct RADAR_TOPIC {
    DLRLOid Oid; // key
};

struct TRACK_TOPIC {
    string Class; // key
    DLRLOid Oid; // key
    float X;
    float Y;
    DLRLOid Radar;
};

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;
};

struct T3D_TOPIC {
    DLRLOid Oid; // key
    float Z;
};
```
Default Mapping file

**TopicModel.idl**

```idl
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;
};
```

**ObjectModel.idl**

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

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

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

**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>
  <local name="w"/>
</classMapping>
Outline

- What is DLRL?
  - Purpose of DLRL
  - DLRL objects characteristics
  - Mapping DLRL <-> DCPS

- Main DLRL entities
  - Cache, CacheAccess, ObjectRoot, ObjectHome Collections, Relations
  - How does that work?
    - Initialisation
    - Reading objects
    - Writing objects

- Other DLRL Entities - helpers
  - Selections, Criterions, Listener
Main DLRL entities (1/2)

**Cache**
- Isolates a set of related objects that will be managed consistently
- Corresponds to one DCPS Publisher and/or one DCPS Subscriber
- An object in the Cache represents the global system state.

**CacheAccess**
- Gathers a consistent set of object copies (clones)
- Allows to perform a consistent set of modifications (Write mode) or take a consistent picture for reading (Read mode)
- An object in a CacheAccess represents either:
  - a temporary system state when in READ_ONLY mode
  - an intended system state when in WRITE_ONLY or READ_WRITE mode.

**ObjectRoot**
- Root for any DLRL object

**ObjectHome**
- Root of typed manager of a set of objects
Main DLRL entities (2/2)

**Collection**
- Root for *multi-valued* attributes or relations
- three basis
  - List
  - Map
  - Set

**Relation**
- Not a real entity (implemented as object attributes)
- Represents an 'association end'
- Two relations can be managed consistently
  - To model a real association
  - When one relation is modified, the other is changed accordingly under the scenes

*(Some DLRL entities are abstract roots to support typed derivations)*
Lifecycle state transition of Objects in a Cache

- **NEW**
  - new instance arrives
- **NOT_MODIFIED**
  - update arrives
  - end of update round
- **MODIFIED**
  - update arrives
  - end of update round
- **DELETED**
  - instance disposed
- **VOID**
  - garbage collected

**read_state**

**write_state**
Lifecycle state transition of objects in a CacheAccess

**read_state**

- **NEW**
  - refresh not updated
  - refresh updated

- **NOT_MODIFIED**
  - refresh updated
  - refresh not updated
  - update deleted OR purge OR refresh not contracted

- **MODIFIED**
  - refresh updated

- **DELETED**
  - garbage collected

**write_state**

- **NEW**
  - refresh not updated
  - refresh updated
  - update deleted OR purge OR refresh not contracted
  - write CacheAccess OR refresh not contracted

- **NOT_MODIFIED**
  - refresh not updated

- **MODIFIED**
  - update deleted OR purge OR refresh not contracted

- **DELETED**
  - garbage collected
CacheAccess in Read mode

DCPS

DLRL

Cache
CacheAccess in Read mode
CacheAccess in a writeable mode

DCPS

DLRL

Cache

CacheAccess
DLRL Initialisation

- Create a Cache (using the CacheFactory)
  - Passing a DCPS DomainParticipant and a CacheUsage as parameters
  - Creates under the scenes the DCPS Publisher and/or DCPS Subscriber
- Register the ObjectHomes to that Cache
- Call Cache::register_all_for_pubsub
  - Creates under the scenes the Topics, DataWriters and/or DataReaders
- **If needed, adjust QoS using DCPS**
  - Means are provided to retrieve the DCPS constructs
- Call Cache::enable_all_for_pubsub
  - Ready to read / write objects
Outline

- What is DLRL?
  - Purpose of DLRL
  - DLRL objects characteristics
  - Mapping DLRL <-> DCPS

- Main DLRL entities
  - Cache, CacheAccess, ObjectRoot, ObjectHome Collections, Relations
  - How does that work?
    - Initialisation
    - Reading objects
    - Writing objects

- Other DLRL Entities - helpers
  - Selections, Contracts, Listener
SelectionCriterion

- A Criterion that is applied to incoming updates to decide whether the corresponding objects should be added to the attached Selection.
  - FilterCriterion: the decision is based on a user-defined algorithm.
  - QueryCriterion: the decision is based on an SQL expression

Selection

- 'Active' sub-set of the instances of a given class
  - Resulting of the application of a SelectionCriterion
    - Manually and/or automatically refreshed
- Listeners can be attached
  - when an object enters / exits the selection
  - when a member of the selection is modified
**Listeners** - to notify incoming updates

- 3 kinds of listeners
  - CacheListener
    - When a bunch of incomings updates starts / ends
    - When the update mode is being changed.
  - ObjectListener
    - When an object is created / modified / deleted
  - SelectionListener
    - When a member enters a selection / is modified / exits the selection

- Order of triggering
  - CacheListener::on_begin_updates
  - modifications are performed in the Cache objects
  - Object and selection listeners
  - CacheListener::on_end_updates
DDS-DLRL Summary

DDS-DLRL is a layer on top of DCPS to ease the management of the data and to integrate it seamlessly in the application

- Object-orientation
- Management of graphs of objects

It promotes **typed interfaces** (as DCPS)

- By means of code-generation

**It does not force a global object model**

- The object model is local
  - better flexibility at system level
- Mapping to the DCPS data model
Thank you for your attention

This presentation has been prepared by RTI and THALES

To get a PowerPoint formatted copy, please contact the authors:

gerardo.pardo@rti.com

virginie.watine@fr.thalesgroup.com