Towards a publish/subscribe control architecture for precision assembly with the Data Distribution Service

Marco Ryll[1] and Svetan Ratchev[1]

1. School of Mechanical, Materials and Manufacturing Engineering, The University of Nottingham; email: [epxmr3 | svetan.ratchev] @nottingham.ac.uk.

Abstract. This paper presents a comprehensive overview of the Data Distribution Service standard (DDS) and describes its benefits for developing robust precision assembly applications. DDS is a platform-independent standard released by the Object Management Group (OMG) for data-centric publish-subscribe systems. It allows decoupled applications to transfer information, regardless of what architecture, programming language or operating system they use. The standard is particularly designed for real-time systems that need to control timing and memory resources, have low latency and high robustness requirements. As such, DDS has the potential to provide the communication infrastructure for next-generation precision assembly systems where a large number of independently controlled components need to communicate. To illustrate the benefits of DDS for precision assembly an example application is presented.

1. Introduction

Current trends in manufacturing such as modularisation of processes require modern assembly systems which integrate a large number of subsystems such as assembly stations, fixturing devices, Human Machine Control (HMI) etc. This leads to increased information exchange, together with ever more demanding requirements for flexibility and reconfigurability.

Due to these challenges, distributed industrial control systems have attracted a vast amount of research interest. Campelo et al. [1] have proposed a distributed control architecture for manufacturing systems which addresses the problem of fault-tolerance to ensure recovery when components fail. Other examples for fault-tolerant, distributed architectures with real-time characteristics include the DACAPO system [2, 3] and DELTA-4 [4]. Delamer et al. [5-7] have described the CAMX framework (Computer-Aided Manufacturing using XML) which is a message-oriented middleware providing event-distribution in form of standardized

XML-messages with a publish-subscribe approach. Another message-oriented publish-subscribe middleware is the Java Messaging Service (JMS). Although, these systems allow robust communication between many nodes, it appears that message-oriented communication consumes larger amounts of the processing time for the translation of the messages in the nodes. In particular, JMS is limited to the Java programming language and therefore lacks platform-independence. Moreover, JMS does not support dynamic discovery of new components in plug & produce manner, since application discovery is administered and centralized [8].

With the Data Distribution Service (DDS) [9], the Object Management Group (OMG) has recently released a platform-independent standard for a data-centric publish-subscribe middleware that is specifically targeted towards the needs of real-time applications with limited resources. Unlike the previously mentioned approaches, DDS does not exchange data encapsulated within messages. Instead, the data structures to be exchanged are modelled with the formal Interface Definition Language (IDL). On the basis of this data definition all source code to accurately transmit and receive information is generated automatically. This way, information exchange can be achieved significantly faster.

This paper aims at providing a comprehensive overview on the DDS standard and its potential benefits for the precision assembly domain. In the first section, the basic concept of a DDS-based application is described and an architectural overview of the entities that establish data transmission is provided. This is followed by a discussion of three features that can facilitate the development of robust precision assembly platforms. In the last part of the paper, a simple example application is described to illustrate some of the advantages of the DDS middleware.

2. Data Distribution Standard

DDS provides common application-level interfaces which allow processes (so-called participants) to exchange information in form of topics. The latter are data-flows which have an identifier and a data type. A typical architecture of a DDS application is illustrated in figure 1. Applications that want to write data declare their intent to become “publishers” for a topic [9]. Similarly, applications that want to read data from a topic declare their intent to become “subscribers”. Underneath, the DDS middleware is responsible to distribute the information between a variable number of publishers and subscribers. It manages the declarations, automatically establishes connections between publishers and subscribers for a matching topic and dynamically detects new participants in the system. Additionally, DDS allows to precisely configure the properties of the information exchange with the Quality-of-Service concept (QoS) which is particularly important for real-time systems with limited resources and to overcome the “impedance mismatch” between systems with different communication requirements.

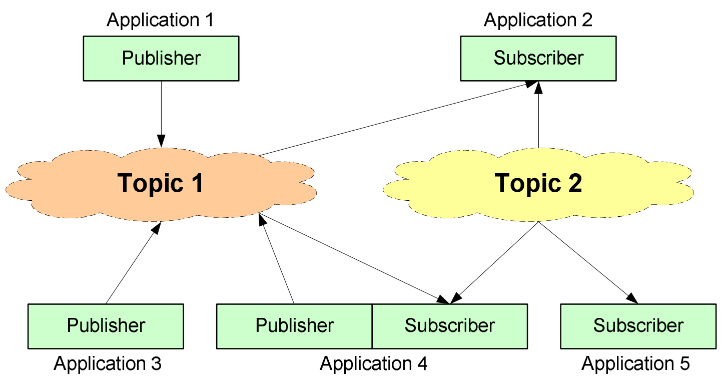

Fig. 1. Decoupling of publishers and subscribers with the DDS middleware

As a result, the utilisation of DDS reduces the complexity of data distribution to simple calls of API-functions which are provided by the standard. This saves development time and is less error-prone than the development of own proprietary data exchange infrastructures. Also, the utilisation of a platform-independent standard allows better integration with other applications running on different architectures. The middleware hides all details like location of the participants, type of the network or what programming language has been used to implement the participants. Since publishers and subscribers are only connected to topics, communication between them is decoupled. This allows the establishment of dynamic communication channels without reconstructing the source code.

2.1. Architectural Overview

The core of the DDS specification is the Data Centric Publish Subscribe model (DCPS) which organises the data exchange between the communicating participants. Figure 2 presents a simplified view of this model. As it can be seen in the diagram all classes that accomplish communication are extended from the central class *Entity*. This class provides the ability to be configured with Quality-of-Service parameters, be notified of events by listener objects, and be attached with conditions that can be waited upon by the application. All child classes of *Entity* have a specialised set of *QoSPolicies* which provides the ability to fine-tune the data exchange.

The publishing side of the communication is represented by the association between a *Publisher* and one or more *DataWriter* objects. The *Publisher* is internally used by the DDS middleware to issue data to be sent. A *DataWriter* acts as a typed accessor to the *Publisher* for the application and is specific for the data type to be sent. The application must use this object if it wants to send data of a given type, which then triggers the *Publisher* to issue the data according to the Quality-of-Service settings. The subscribing end of the communication is similarly structured. The *Subscriber* is internally responsible for receiving published data and making it available according to the QoS-settings. The application can access the received data by *DataReader*-objects generated for each data type.

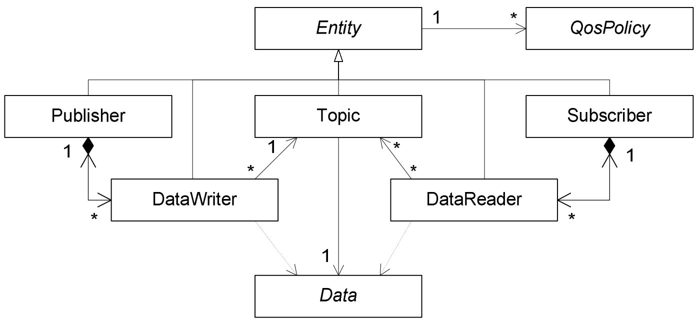

Fig. 2. Overview of the DCPS model (based on [9])

The association between publications and subscriptions is accomplished by means of *Topic* objects. A *Topic* associates a unique name, a data type and Quality-of-Service settings related to the data itself. DDS middleware implementations provide tools that automatically generate the code for these classes on the bases of the data type definitions. The application must use these classes and call the provided methods in order to achieve communication.

2.2. Relevant features for the development of precision assembly platforms

As mentioned before, precision assembly platforms are increasingly implemented as dynamic distributed systems where a number of components need to exchange data in a robust, platform-independent and deterministic way. According to Joshi [10] the key technical challenges for the development of such systems are: (1) impedance mismatch; (2) dynamic real-time adaptation and (3) incremental and independent development. The first challenge refers to the problems with the integration of applications that impose different requirements for the data exchange, such as data volume, data rates or timing constraints. Dynamic real-time adaptation addresses the need to discover topology changes as components are added or removed from the platform. The third challenge refers to the fact that the various subsystems of an assembly platform are often developed by independent parties which upgrade their components incrementally whilst utilizing different operating systems and programming languages. The following sections give an overview on aspects of DDS that can reduce the impact of these challenges.

2.2.1. The Quality-of-Service Concept

With the Quality-of-Service concept, DDS provides a sophisticated means to dynamically tailor data distribution according to the application requirements. In particular, QoS provide the ability to control and limit the use of resources like network bandwidth or memory as well as reliability, timeliness and persistence of the data transfer.

The DDS QoS model is a set of classes which are derived from *QosPolicy*. Each of them controls a specific aspect of the behaviour of the service and as shown in figure 2 they can be attached to all objects that accomplish communication. The middleware automatically checks if the settings for the publishing and subscribing side are compatible. Communication is only established if the offered communication properties of the publisher meet the requested behaviour of the subscriber. The complete specification can be found in [9].

The QoS concept solves a number of typical problems in the development of distributed applications. For example, with the TIME_BASED_FILTER QoS a subscriber can precisely define how often it needs to be updated with new data. This prevents it from flooding when it is integrated with systems that produce data at much higher rates (impedance mismatch). The combination of OWNERSHIP and OWNERSHIP_STRENGTH provides an easy way of integrating redundancy and a seamless failover to backup systems. The HISTORY QoS alleviates the challenge of providing late-joining processes with already sent (so-called historical) data. These features are demonstrated in section 3.2.

2.2.2. Automatic Discovery and dynamic real-time adaptation

In order to react to changing product or process requirements it must be possible to dynamically add or remove components without the need of rebuilding the control software. For example, an assembly line might be extended by modules with additional functionalities or monitoring devices are plugged and unplugged during the production process without affecting the overall process.

DDS addresses that aspect with its decoupled publish-subscribe architecture approach and the automatic discovery of participants. Since publishers and subscribers are only connected to topics, communication between them is decoupled. This allows the establishment of dynamic communication channels without reconstructing the source code. New publishers and subscribers for a topic can appear at any time and the DDS middleware interconnects them automatically.

Similarly, the middleware provides mechanisms to inform the application when participants have been removed.

2.2.3. Platform-independence and independent development

The components of modern manufacturing systems are often developed by different vendors. As a consequence, these subsystems might be implemented on the basis of different hardware architectures, operating systems and programming languages. This increases the challenge of integrating them into a working system. Also, each component might be subject to incremental changes or upgrades.

DDS is defined by a Platform Independent Model (PIM) and thus can be implemented for any combination of processor architecture, programming language and operating system. Commercially available DDS middleware implementations offer solutions for the programming languages C, C++ and Java and a wide variety of operating systems such as VxWorks, Windows, Lynx and Unix derivates. Since DDS hides the communication aspects from the application code it allows the seamless integration of an application written in C running on VxWorks with an application developed with Java running on a Windows PC. Hence, the usage of DDS as a backbone for the communication within the assembly system can significantly reduce the difficulties of the system integration task. Additionally, the data-centric publish-subscribe paradigm introduces less dependencies between the applications than conventional object-oriented or client-server approaches [10]. This is because in a data-centric architecture, applications are only connected by the data model and do not expose behaviour. Since the data model is usually the most constant aspect of an application, DDS supports the incremental and independent development of subsystems.

3. Example Application

This section aims to illustrate some of the previously mentioned features of DDS with a simplified example application. It was developed using the commercially available DDS implementation from Real-Time Innovations, Inc., called RTI DDS 4.1e. The example uses the DDS standard for the implementation of a sensor-based active fixturing system. A simplified overview of the proposed system is shown in figure 3.

The system consists of a variable number of physical fixture modules, a fixture control software, a variable number of Human Machine Interfaces (HMI). For the sake of simplicity, each module consists of one linear actuator and three sensors. The former acts as the locating and clamping pin against the workpiece, while the sensors feedback reaction force, position and temperature of the contact point. The fixture modules are implemented as smart devices with local control routines for their embedded sensor/actuator devices. It is further assumed that each fixture module is configured with a unique numerical identifier and with meta information about its sensors and actuators. This way, the module is able to convert the signals coming from the sensors (e.g. a voltage) into meaningful information (e.g. reaction force in Newton) which is then published via DDS. The fixture control implements the global control routines of the fixture. It processes the data coming from the various modules and controls the movement of the actuators by publishing their desired status.

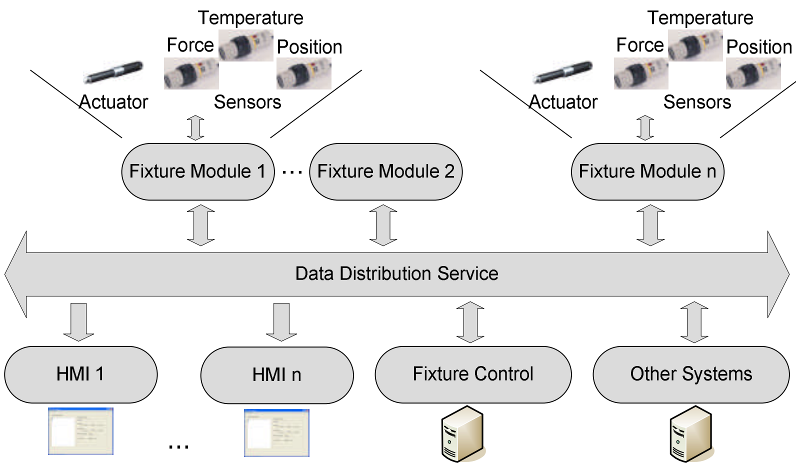

Fig. 3. Overview on the example application

In order to demonstrate some features of DDS we define the following requirements for the overall system:

1. At the initial start-up each fixture module publishes meta information about its sensors and actuators in order to allow subscribing applications to interpret the sensor data. Late-joining applications shall receive this information automatically.
2. HMI applications shall receive force sensor readings every 500ms and temperature readings every 1000ms, regardless how fast the modules publish this information.
3. The fixture control and connected HMI applications only consider the temperature readings of 1 fixture module at a time. If no data from this most-trusted sensor source is received within 4000ms, temperature data shall automatically be received from another module, allowing a seamless failover.

Details about the control logic of this application are not described within this paper. Instead the description concentrates on the data modelling and the definition of the QoS-settings to meet the requirements.

3.1. Design of Data Structures

The development of a data-centric application starts with the definition of the data structures that shall be exchanged between the applications. In our simple example, we create two data structures for each sensor type. The first data structure contains the actual sensor data to be transferred during the assembly process. It contains fields to uniquely identify the sensor and a field for the current sensor reading. In order to interpret the sensor values correctly, subscribers need additional meta-information. These details are modelled in an additional structure and only need to be published when the application starts or if sensors are exchanged. This way, sensor data and its interpretation are separated and network load can be reduced significantly. Subscribing applications can be dynamically reconfigured to accurately interpret incoming data from different sensors. As an example, the data structures for force sensors are shown below according to the Interface Definition Language (IDL). The structure *Force* is used to transmit the sensor readings, whereas *ForceSensorInfo* contains meta-information to interpret the sensor data correctly. For the other sensor types the data structures are defined accordingly.

```
struct Force{
    long module_id;
    lond device_id;
    double value;
};
```

```
struct ForceSensorInfo{
      long module_id;
      long device_id;
      MeasuringUnit unit;
      double maxForce;
      double minForce;
      double resolution;
      double sampleRate;
};
```

Based on these data type definitions, the source code for all the subscribers, publishers, data readers and data writers are automatically generated in the specified programming language. These classes have to be used by the application programs that implement the fixture modules, the fixture control and any other participating system like monitoring applications.

3.2. Selection of Quality-of-Service parameters

To tailor the data transfer according to the requirements, the QoS parameters for each data reader and data writer need to be configured in the application source code. Table 1 shows for each requirement the QoS settings for the data readers on the subscribing side and data writers for publishing applications.

When a fixture module executes its initializing routine it retrieves the meta-data for each sensor from a configuration file and creates the corresponding data structures. The data writers for these data types are configured according to table 1 and each fixture module publishes this information only once. After this publication it can initialize the publishers for the actual sensor data transmission and start publishing the sensor reading. Subscribing applications like HMIs or the fixture control first initialize the subscribers for the meta information and hence retrieve these data before they start subscribing to the actual sensor data. Since the QoS parameters are set to HISTORY.depth = 1 and DURABILITY.kind = TRANSIENT_LOCAL it is assured by the middleware that even late-joining applications will get the necessary information to interpret all sensor readings. This approach significantly saves network resources, since meta information is only transferred when it is necessary. DDS allows the implementation of this feature with only minor programming effort, whereas in traditional distributed systems providing late-joining applications with historical data is an error prone and complex task. The same argument applies for the second requirement. Monitoring applications can limit the number of sample readings simply by setting the time-based filter QoS on their data readers. This way they are protected from being flooded with too much data and can spent more resources for the graphical user interface. In traditional client-server based applications this “impedance mismatch” is a major problem. DDS overcomes this problem by the simple setting a QoS parameters.

Requirement	**Data type**	**QoS DataReader**	**QoS DataWriter**
1	ForceSensorInfo, TemperatureSensor Info, DisplacementSensorInfo	HISTORY.depth = 1 RELIABILITY.kind = RELIABLE DURABILITY.kind = TRANSIENT_LOCAL	HISTORY.depth = 1 DURABILITY.kind = TRANSIENT_LOCAL

2	Force
	
	Temperature
		TIME_BASED_FILTER = 500ms
		TIME_BASED_FILTER = 1000ms
			-
			
			-

3	Temperature	OWNERSHIP.kind = EXCLUSIVE DEADLINE.period.sec = 4	OWNERSHIP.kind = EXCLUSIVE OWNERSHIP_STRENGTH. value = module_id of the fixture module DEADLINE.period.sec = 4

Table 1. Qos Settings for the Applications

The third requirement allows subscribing applications to get temperature readings only from one most-trusted sensor. If this sensor stops working because of damage or other reasons, the applications shall automatically use the readings from a temperature sensor of another fixture module. This automatic and dynamic failover to a backup sensor would require enormous programming effort if implemented manually. With DDS we have to set the OWNERSHIP.kind parameter to “exclusive” to ensure that readers will only receive data from a single sensor. Additionally, each temperature data writer is configured with the identifier of its corresponding fixture module, resulting in a hierarchic order. The data readers in each subscribing application will only receive temperature readings from the fixture module with the highest identifier. In this context, the DEADLINE QoS specifies that the subscribers will automatically failover to the sensor of the fixture module with second-highest ID if it does not receive data within the specified time period. This way, fault-tolerant distributed applications can easily be developed with the ability to dynamically react to failures in the system.

4. CONCLUSIONS

In this paper, a new standard for data-centric publish-subscribe communication has been presented and put in context to the development of next-generation precision assembly platforms. The standard is called Data Distribution Service and is particularly targeting real-time applications which need to manage resource consumption and timeliness of the data transfer. DDS allows platform-independent many-to-many communication and alleviates a number of common problems which are of particular interest for the development of distributed assembly systems. For example, with its sophisticated Quality-of-Service support communication can be tailored according to the system requirements and typical challenges such as the delivery of historical data to late-joining applications is achieved automatically and in an efficient manner. Additionally, DDS automatically discovers when components are plugged in or removed. The example presented in this paper shows how these features can potentially be integrated to develop a plug & produce capable active fixturing system for precision assembly.

Acknowledgments The reported research is conducted as part of the ongoing European Commission FP6 funded integrated project (FP6-2004-NMP-NI-4) - AFFIX “Aligning, Holding and Fixing Flexible and Difficult to Handle Components”. The financial support of the EU and the contributions by the project consortium members are gratefully acknowledged. The authors are also grateful to Real-Time Innovations, Inc. for their support of the research with a software grant for the RTI Data Distribution Service middleware.

References

1. J.C. Campelo, et al., Distributed industrial control systems: a fault-tolerant architecture, Microprocessors and microsystems, Vol. 23 (1999), 103-112.
2. B. Rostamzadeh, et al., *DACAPO: a distributed computer architecture for safety-critical control applications*,IEEE International Symposium on Intelligent Vehicles, Detroit, USA, 1995
3. B. Rostamzadeh and J. Torin, *Design principles of fail-operation/fail-silent modular node in DACAPO*,Proceedings of the ICEE, Tehran, Iran, 1995
4. J. Arlat, et al., *Experimental evaluation of the fault tolerance of an atomic multicast system*, IEEE Transactions on reliability, Vol. 39 (1990).
5. I.M. Delamer and J.L. Martinez Lastra, *Evolutionary multi-objective optimization of QoS-Aware Publish/Subscribe Middleware in electronics production*, Engineering Applications of Artificial Intelligence, Vol. 19 (2006), 593-697.
6. I.M. Delamer and J.L. Martinez Lastra, *Quality of service for CAMX middleware*, International Journal of Computer Integrated Manufacturing, Vol. 19 (2006), pp. 784-804.
7. I.M. Delamer, J.L. Martinez Lastra, R. Tuokko, *Design of QoS-aware framework for industrial CAMX systems*,Proceedings of the Second IEEE International Conference on Industrial Informatics INDIN 2004, Berlin, Germany, 2004
8. J. Joshi, *A comparison and mapping of Data Distribution Service (DDS) and Java Messaging Service (JMS)*, Real-Time Innovations, Inc., Whitepaper, 2006, Available from: www.rti.com.
9. Object Management Group, *Data Distribution Service for Real-Time Systems, Version 1.2*, 2007, Available from: www.omg.org, June 2007
10. J. Joshi, *Data-Oriented Architecture*, Real-Time Innovations, Inc., Whitepaper, 2007, Available from: www.rti.com.