Unifying the Global Data Space using DDS and SQL

OMG RT Embedded Systems Workshop
13 July 2006

Gerardo Pardo-Castellote, Ph.D.
CTO
gerardo.pardo@rti.com

Fernando Crespo Sanchez
fernando.crespo@rti.com

www.rti.com
Outline

• Motivation & Background
• Use cases for Global Data
• Mapping between DDS and RDBMS
• Proof-of-concept
• Conclusion
Motivation for a Unified Global Data Space

- Leverage strength of two APIs
  - SQL/ODBC:
    - Optimized data manipulation/management
  - DDS:
    - Optimized data distribution

- Expand the universe of accessible data
  - Access to existing DB-stored data
  - Access to DDS data from legacy DB-based applications

- Exploit complementary capabilities
  - Data in flight (QoS, latency, notifications, real-time)
  - Data storage (persistence, indexing, unlimited capacity)

- Save time and money!
  - Get power & flexibility using standard API's
Background

- What is DDS?
- What is a RDBMS?
- Do we need both in the same system?
- Alternative approaches
DDS Enables Global Data

Address in Global Data Space = (DomainId, Topic, Key)
- Each topic corresponds to a 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 read from multiple instances of a single topic
- Multiple DataReaders may read from the same instances
DDS elements

- Metamodel: Data Distribution PIM
  - Concept of Global Data Space
  - Domain, Topic, Key,
  - Concept of Entity, DomainParticipant, DataReader, DataWriter,
  - Concept of Entity Listener, QoS, Conditions, WaitSets

- Abstract API: DDS PIM
  - Classes on the DDS PSM
  - Operations on the DDS Entities and semantic meaning
  - Description of the available QoS and related behavior

- Concrete programming APIs (C++, Java, etc.)
  - Actual programming API’s
DDS Example (boxes demo)

- **Topics**
  - Square, Circle, Triangle
    - all of type Shape
  - Attributes
    - of type Attributes

- **IDL data types:**

  ```
  struct Shape {
    string<16> color; // key
    int x;
    int y;
    int size;
  };

  struct Attributes {
    string<16> shape; // key
    string<16> color; // key
    float speed;
  }
  ```
RDBMS enables persistence and manipulation of data

Address in a RDBMS data space

= (Database, Table, PrimaryKey)

- Each Table stores multiple records, all with the same schema (data-type)
- Each record represents a unique data element
- Records are identified by a primary key
- Applications can read and modify records or groups of records
  - SQL language provides means to address groups of records
- New tables can be ‘materializes’ by combining records from different tables
RDBMS Elements

- **Metamodel: Relational Model**
  - Tables, Records, Primary Keys, Foreign Keys
  - Views

- **Abstract API: SQL**
  - SQL schema
  - SQL statements SELECT, INSERT, UPDATE
  - SQL clauses (WHERE, ORDERED BY, etc.)

- **Concrete programming APIs**
  - ODBC (C API)
  - JDBC (Java API)
  - other e.g. Oracle’s PL/SQL (C API)
**RDBMS Example (boxes demo)**

### “Squares” Table

<table>
<thead>
<tr>
<th>color</th>
<th>x</th>
<th>y</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>“red”</td>
<td>14</td>
<td>23</td>
<td>70</td>
</tr>
<tr>
<td>“blue”</td>
<td>200</td>
<td>67</td>
<td>50</td>
</tr>
</tbody>
</table>

### “Circles” Table

<table>
<thead>
<tr>
<th>color</th>
<th>x</th>
<th>y</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>“red”</td>
<td>43</td>
<td>22</td>
<td>50</td>
</tr>
<tr>
<td>“black”</td>
<td>132</td>
<td>66</td>
<td>100</td>
</tr>
</tbody>
</table>

### “Attributes” Table

<table>
<thead>
<tr>
<th>shape</th>
<th>color</th>
<th>size</th>
<th>speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>“square”</td>
<td>“red”</td>
<td>70</td>
<td>5.6</td>
</tr>
<tr>
<td>“square”</td>
<td>“blue”</td>
<td>50</td>
<td>22.45</td>
</tr>
<tr>
<td>“circle”</td>
<td>“red”</td>
<td>70</td>
<td>101.3</td>
</tr>
<tr>
<td>“circle”</td>
<td>“black”</td>
<td>100</td>
<td>45.4</td>
</tr>
</tbody>
</table>

(c) 2006 Real-Time Innovations, Inc
Do I need both in a system?

Yes. They both have complementary capabilities!

- DDS is good for:
  - Data “in-flight”: sending & receiving data
  - High performance communication and notifications
  - Access to real-time nodes
  - QoS and resource control

- RDBMS/SQL is good for
  - Storing and persisting data
  - Data analysis, correlations, fusion
  - Data organization
  - Integration with ODBC clients (GUIs, web-services)

... most complex systems are already doing it...
How to use DDS and RDBMS?

- Until now:
  - Two disconnected data-spaces
  - Custom mapping of data-models
  - Custom application-level bridging
A better way:
Standards-Based Global Data Space

All data accessible to all interested applications using either API

- Single, unified, data-space
- Transparent mapping of data-models
- No need for application-level bridging
Outline

- Motivation & Background
- Use cases for Global Data
- Mapping between DDS and RDBMS
- Proof-of-concept
- Conclusion
Use cases: DDS to DDS

● DDS write to DDS read:
  – Real-Time data distribution
  – QoS aware communications
  – High performance messaging and events
  – Publish-subscribe communications
Use cases: DDS write to SQL read

- DDS write to SQL read:
  - Logging
    - Store all historic data values
  - Data analysis/Data mining
    - Look at trends, correlate data from multiple topics
    - Create custom data-views
  - Integration to ODBC analysis/monitoring clients
    - Use web-clients, excel, exiting GUIs to see DDS data
Use cases: SQL write to DDS read

- SQL write to DDS read:
  - Real-Time distributed database data monitoring
    - Get notified when any record is modified
    - Get notification anywhere, even RT nodes
  - Changes to data-sets
    - E.g. Change departure time of all aircraft leaving after 5pm and delay one hour
Use cases: SQL to SQL

- DDS write to SQL read:
  - Classic data-base applications
  - Database replication full or partial
    - Selected table replication
    - Mix of heterogeneous (multi-vendor) databases
    - Mix of in-memory and disk-based databases
Outline

- Motivation & Background
- Use cases for Global Data
- Mapping between DDS and RDBMS
- Proof-of-concept
- Conclusion
RDBMS Elements

- **Metamodel: Relational Model**
  - Tables, Records, Primary Keys, Foreign Keys
  - Views

- **Abstract API: SQL**
  - SQL schema
  - SQL statements SELECT, INSERT, UPDATE
  - SQL clauses (WHERE, ORDERED BY, etc.)

- **Concrete programming APIs**
  - ODBC (C API)
  - JDBC (Java API)
  - other e.g. Oracle’s PL/SQL (C API)
DDS ⇔ RDBMS mapping at the metamodel and abstract levels

**DDS**
- Metamodel: Data Distribution PIM
  - Global Data Space
  - Domain, Topic, Key,
- Abstract API: DDS PIM
  - Classes and operations on the DDS PSM
  - QoS
- Concrete programming APIs

**RDBMS**
- Metamodel: Relational Model
  - Tables, Records,
  - Primary Keys, Foreign Keys
- Abstract API: SQL
  - SQL schema
  - SQL statements
- Concrete programming APIs
## Mapping (behavior)

<table>
<thead>
<tr>
<th>Data Distribution Service (DDS)</th>
<th>Database Management System (DBMS)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL Type</strong></td>
<td>Table Schema</td>
</tr>
<tr>
<td><strong>Topic</strong></td>
<td>Table</td>
</tr>
<tr>
<td><strong>Data-object Instance</strong></td>
<td>A single <strong>Row</strong> in a Table if no history is stored; or <strong>Multiple Rows</strong> in a Table if history is stored.</td>
</tr>
<tr>
<td><strong>Instance Key</strong></td>
<td><strong>Primary Key</strong> in Table</td>
</tr>
<tr>
<td><em><em>DataWriter::write</em>() [all variations]</em>*</td>
<td><strong>INSERT</strong> or <strong>UPDATE</strong></td>
</tr>
<tr>
<td><strong>DataWriter::dispose()</strong></td>
<td><strong>DELETE</strong></td>
</tr>
</tbody>
</table>
Mapping (data model)

- **IDL struct** → flattened out into a table schema
  - column name → full hierarchical member name
  - field type → mapping of IDL primitive type to corresponding SQL type
- **IDL union** → same as struct with additional discriminator column
- **IDL sequence** → several possible mappings
  - In-line
    - length + nullable max-length columns [bounded only]
  - Extra table
    - Can be shared per sequence type or dedicated
  - Octet sequence → SQL binary
Mapping (data model)

- IDL array ➔ in-line expansion
  - Octet array ➔ SQL binary
- IDL valuetype ➔ two options
  - Performance-oriented:
    - Same as struct (including base valuetype fields)
  - Relational-model oriented
    - Expansion tables as in the DLRL mapping
### Almost 1-1 mapping of primitive types

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>IDL Field Name</th>
<th>SQL Type</th>
<th>Table Field Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR</td>
<td>my_field</td>
<td>CHAR(1)</td>
<td>“my_field”</td>
</tr>
<tr>
<td>WCHAR</td>
<td>my_field</td>
<td>WCHAR</td>
<td>“my_field”</td>
</tr>
<tr>
<td>OCTET</td>
<td>my_field</td>
<td>BINARY(1)</td>
<td>“my_field”</td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>my_field</td>
<td>TINYINT</td>
<td>“my_field”</td>
</tr>
<tr>
<td>SHORT</td>
<td>my_field</td>
<td>SMALLINT</td>
<td>“my_field”</td>
</tr>
<tr>
<td>UNSIGNED SHORT</td>
<td>my_field</td>
<td>SMALLINT</td>
<td>“my_field”</td>
</tr>
<tr>
<td>LONG</td>
<td>my_field</td>
<td>INTEGER</td>
<td>“my_field”</td>
</tr>
<tr>
<td>UNSIGNED LONG</td>
<td>my_field</td>
<td>INTEGER</td>
<td>“my_field”</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>my_field</td>
<td>DOUBLE</td>
<td>“my_field”</td>
</tr>
<tr>
<td>FLOAT</td>
<td>my_field</td>
<td>REAL</td>
<td>“my_field”</td>
</tr>
<tr>
<td>STRING&lt;LENGTH&gt;</td>
<td>my_field</td>
<td>VARCHAR(LENGTH)</td>
<td>“my_field”</td>
</tr>
<tr>
<td>WSTRING&lt;LENGTH&gt;</td>
<td>my_field</td>
<td>WVARCHAR(LENGTH)</td>
<td>“my_field”</td>
</tr>
<tr>
<td>LONG,LONG</td>
<td>my_field</td>
<td>BIGINT</td>
<td>“my_field”</td>
</tr>
<tr>
<td>UNSIGNED_LONG_LONG</td>
<td>my_field</td>
<td>BIGINT</td>
<td>“my_field”</td>
</tr>
<tr>
<td>LONG DOUBLE</td>
<td>my_field</td>
<td>BINARY(128)</td>
<td>“my_field”</td>
</tr>
</tbody>
</table>
Mapping examples

IDL Type

```c
struct MyStruct {
    long my_key_field; //@key
    short my_short_field;
};
```

```c
struct MyStructContainer {
    long my_key_field; //@key
    MyStruct my_struct_field;
};
```

```c
struct MySequenceContainer {
    long my_key_field; //@key
    sequence<long,2> my_seq_field;
};
```

SQL Table Schema

```sql
Create Table "MyStruct" (  
    "my_key_field" INTEGER NOT NULL,  
    "my_short_field" SMALLINT NOT NULL,  
    PRIMARY KEY("my_key_field")
);
```

```sql
Create Table "MyStructContainer" (  
    "my_key_field" INTEGER NOT NULL,  
    "my_struct_field.my_key_field" INTEGER NOT NULL,  
    "my_struct_field.my_short_field" SMALLINT NOT NULL,  
    PRIMARY KEY("my_key_field")
);
```

```sql
Create Table "MySequenceContainer" (  
    "my_key_field" INTEGER NOT NULL,  
    "my_seq_field#length" INTEGER NOT NULL,  
    "my_seq_field[0]" INTEGER,  
    "my_seq_field[1]" INTEGER,  
    PRIMARY KEY("my_key_field")
);
```
Mapping extends to the wire representation

```cpp
struct MyType
{
    int i;
    double x;
    string<100> s;
};
```

**DDS Type**
- **DDS-DBMS Auto-Generated Code**
  - (UserDataReader, UserDataWriter)

**DBMS Table**
- **MyType**
- **Slot #**

<table>
<thead>
<tr>
<th></th>
<th>i</th>
<th>x</th>
<th>s</th>
<th>Slot #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IN</td>
<td>DO</td>
<td>VARCH</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
<td>(100)</td>
<td></td>
</tr>
</tbody>
</table>

**DDS-DBMS**
- Generic type serialization/de-serialization using the Table Schema
  - (BridgeDataWriter, BridgeDataReader)

**Serialized Wire Format Representation on the “Network”**
Configuration

- Table name ➔ DDS topic_name + domain_id
  - Can allow multiplexing:
    - Topics of same type can be mapped into a single table
    - A table can be published into multiple topics

- Partial/custom mappings
  - Unmapped Topics/Tables
  - Unmapped fields
    - DBMS only fields
    - DDS only fields
  - Custom mappings

- DDS QoS

- Database history
Configuration approaches

- Configuration file (static)
- Configuration tables (static or dynamic)
  - Publication, Subscription tables
  - Can use SQL or DDS to configure
- Other?
  - command-line
  - dedicated API e.g. web-service
Outline

- Motivation & Background
- Use cases for Global Data
- Mapping between DDS and RDBMS
- Proof-of-concept
- Conclusion
Proof of concept
Conclusion

The integration of relational databases and DDS:

- Can be formally defined
- Is practical and feasible
- ... and offers powerful benefits:
  - Leverages strength of two API standards
  - Expands the universe of accessible data
  - Exploits complementary capabilities
  - Saves time and money!
Thank you

Gerardo Pardo-Castellote, Ph.D.  
gerardo.pardo@rti.com

Fernando Crespo Sanchez 
fernando.crespo@rti.com

www.rti.com