International Journal of Mechanical, Industrial and Aerospace Sciences
ISSN: 2517-9950
Vol:2, No:5, 2008

Application of the Data Distribution Service for Flexible Manufacturing Automation

Marco Ryll, and Svetan Ratchev

***Abstract*—**This paper discusses the applicability of the Data Distribution Service (DDS) for the development of automated and modular manufacturing systems which require a flexible and robust communication infrastructure. DDS is an emergent standard for data-centric publish/subscribe middleware systems that provides an infrastructure for platform-independent many-to-many communication. It particularly addresses the needs of real-time systems that require deterministic data transfer, have low memory footprints and high robustness requirements. After an overview of the standard, several aspects of DDS are related to current challenges for the development of modern manufacturing systems with distributed architectures. Finally, an example application is presented based on a modular active fixturing system to illustrate the described aspects.

***Keywords*—**Flexible Manufacturing, Publish/Subscribe, Plug & Produce.

I. Introduction

DURING the past decades, the manufacturing sector has faced challenges, such as product customisation, shorter product life cycles and increased quality requirements due to the fierce competition. To address these problems and to meet the reconfiguration requirements, a trend has developed towards modularisation and automation of manufacturing systems. This, however, leads to an increased information exchange between subsystems which can have very different requirements for the data exchange. In particular, communication is not only taking place horizontally among the components on the shop floor. Production systems also need to be integrated in vertical direction with other enterprise subsystems, such as administrative or warehousing systems. As a result, flexible and open communication infrastructures are required that allow for the integration of a large number of different processes like assembly modules, fixturing systems, Human Machine Interfaces (HMI) and other parts of the factory.

To address these challenges, distributed automation concepts have been subject to extensive research effort. Nakatani et al. [1] have developed a distributed data acquisition and device control system for experiment instruments that is based on a modular structure and communication via a client/server architecture. Mitschang [2] has presented an approach for a shared information space that facilitates collaboration and data exchange in a heterogeneous application environment over multiple communication protocols. The problem of fault-tolerance and robustness of manufacturing systems has been addressed by Campelo et al. [3] with a distributed control architecture which ensure sautomatic recovery when components fail. Other examples for fault-tolerant, distributed architectures with real-time characteristics include the DACAPO system [4], [5] and DELTA-4 [6]. To alleviate the complexity of achieving flexible and deterministic communication, middleware technologies are evolving as an additional layer between the application and the operating system. In this context, Delamer et al. [7]-[9] have described a message-oriented framework for production systems that enables data exchange among machines and control software applications through the publish/subscribe paradigm. In this system, messages and events are defined in XML-format and distributed through clusters of message broker systems. Thus, the concept allows robust communication between an arbitrary number of processes. However, message-oriented communication requires additional processing time in each node for the interpretation of message. Further, the concept of message brokers introduces centralised entities which are potential performance bottlenecks and failure points.

Recently the Object Management Group (OMG) has released the Data Distribution Service (DDS) [10] as a platform-independent standard for data centric publish/subscribe middleware systems. The standard ensures deterministic information exchange and allows the managing of many aspects of the communication behaviour to meet application requirements. As opposed to message-oriented approaches, DDS does not exchange data in the form of messages. In DDS data is formally defined in a platform-independent way which is the basis for the automatic generation of communication source code for specific target platforms. This way, the standard realises communication significantly faster.

This paper provides an overview on the DDS standard and its applicability for automated manufacturing systems. In the next section, an architectural overview of the middleware is provided. This is followed by a discussion of relevant features

M. Ryll is a Research Postgraduate at the Precision Manufacturing Centre, School of Mechanical, Materials and Manufacturing Engineering, University of Nottingham, NG7 2RD UK (phone: +44(0)1158466083; fax: +44(0)1159513800; e-mail: epxmr3@ nottingham.ac.uk).

S. Ratchev, is Professor at the School of Mechanical, Materials and Manufacturing Engineering and Head of the Precision Manufacturing Centre, University of Nottingham, Nottingham, NG7 2RD UK (e-mail: svetan.ratchev@ nottingham.ac.uk).

that have the potential to support the development of robust, yet flexible manufacturing systems. Finally, the development of a software-based prototype application is presented to illustrate some of the aspects of the DDS middleware. This is based on a modular active fixturing system.

II. Overview on the Data Distribution Service

DDS provides easy-to-use communication services which allow participants to share data through topics. The latter form a so-called "global data space" that is accessible to all interested applications [10]. Processes that want to send data declare their intent to become "publishers" for a topic. Similarly, applications can declare their intent to become "subscribers" for a topic if they require data from it. Underneath, the middleware automatically detects new participants in the system and establishes connections between the publishers and subscribers for a matching topic. Fig. 1 illustrates the global data space with three topics and five participants. The arrow directions indicate if an application is a publisher or a subscriber for a certain topic. Specifically, an ingoing arrow marks the application as a subscriber while an outgoing arrow declares it as a publisher.

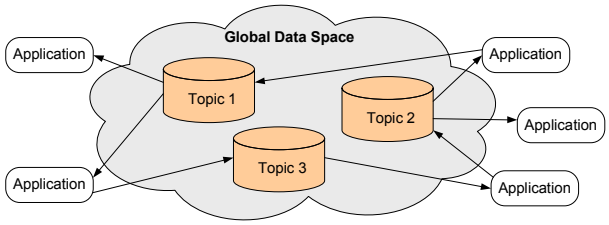

Fig. 1 Overview of the publish/subscribe concept

As a result of the publish/subscribe concept, communication is decoupled through the topics and flexible many-to-many communication between a large number of participants is supported. Moreover, the interfaces offered by DDS hide all details of the communication from the application source code of the participants, thereby significantly reducing their complexity. More specifically, utilising ready-made services saves development time and is less error-prone than developing proprietary communication infrastructures.

A. The Data Centric Publish/Subscribe Model

Data exchange with DDS is realised according to the class structure defined in the Data Centric Publish Subscribe model (DCPS). This model describes the interfaces and relations of all entities that participate in the communication which is shown in Fig. 2. Although, fundamental knowledge of these classes and their relationships is important to understand DDS, it should be noted that they are automatically generated for each application based on the data type definitions. The applications must instantiate these classes and use the provided methods in order to achieve communication.

The core of the model is the class *Entity*. It is configurable with Quality-of-Service policies and can be attached with listener objects to be notified about events. Due to the inheritance relationship these characteristics are passed on to all other classes of the model, each of them defining a specialised set of *QoSPolicies* to fine-tune data transfer.

The class *Topic* represents a data flow that is defined by an unique identifier and a data type. More specifically, it connects the publishing and the subscribing ends of the communication. The former consists of the *Publisher* class that is internally used by the middleware to send out data. It is associated with multiple instances of the class *DataWriter* which provides a data type specific access for the application to trigger the publisher. The subscribing side of the communication is similarly structured. Internally, data is received and made available by *Subscriber* objects. These can be accessed by the application through data type specific *DataReader* objects.

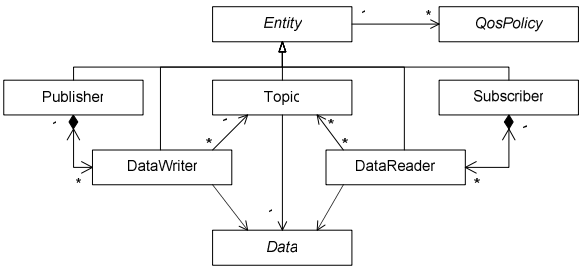

Fig. 2 Simplified Class diagram of the DCPS model (adopted from [10])

B. The Quality-of-Service Concept

The term Quality-of-Service refers to a general concept used to specify and control the behaviour of the communication service. The concept offers the advantage that the application developer only needs to indicate 'what' is required rather than 'how' this behaviour is achieved [10]. In particular, QoS provides the ability to manage the use of resources like network bandwidth or memory as well as reliability, timeliness and persistence of the data transfer.

The QoS model defined for DDS is a set of classes which are derived from *QosPolicy* and therefore can be attached to all objects that are involved in the communication. Each of these policies associates a name with a value and controls a specific aspect of the behaviour of the service. The specification defines separate semantics for the publishing and the subscribing side of each QoS parameter. To ensure correct communication, the QoS policies at the publisher side must be compatible with those at the subscribing end. To address this issue, the middleware automatically verifies if the QoS settings for corresponding publishers and subscribers match according to the subscriber-requested, publisher-offered pattern. According to this pattern, 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 [10].

The utilisation of QoS settings particularly addresses the needs of embedded real-time applications because it provides predictability and resource control. At the same time the

concept preserves the flexibility inherent to the publish/subscribe model. Additionally, a number of challenges that exist for the development of flexible manufacturing systems can be solved with that concept which is demonstrated in section IV.

III. Addressed Automation Challenges

DDS is an application-neutral middleware standard. It has been predominantly adopted for distributed communication in the defence & aerospace sector and for traffic control systems. Although, a few industrial automation systems have been implemented with DDS, it appears that the standard has not yet gained much significance for the automation of manufacturing systems. The reasons for this are manifold. On the one hand, the standard is still comparatively young and the high investment risks of manufacturing systems often force suppliers and end-users to use well-proven traditional technologies. On the other hand, there is still a lack of a support ecosystem that facilitates enhancing de facto standard control systems like PLCs with DDS-based communication without deep programming knowledge. However, as manufacturing systems are gradually moving towards utilising modular concepts and data distribution via Ethernet, DDS becomes an increasingly interesting technology for this sector. The reason for this is that DDS offers off-the-shelf features that have the potential to solve a number of challenges of distributed plug & produce manufacturing systems. The following sections discuss these challenges and show how they are addressed by the Data Distribution Service.

A. Impedance Mismatch

This refers to the problems with the integration of applications that have different requirements for the data exchange, such as data volume, data rates or timing constraints [11]. In a complex distributed system, some applications might produce data at much higher rates than others are able to consume. An example of this is the integration of administrative and business applications with the field-devices of the shop floor. Also Human Machine Interfaces often monitor processes which produce data at a very high frequency whereas only a subset of the data needs to be displayed. It would be fatal if the HMI was notified about every change of the sensor data, since it requires sufficient processing time for its graphical user interface.

With traditional approaches like client/server-based architectures this is a major problem because of the tight coupling between the data producer and those applications consuming the information. DDS overcomes this problem with minor programming efforts with its QoS concept. The subscriber objects can be configured with the TIME_BASED_FILTER QoS to precisely specify the minimum separation between received data samples. This way, the subscribing application can fully control how often it is updated with new data samples, regardless of how fast changes occur.

Furthermore, there is a trade-off between delivery reliability and delivery timing [12]. For some applications fast data transfer is decisive whereas for others the reliability is more important. For instance, a data channel transmitting a safety-critical command sequence must ensure that all commands are received in the right order, even if this imposes delays. This means, that the communication infrastructure must ensure resending of lost messages. On the other hand, for a controller processing time-critical sensor input it is more important to be updated with the most-recent sensor information and it is acceptable to ignore a few lost signals in between. DDS addresses this trade-off with the QoS policy RELIABILITY which can be attached to DataWriter and DataReader-objects. The policy has two possible values, RELIABLE and BEST_EFFORT. The first value makes sure that no data gets lost during the communication, while the second value can be used for connections that can ignore lost packages. Thus, each communication channel can be individually configured according to the trade-off decision without any further programming effort.

B. Plug & Produce

In order to react to changing product or process requirements it must be possible to dynamically add or remove modules into the production system. For instance, a modular fixturing system as described in section IV must be extendible with further fixture modules to reconfigure for a different workpiece. Similarly, it should be possible to plug and unplug HMI applications to and from a production system without affecting the overall process. This results in dynamically changing network topologies where the appearance and disappearance of modules need to be discovered.

DDS addresses this aspect with the automatic discovery of participants. New publishers and subscribers for a topic can appear at any time and the middleware provides mechanisms to notify the applications about their discovery. Similarly, applications are informed when participants are removed. Consequently, this simplifies the development of the distributed communication infrastructure of the manufacturing system.

Secondly, for plug & produce systems it is critical to provide late-joining modules with information that has been shared before they joined the system. For example, a late-joining module needs to be provided with information to interpret data coming from other subsystems or with the latest state information. As the example described in section IV shows, DDS provides a simple, yet effective mechanism to automatically redistribute data samples to late-joining applications by means of the QoS concept.

C. Platform-Independence

The subsystems comprising a production system are often developed by independent suppliers. This results in heterogeneous application landscapes with processes that utilise different hardware architectures, operating systems and programming languages. Not only does this impose problems of integrating these components to smoothly work together.

Also, each component might be subject to incremental changes or upgrades.

The DDS standard itself is defined by a Platform Independent Model (PIM). Secondly, data to be exchanged with DDS is also defined in a platform-independent way, using a subset of the Interface Definition Language (IDL) standard [13]. Consequently, it can theoretically support any combination of processor architecture, programming language and operating system. Available DDS middleware solutions allow seamless communication between the programming languages C, C++ and Java and many operating systems such as VxWorks, Windows, Lynx and Unix derivates. Moreover, using DDS makes all these details transparent for the application source code. In other words, no additional programming effort is required to establish data exchange between different platforms or to migrate subsystems from one platform to another. Hence, DDS can significantly reduce the difficulties of the system integration task for the automation of future manufacturing systems.

Finally, the data-centric approach of DDS connects individual applications only by means of the data model and decouples information exchange by means of topics. For these reasons DDS reduces dependencies between processes compared to traditional object-oriented or client/server approaches. While the latter lead to tightly-coupled applications that are connected by their behaviour in the form of method interfaces, DDS establishes interactions by means of the most constant part of a system which is the data model. This supports the incremental and independent development of individual subsystems as long as these changes are only concerned with the behaviour of the participants.

IV. Example

This section presents the development of a software-based prototype for a modular active fixturing system. Fixtures are devices that are used to support, locate and hold a workpiece into a desired orientation in space during manufacturing. Hence, they are essential in guaranteeing the quality of the final product in machining and assembly processes [14]. The prototype was developed using the commercially available DDS implementation RTI DDS 4.1e from Real-Time Innovations, Inc and demonstrates several of the previously discussed aspects.

A. Basic Concept of the Modular Active Fixture

The proposed fixturing system utilises sensor feedback to dynamically adapt clamping forces during the manufacturing process. Fig. 3 presents the conceptual design principle of this system. The fixture comprises an arbitrary number of fixture modules which are mounted on a rail frame. The latter allow flexible positioning and movement of the fixture modules when the fixture needs to be reconfigured for another workpiece. Each fixture module has a linear actuator that acts as the locating and clamping pin against the workpiece. Additionally, it incorporates sensors to feedback displacement, force and temperature.

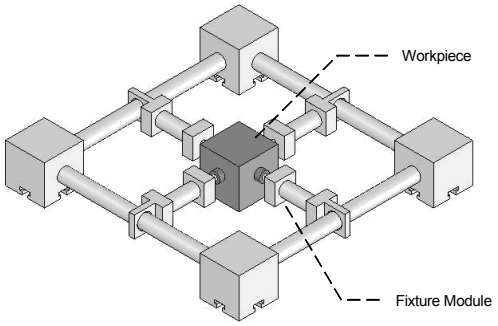

Fig. 3 Conceptual Design of the fixturing system [15]

From a software point-of-view, a distributed approach is proposed where each fixture module has its own local software program. These programs need to interact with each other and a global coordinating software, called fixture control, which realises the overall fixturing strategy. Additionally, the infrastructure shall be open to allow communication with other subsystems like HMIs, the machine control and others.

B. Overview of the Software-Based Prototype

On the basis of this concept, three software programs have been developed using the programming language C++, namely FixtureModule.exe, FixtureControl.exe and HMI.exe. They are used to demonstrate the basic communication taking place in the described fixturing system and to illustrate the previously described aspects of DDS like automatic discovery, impedance mismatch etc.

The FixtureModule program stands for the local software running on each module. It is provided with a unique numerical identifier (module-ID) by means of a command line parameter. Furthermore, each module is configured with information about its internal device structure which in this prototype is fictional only. In a fully functional prototype, each module software would be responsible for accessing the sensor devices and transforming the raw sensor signals into meaningful information (e.g. Force in Newton). Similarly, it would have to realise the logic to generate the signals for its actuator in order to achieve a certain desired state, i.e. displacement or reaction force. In this prototype, however, the module software simulates the existence of internal sensors and actuators by instantiating software objects for them. The objects representing the sensor devices continuously produce random data. On the other hand, the software-object that represents the actuator changes its internal positional information in order to imitate an actuation. In order to allow other systems to communicate with the module software in a meaningful way, it is required that the module's capabilities are published when the module is launched. For instance, other systems need to be able to verify if the fixture module measures force in Millinewton (mN) or Newton (N) etc. After this, it continuously publishes its current sensor states and expects desired states for its actuator.

The fixture control program (FixtureControl.exe) coordinates the global behaviour of the overall fixturing

system. For this, it needs to discover all fixture modules on the platform and to retrieve their capability information before it can communicate with them in a meaningful way. During the manufacturing process, it interacts with the fixture modules by continuously receiving their sensor states and issuing desired states for their actuators.

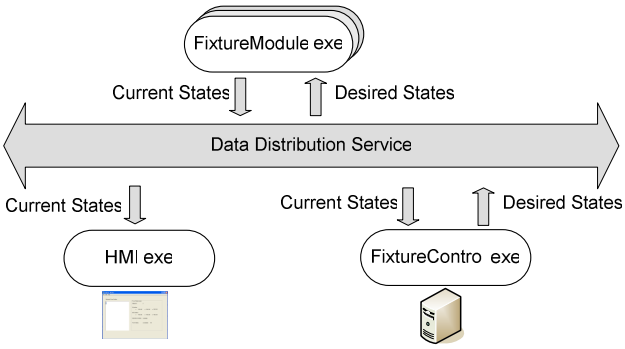

Fig. 4 Overview on the example application

As the third program, a simple Human Machine Interface (HMI.exe) has been developed for demonstration purposes. Similar to the fixture control it needs to dynamically discover the fixture modules and their capabilities. It is constantly updated with the sensor states of all modules and displays them on the graphical user interface. However, unlike the fixture control, it does not send any information to the modules. This paper concentrates on the design of the distributed data transfer with DDS. For this reason, no further details about the application logic of these three programs are reported.

C. Further Requirements

In order to illustrate how DDS addresses the challenges discussed in section III, a number of additional requirements for the overall system have been defined as follows:

1. Fixture modules must transmit additional information to allow interpretation of their data to other systems. This information should be issued at the initial start-up of each fixture module. Subsystems that are launched at a later time must be automatically provided with this information. In other words, no particular start sequence should be imposed for the fixturing system.

2. To illustrate the impedance mismatch it is defined that HMI applications shall be updated with force sensor readings only every 1000ms, regardless of how fast the modules publish this information. In a similar way, HMIs shall receive only one temperature sample every 4000ms.
3. Although all fixture modules have a temperature sensor, it is defined that the fixture control and the HMI applications only receive temperature readings from one fixture module at a time. If this most-trusted module fails to deliver a sample within 6000ms, temperature data shall automatically be received from another module.

D. Design of the Data Model

The first step of the data-centric application development is the definition of the data structures to be exchanged between the processes. In this context, there is a trade-off between efficient data transfer and flexible interpretation of data. On one hand, it shall be allowed to add fixture modules with varying capabilities, i.e. different hardware characteristics and representations of data. This makes it necessary that each module informs other systems about its capabilities which define how data has to be interpreted. On the other hand, it is not efficient to publish this meta-information with every sensor update. For this reason, it is proposed to separate actual state data from meta-information to interpret it. Therefore, two data structures are created for each capability of a fixture module. The first data structure is used for the transmission of the sensor readings or desired states for actuator devices during the manufacturing process. It is a very simple data structure that only consists of a field for the numeric module-ID and the data itself. Below an example is provided for the data structure for force sensor readings. Each attribute is defined with a data type, followed by a name.

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

Since this structure does not contain any information on how to use the data, an additional data structure is defined for each capability. It contains attributes describing the characteristics of the relevant capability like measuring range, resolution etc. In this prototype the meta-information only contains the measuring range for capabilities that result from the existence of sensor devices. This is further defined by attributes for the minimum and maximum measuring value, as well as the measuring unit. For the latter unique numerical constants have been defined. The following listing provides the data definitions for the capability that results from the existence of a force sensor. Similar structures have been defined for the other capabilities.

```
struct MeasuringRange {
  double min;
  double max;
  long unit;
};

struct SenseReactionForceCapability{
  MeasuringRange measuringRange;
};
```

Instead of publishing each of these data types separately, they are further aggregated in a single record which finally provides a complete representation of the module functionalities. This structure is shown below and contains the

numerical identifier of the fixture module and has attributes for each capability.

```
struct FixtureModuleCapabilityDef{
    long id;
    SenseTipPositionCapability
       senseTipPositionCapability;
    AdjustTipPositionCapability
       adjustTipPositionCapability;
    SenseBodyPositionCapability
       senseBodyPositionCapability;
    AdjustBodyPositionCapability
       adjustBodyPositionCapability;
    SenseReactionForceCapability
       senseReactionForceCapability;
    AdjustClampingForceCapability
       adjustClampingForceCapability;
    SenseTemperatureCapability
       senseTemperatureCapability;
};
```

During the initialisation routine of a fixture module, this data structure is instantiated and published as one sample. All other subsystems are thereafter able to communicate with this module in a meaningful way. This separation approach reduces network load and processing time during the manufacturing process. The reason for this is that at this time only simple data structures are exchanged through the network.

All data type definitions were saved in the file testbed.idl and with the following command all required classes for the data exchange with DDS are generated in the programming language C++. The three applications of the prototype must create objects of these classes and use their interfaces to achieve communication.

```
rtiddsgen -language C++ testbed.idl
```

E. Design of the Topic Structure

For the prototype five individual data topics have been defined. Each of them is bound to one data type which is illustrated in Fig. 5.

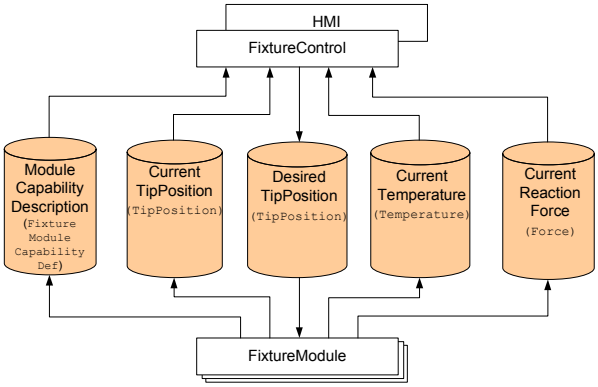

Fig. 5 Topic structure of the prototype

In the centre of the picture, the data topics are displayed with their unique identifier. Additionally, the data type that is exchanged through this topic is provided in brackets. Similar to Fig. 1, the arrow directions indicate whether an application is a publisher or subscriber with respect to a certain topic. Thus, the three applications play different roles for each topic. For example, the fixture module programs publish data into the topics "Module Capability Description", "Current Tip Position", "Current Temperature" and "Current Reaction Force" while the FixtureControl-program and the HMI are subscribers for these topics. Since the fixture control application is a publisher for the topic "Desired Tip Position", interaction is established between the modules and the fixture control.

F. Selection of Quality-of-Service Parameters

The next step is to configure the communication to meet the previously defined requirements. For this, QoS parameters need to be attached to the data writers and data readers which realise the information exchange.

The first requirement is a result from the separation of data structures as described in section IV.D and the strategy that each module publishes its capabilities only once during its initialisation routine. This means, a mechanism must be provided that allows late-joining processes to receive the capability information. In traditional, particularly client/server-based systems, the problem of redistributing historical data is often solved by periodical broadcasts or by explicit requesting the required information in a synchronous message sequence. Both approaches jeopardise timing-determinism and add complexity to the application logic of the modules. With minor programming effort, DDS can entirely take responsibility for the automatic redistribution of data whenever a new subscriber for this data topic is discovered. Hence, the development of the module software is significantly simplified.

For the realisation of this strategy, the data writers and data readers for the module capability descriptions need to be attached with the QoS settings as shown in Fig. 6.

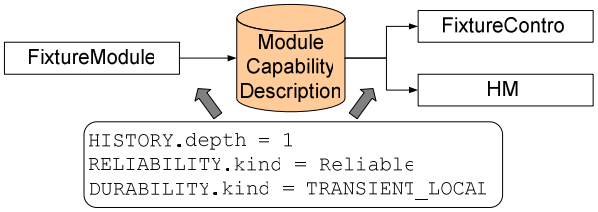

Fig. 6 QoS settings for Requirement I

For the publishing side, the QoS parameter HISTORY specifies if and how many published data samples are maintained for late-joining subscribers. With its attribute *depth* set to 1 and the DURABILITY.kind parameter defined as TRANSIENT_LOCAL, it is assured that the last published sample is stored locally. Finally, this strategy is only applicable for reliable data transfer which is specified by the value of the RELIABILITY parameter.

The second requirement aims at illustrating the impedance mismatch that can occur when applications with different communication requirements are integrated. For the HMI-

application it is critical not to be flooded with too much data because otherwise it would not have sufficient resources to handle its graphical user interface. Hence, a mechanism must be implemented to limit the number of samples. As described in section III, DDS provides a simple, yet effective means for this requirement. In the source code of the HMI application, the data reader objects for the force and temperature readings are attached with the TIME_BASED_FILTER parameter as illustrated in Fig. 7.

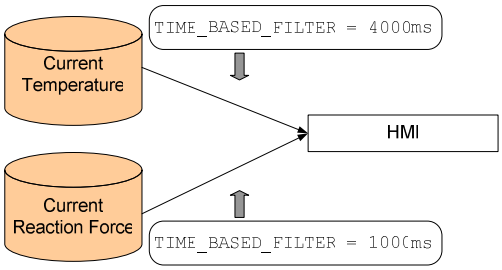

Fig. 7 QoS settings for requirement II

Finally, the third requirement demonstrates how robust systems can be implemented with DDS that are able to seamlessly switch from one data source to another, in case of failures. In production systems critical data is often delivered by redundant hardware. During normal operation, only data from a most-trusted data source is considered. However, if this source fails to deliver data for specified period, other applications must smoothly fail over to a backup. Again, the implementation of this feature in a manual way is a complex task and would further complicate the application source code of the fixture modules. Particularly for system architectures which rely on concepts like the client/server mechanisms or point-to-point communication, this is hard to realise due to the tight couplings these approaches induce.

Using DDS, the challenge is addressed with the QoS parameters OWNERSHIP and DEADLINE. As illustrated in Fig. 8 all data writer and data reader objects that access the topic for the temperature values are defined as “exclusive”. This indicates that temperature readings can only be updated by one data writer.

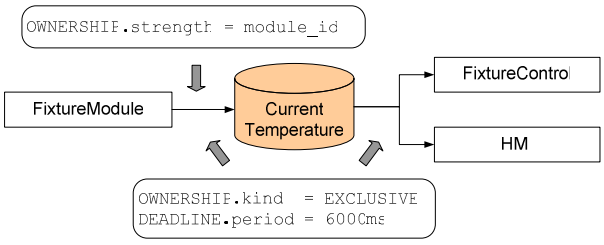

Fig. 8 QoS settings for requirement III

The selection of the most-trusted data writer is controlled by the setting of the OWNERSHIP_STRENGTH policy. For this reason, each temperature data writer is configured with the numeric ID of its fixture module. The middleware ensures that only the data writer with the highest strength is allowed to update the temperature readings. Finally, the DEADLINE parameter specifies the longest acceptable time span before subscribers automatically take temperature samples from the fixture module with next-higher ID. This way, fault-tolerant distributed applications can easily be developed with the ability to dynamically react to failures in the system.

G. Tests

To verify that the three applications can communicate over DDS according to the requirements, a number of qualitative test scenarios have been carried out. In particular, it was tested if the fixture modules and their capabilities are automatically recognised by other subsystems, regardless of the point in time when the latter are started.

In the first test the fixture control and one instance of the HMI are started. As expected, both applications attach their publisher and subscriber objects to the relevant data topics as described in section IV.E and remain idle, since no fixture modules have been started, yet. Then, three instances of the fixture module program are launched. Each program is started with a different identifier. Consequently, both the fixture control and the HMI discover the three modules, receive their capabilities and start receiving sensor states from each of them. Fig. 9 shows a screenshot of the HMI-application after all programs have been started. All discovered fixture modules are displayed in the list on the left side of the user interface while the current sensor data of the selected module are constantly updated on the right side. The picture also indicates that the HMI has correctly received the capability descriptions. The selected module (1) was configured to issue force readings as values in Newton (N). Obviously, the HMI received this information and is therefore correctly displaying the received measuring unit.

Thirdly, it is obvious that the displayed temperature values stem from the module with the highest ID (3). During the test, this module program was aborted. Subsequently, it disappeared from the list in the HMI and after 6 seconds temperature values were taken from the module with the next-higher ID (2).

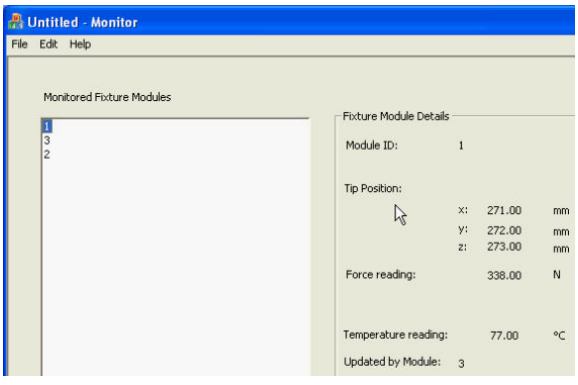

Fig. 9 Screenshot of the HMI during the test

In the second test it is verified if late-joining applications

can be provided with module capability descriptions. For this, the three fixture modules are started first and after a while the fixture control and one HMI is started. This means, that the former have published their capabilities before the latter processes have joined. The test proved that launching the programs in reverse order does not influence the overall functionality of the fixturing system. Like in the first test, the HMI was able to interpret the incoming sensor data from each fixture module. This demonstrates that the proposed system has a great level of flexibility and does not impose a specific sequence for its initialisation.

The final test illustrates the flexibility of the approach and shows how the DDS infrastructure of the fixturing system forms an open and scalable environment. In addition to the fixture control and the three fixture modules programs, another three instances of the HMI.exe are started. As expected, plugging in further HMIs did not influence the functionality of the modules or the fixture control. The reason for this is the decoupling of the communication through the data topics. In other words, the modules are not aware of the existence of the HMIs. It is acknowledged that a real manufacturing system hardly requires three HMIs. However, in this test scenario they stand for any kind of peripheral applications that might need to be connected with the fixturing system.

V. Conclusion

This paper has presented key aspects of the Data Distribution Service and has put them in context with flexible manufacturing. In particular, features like the Quality-of-Service concept, the automatic discovery of network topology changes and the redistribution of data to late-joining applications have been discussed against the background of typical challenges of distributed automation systems. These aspects are illustrated in a software-based prototype application that is based on a modular active fixturing system. The individual design steps for achieving communication with DDS have been described for this prototype and qualitative tests have been carried out. Future research will focus on extending the data structures and the development of a physical prototype for the proposed system.

Acknowledgment

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] T. Nakatani, K. Nakajima, S. Torii and B. Wataru Higemoto, "Prototype of network distributed control system for MLF/J-PARC", Physica B, 2006, vol. 385-386, pp. 1327-1329.
[2] B. Mitschang, "Data propagation as an enabling technology for collaboration and cooperative information systems ", Computers in Industry, 2003, vol. 52, pp. 59-69.
[3] J.C. Campelo, et al., "Distributed industrial control systems: A fault-tolerant architecture", Microprocessors and microsystems, 1999, vol. 23, pp. 103-112.
[4] B. Rostamzadeh, H. Lonn, J. Snedsbol and J. Torin, "DACAPO: A distributed computer architecture for safety-critical control applications", in IEEE International Symposium on Intelligent Vehicles, Detroit, USA, 1995.
[5] B. Rostamzadeh and J. Torin, "Design principles of fail-operation/fail-silent modular node in DACAPO", in Proceedings of the ICEE, Tehran, Iran, 1995.
[6] J. Arlat, et al., "Experimental evaluation of the fault tolerance of an atomic multicast system", IEEE Transactions on reliability, 1990, vol. 39, no. 4.
[7] 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, 2006, vol. 19, pp. 593-697.
[8] I.M. Delamer and J.L. Martinez Lastra, "Quality of service for CAMX middleware", International Journal of Computer Integrated Manufacturing, 2006, vol. 19, no. 8, pp. 784-804.
[9] I.M. Delamer, J.L. Martinez Lastra and R. Tuokko, "Design of QoS-aware framework for industrial CAMX systems", in Proceedings of the Second IEEE International Conference on Industrial Informatics INDIN 2004, Berlin, Germany, 2004.
[10] Object Management Group, "Data Distribution Service for real-time systems, version 1.2", 2007, Available from: www.omg.org, June 2007.
[11] J. Joshi, "Data-oriented architecture", Real-Time Innovations, Inc., unpublished, 2007, Available from: www.rti.com.
[12] Real-Time Innovations, Inc., "Can ethernet be real time?" Real-Time Innovations, Inc., unpublished, 2006, Available from: www.rti.com.
[13] Object Management Group, "Common Object Request Broker Architecture: Core specification, version 3.0.3", 2004, Available from: www.omg.org, August 2007.
[14] M. Ryll, T.N. Papastathis and S. Ratchev, "Towards an intelligent fixturing system with rapid reconfiguration and part positioning", Journal of Materials Processing Technology, 2007, to be published, corrected Proof.
[15] T.N. Papastathis, M. Ryll and S. Ratchev, "Rapid reconfiguration and part repositioning with an intelligent fixturing system", in ASME International Conference on Manufacturing Science & Engineering (MSEC2007), Atlanta, Georgia, 2007.