A framework for DRE middleware, an application to DDS

Jérôme HUGUES, Laurent PAUTET
GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
jerome.hugues@enst.fr, laurent.pautet@enst.fr

Fabrice KORDON
Université Pierre & Marie Curie, Laboratoire d'Informatique de Paris 6/SRC
4, place Jussieu, F-75252 Paris CEDEX 05, France
fabrice.kordon@lip6.fr

Abstract

Heterogeneous non-functional requirements of DRE system puts a limit on middleware engineering; building an application-tailored middleware becomes a challenge.

In this paper, we show how we use the PolyORB middleware and its architecture as a framework to implement DDS, the Data Distribution Services (DDS) recently published by the OMG. We demonstrate how the architectuer proposed by PolyORB enables a rapid implementation of this specification, and allows for extreme tailorability to support application requirements.

1 Introduction

The many and heterogeneous constraints of distributed applications deeply impact the development of distribution middleware. Middleware should support developers when designing, implementing and deploying such systems in heterogeneous environments and evaluate so called "non functional" requirements (such as QoS or reliability).

Usually, a system is designed as shown in Figure 1. Applications interact with middleware implementing one specification (e.g. CORBA). Non functional requirements are considered during the implementation of the application (fault tolerance, resources, etc.).

One could select a COTS middleware that fulfills all its non-functional requirements. However, this situation is not realistic: each product has its strengths and weaknesses.

The implementation of non-functional elements at the middleware level is of interest: it allows to factor out common code, reused by many applications. More code is reused and thus more tested and trusted. Yet, it induces a more complex and heavy architecture.

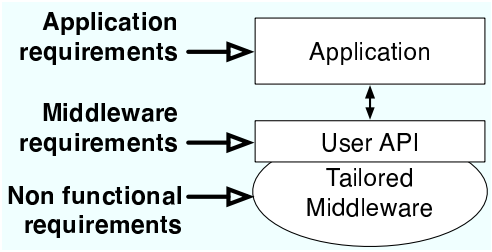

Figure 1. Various levels of requirements

Such an approach is not affordable for Distributed, Real-time and Embedded Systems (DRE). Meeting their non-functional requirements imposes precise control over the middleware to ensure that temporal or resources guidelines are met. This set of constraints is usually defined as a set of "Quality of Services" policies and parameters.

As an example, Zen-Kit [4] allows middleware customization (such as control over memory footprint) by controlling the actual components embedded by the application, while minimizing the difficulty of this custom configuration. However, this approach is focused on RT-CORBA: it does not allow the modification of the middleware architecture by adding new components that would diverge from the standard such as a lightweight invocation protocol.

Hence, there is a need for a finer control over the architecture and services inside the middleware. It may also require modifications to the application API (e.g. to let the application provide information to define some non-functional aspects such as QoS). This can be seen as a process similar to the one of hardware/software co-design: the responsibility of supporting non-functional requirements is shared by both the application and the middleware.

The "Data Distribution Services" (DDS) [12] may act

this way. It is built around MDA (Model Driven Architecture) [10, 11], it defines a Platform Independent Model (PIM) of its core constructs. Then, middleware developers can derive any Platform Specific Model (PSM) that matches the applications requirements.

This allows the construction of different implementations of DDS, each one dedicated to very specific needs, allowing for efficient and precisely tuned middleware.

In turn, this puts a strict constraint on the application: it has to conform to one specific PSM. Incompatible PSMs and their implementation reduce applications' portability. It is another instance of the "Middleware Paradox" that limits interoperability between middleware technologies [1].

We claim that DDS is an promising step to elaborate middleware dedicated to a given application that supports its non-functional requirements. Yet, this requires a framework to define this family of middleware that maximizes application portability. This paper presents how we use PolyORB as a framework for supporting DDS.

We present in the next section the key concepts of DDS, then we analyze their implication when defining a supporting middleware. We introduce the key concepts of the "schizophrenic" middleware architecture and demonstrate how they can be used to efficiently prototype DDS, defining one PSM for the user, and several hooks to tailor middleware internals. Finally, we analyze our implementation and compare it to existing DDS implementations.

2 DDS' overview

In this section, we present the core concepts of DDS and discuss the implementation of a DDS middleware.

The Data Distribution Service (DDS) aims at defining a communication infrastructure that covers the needs of large-scale, distributed real-time applications. It is a new specification adopted by the OMG at the end of 2004.

DDS follows the Publisher/Subscriber distribution model defined in [13]. It is data-centric: the unit of transmission is data. The underlying data model use a global data space to identify data circulating in the system.

The DDS specification defines high-level standardized interfaces and behavior. It introduces two layers:

• DCPS (*"Data-Centric Publish/Subscribe"*), a mandatory low-level layer that handle the efficient delivery of data to the proper recipients. This layer provides support for 21 Quality of Service policies;

• DLRL (*"Data Local Reconstruction Layer"*), an optional high-level API that eases access to and filter data manipulated by DDS's DCPS layer.

DDS defines several compliance points, from "Minimal Compliance" (core of DCPS) to "complete", including many services and the DLRL. Each compliance level indicates the number of services available.

2.1 DDS Architecture Conceptual Outline

In this section, we describe DDS at a conceptual level. We focus on the DCPS layer.

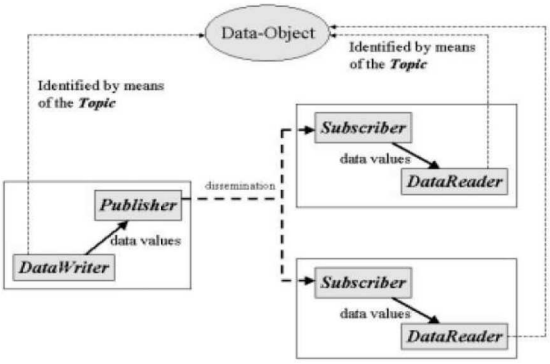

Figure 2. DDS entities from [12]

The information flow in the DDS is presented in figure 2. The main actors of the DDS data-flow are the:

`Publisher` and `DataWriter` on the sending side, and the `Subscriber` and `DataReader` on the receiving side. `Topics` are used to identify and collect data.

`DomainParticipant` is the factory class for `Publisher` and `Subscriber`. It is created by any DDS application prior to `Publisher`/`Subscriber` creation.

`Publisher` handles information dissemination, or publication. It aggregates many `DataWriters`, each `DataWriter` is associated with a type of data. `DataWriter` are used by the application to send data.

`Subscriber` handles the retrieval of published data. As for the `Publisher`, the `Subscriber` aggregates several `DataReaders`, each associated with a type of data. The application accesses data through the `DataReader`.

`Topic` associates a unique name and a data-type. This topic name is used by the `Publisher` and `Subscriber` sides to send and receive data. A `Topic` might allocate some storage for data. A `Topic` aggregates data in groups called instances. Each instance is identified by a key inside the `Topic`. An instance contains many samples, each one represents a value for the instance. `Publications` and `Subscriptions` use samples as the communication unit.

DDS supports two notification mechanisms:

• *Listener-based:* This is the default and mandatory notification mechanism. It provides a generic mechanism to notify the application of the occurrence of relevant asynchronous events, such as data arrival, QoS setting violation etc. Each DDS entity has its own set of listeners. The specification only defines which are the listeners a particular entity accepts, and in which conditions they are to be called.

• *Condition-based:* This mechanism allows the application to wait for a condition to occur. Conditions can be

triggered by the application, the status of an entity or the arrival of data. The application can wait simultaneously on several conditions using a `WaitSet`.

These two notification mechanisms allow one to choose between synchronous and asynchronous notification.

The Data Distribution Service supports Quality of Service (QoS). QoS provides the possibility to tailor the behavior of DDS entities: each Entity supports its own set of QoS policies. QoS policies are many and diverse, they address different areas of middleware behavior, ranging from time management to resource utilization, transport configuration, fault tolerance, persistence, data presentation, etc.

2.2 Deriving an implementation of DDS

The elements we presented provide only a brief overview of DDS capabilities. In this section, we discuss their design principles, strength and prerequisites.

DDS differs from typical CORBA-based OMG specifications in many ways, including its form and contents. DDS is built around MDA (Model Driven Architecture) [11]: it defines a Platform Independent Model (PIM) of its core constructs. This PIM is defined using UML, fully describing the API and semantics of the DDS entities, their role, parameters and the various QoS policies.

From this PIM, middleware developers are free to derive any Platform Specific Model (PSM) that matches their design goals. The mapping has to preserve the overall semantics, but can rely on any intermediate technologies before producing the final implementation.

We note that DDS relaxes many normative points typical from CORBA-related specifications, including protocol, data representation, run-time, etc. This allows for platform and application-specific optimizations.

DDS targets DRE systems, the mapping should abide with engineering guidelines from this domain, and the heterogeneous hardware platform (e.g. supporting OS, dedicated I/Os). Defining the PSM should take advantage or cater for these constraints.

Such a freedom in defining its own mapping is interesting and unprecedented in middleware specifications: this allows for the implementation of fine-tuned middleware, while using one common conceptual framework at both the API and semantics levels.

Yet, we believe this introduces one strong limitation on the use of DDS: different needs might lead to different and incompatible mappings at the PSM level, impeding code reusability between DDS infrastructures.

This situation would be very similar to the "Middleware Paradox" that prevents interoperability between applications built around heterogeneous middleware.

There is a need to map concepts proposed in DDS into a safe architecture that enforces reusability of major pieces of code. This architecture also has to enhance adaptability of the middleware: developers must be able to safely adapt middleware services to the needs of an application domain.

We claim that the value added of any middleware, including DDS ones is not in its API, but in its internals and its support for the underlying OS and hardware platforms, its PIM. We advocate for enforcing separation of concerns at the middleware level to provide the user with a stable API, providing a uniform view of the PSM, and with a set of tailorability mechanisms to precisely adapt the middleware to application needs. These two complementary facets allow for the easy adaptation of the middleware.

In the following sections, we detail our approach to build DDS middleware that preserves its API and allows for many tailorability strategies, based on the use of the schizophrenic middleware architecture.

3 Defining a design and implementation framework for fine-tuned middleware

In this section, we discuss our approach to design a middleware dedicated to the requirements of a given application. This approach can be viewed as a co-design between the application and its supporting middleware. To enforce this design process, we provide a highly generic middleware architecture (also known as the "schizophrenic" architecture) and a methodological guide to instantiate it.

3.1 PolyORB, a highly tailorable middleware

Actual middleware has to fulfill the system requirements. Some solutions are based on "rigid" specifications; this is the case for most CORBA implementations. Such middleware architectures are targeted to a certain application domain. They can be adapted to other fields of application (RT-CORBA, minimum CORBA...). Yet, implementations are not as efficient as specifically designed middleware [9].

Thus, there is a need for middleware that can really fit many different systems. To do so, both a tailorable and verifiable architecture is required.

3.1.1 The need for a tailorable architecture

Solutions have been proposed to design tailorable middleware. *Configurable* middleware defines an architecture centered on a given distribution model [14] (e.g. distributed objects, message passing, etc.); this architecture can be tuned (tasking policy, etc.). *Generic* middleware [2] provides a canonical architecture, which has to be instantiated to create middleware implementations. Those implementations are called *personalities*. Generic middleware is not bound to a particular middleware model; however, various personalities seldom share a large amount of code.

Configurable and generic middleware architectures address the tailorability issue, as they ease middleware adaptation. However, they do not provide complete solutions, as they are either restricted to a class of distribution model, or their adaptation is too expensive.

3.1.2 Decoupling middleware functions

As a solution to address those issues, we proposed the schizophrenic middleware architecture [17]. It separates concerns between distribution model APIs, communication protocols, and their implementations. Schizophrenic middleware refines the definition and role of personalities.

The schizophrenic architecture consists of three layers: *application-level* and *protocol-level* personalities built around a *neutral* core. Application uses the application personalities; protocol personalities operate with the network.

Application personalities constitute the adaptation layer between application components and middleware through a dedicated API or code generator. They provide APIs to interface application components with the core middleware; they interact with the core layer in order to allow the exchange of requests between entities.

Application personalities can instantiate middleware implementations such as CORBA, the Distributed System Annex of Ada 95 (DSA), the Java Message Service (JMS), etc.

Protocol personalities handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages exchanged using a chosen communication network and protocol. Protocol personalities can instantiate middleware protocols such as IIOP (for CORBA), SOAP (for Web Services), etc.

The neutral core layer acts as an adaptation layer between application and protocol personalities. It manages execution resources and provides the necessary abstractions to transparently pass requests between protocol and application personalities in a neutral way. It is completely independent from both application and protocol personalities.

The neutral core layer enables the selection of any combination of application and/or protocol personalities. Several personalities can be collocated and cooperate in a given middleware instance, leading to its "schizophrenic" nature.

3.1.3 The middleware core architecture

The middleware core provides neutral, model-independent functions. It relies on the identification of the functions involved in request processing and their flexible implementation that maximize code reusability among personalities. Figure 3.1.3 details the neutral layer.

The inner heart of the neutral layer is embodied by the "μBroker" [7]. The μBroker provides all the basic middleware mechanisms: I/O, task scheduling, etc. It is formally described, to support verification facilities. Hence it is possible to ensure its properties (no deadlock, no livelock, etc.), and we actually verified some configurations.

The personalities do not directly interact with the μBroker. They are built on top of seven fundamental services that provide the client-server interactions found in most distribution models. Those services define the canonical operations performed in a middleware implementation.

First, the client looks up server's reference using the *addressing* service (1). Then, it uses the *binding* factory (2) to establish a connection with the server, using one communication channels (e.g. sockets, protocol stack).

Request parameters are mapped onto a representation suitable for transmission over network, using the *representation* service (3) (e.g. CORBA CDR).

A *protocol* (4) supports transmissions between the two nodes, through the *transport* (5) service; it establishes a communication channel between the two nodes.

Upon the reception of a request, the middleware instance ensures that a concrete entity is available to execute the request, using the *activation* service (6). The *execution* service (7) assigns execution resources to process the request.

The composition of the core services around the μBroker allows for the implementation of many distribution models.

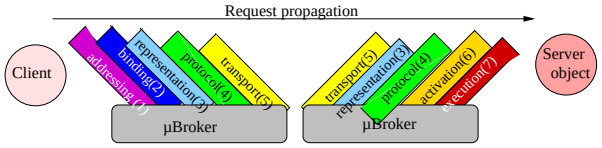

Figure 3. PolyORB's core architecture

3.1.4 Assessment

The seven fundamental services provide the basic middleware operations. The μBroker can be formally described to provide verification facilities, to ensure real-time properties. Those elements can be configured to create a middleware instance for particular system requirements.

In [17], we present PolyORB our implementation of a schizophrenic middleware. PolyORB a free software middleware supported by AdaCore[1], PolyORB's research activities are hosted by the ObjectWeb consortium[2]. We assessed its suitability as a middleware platform to support multiple heterogeneous specifications (CORBA, Ada Distributed Systems Annex, Web Applications, Ada Messaging Service close to Sun's JMS) and as a COTS for industry projects.

[1] `http://www.adacore.com`
[2] `http://polyorb.objectweb.org`

3.2 A methodology to design new personalities

A methodological guide details the different steps to instantiate PolyORB (figure 4) from a specific set of application requirements and the implied distribution model (step 1). It is intended to give the user the proper knowledge to tailor PolyORB. There are several ways to adapt PolyORB to the application requirements (step 2):

• Use an existing personality. PolyORB already comes with CORBA, RT-CORBA, DSA, MOMA (Ada-like JMS), DDS and the existing configuration parameters;

• Design a new personality: design or refine some of the fundamental components, by re-using fundamental components already developed from existing personalities or from the neutral core; overloading them or designing new variant of fundamental components from scratch.

Note that when a new personality is designed, we get back to the generic architecture (step 3) to decide whether the new features would be useful for other personalities. In this case, there are two possible policies:

• This feature has a simple and generic enough implementation that can be reused by other personalities, then the feature is integrated in the pool of neutral core layer components, e.g. concurrency policies, low-level transport;

• This feature is intrinsically specific to a personality, the implementation enhancement is kept at the level of the protocol or application personalities, e.g. GIOP message management, DDS specific API.

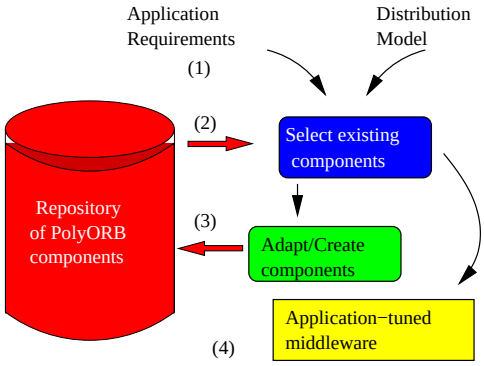

Figure 4. Designing new personalities

Finally the user derives one assembly of components: the fine-tuned middleware adapted to its initial needs (step 4).

This procedure may also be repeated to adapt more precisely components, allowing for evolving design of some core elements without impeding the whole assembly.

In this section, we have defined the middleware architecture and associated methodology used to implement middleware. We enforce a strong separation of concerns between the different functions involved in the middleware and we combine them to form the required implementation.

In the next section, we detail how this process enables the construction of a DDS middleware.

4 Building a DDS personality

In this section, we introduce the design of a DDS (OMG specification v1.0) personality for PolyORB.

4.1 Design requirements

We presented the overall architecture and concepts of PolyORB, focusing on the distribution functions it provides, and the personality as a structuring paradigm to implement a distribution mechanism or specifications. We now propose to study how to implement DDS in our framework.

We note that DDS defines the semantics of many application objects, their API, but do not impose any exchange protocol, representation or transport mechanisms similar to CORBA's GIOP and CDR.

Hence, DDS is an application-oriented specification, the implementation has to define its own core and protocol elements. Transposed in the context of the schizophrenic middleware architecture, DDS requires a specific application personality, it is the developer responsibility to carefully select the neutral core and protocol components.

Therefore, the initial implementation of DDS will focus on the implementation of a dedicated application personality that supports DDS entities and their semantics. The tailorability of the middleware will be provided by carefully selecting the other middleware components.

PolyORB proposes a MOM application personality: MOMA. MOMA implements an API very close to that of Sun's JMS [16], it is a MOM with the notion of Publisher and Subscriber, with Topics to identify data, etc. All these entities are also defined in DDS.

Therefore, DDS is notionally close to JMS. It differs not in its functional elements but in the definition of real-time oriented QoS policies and in implementation guidelines that strives for efficiency and determinism.

Besides, PolyORB implements the RT-CORBA specifications. To support this API, PolyORB's core and GIOP protocol personality have been enhanced to provide support for priority-aware request propagation and QoS policies.

Hence, up to a tailoring of some MOMA and PolyORB's internals, and the careful selection of other components, PolyORB's architecture provide sufficient support to build a DDS personality. We detail this point in the next section.

4.2 Reusing middleware components for DDS

We now review some existing elements we use as a framework to implement DDS :

MOMA provides MOM canonical entities and control on the message lifecycle [6]. Entities provides the user with the capability to define topics, subscribe and publish data. All entities rely on the "servant-like" pattern: they are registered to the μBroker and the application entity manager, they serve incoming requests and store/retrieve messages.

Message exchange between entities is mapped to a set of requests these servants can serve, and that can in turn be mapped onto GIOP request messages.

These elements provide the basis to implement DDS entities on top of MOMA, allowing for code reuse.

Neutral Core Middleware controls request propagation, the management of application entities. It schedules tasks to complete request execution, in conformance with the QoS policies set up by the application.

The Neutral Core Middleware is highly configurable and tailorable: each step in request processing can be adapted. Previous work on the RT-CORBA specifications added priority-based request execution mechanisms to the core, and in particular the `CLIENT_PROPAGATED` model to set up the priority at which a threads executes a request.

Application entities can be registered with QoS policies similar to the `RTPortableServer`'s API, without the encumbrance of the CORBA's concepts and its mapping to a programming language. The core provides supports of the concepts, the personalities add all required wrapper to match a model, e.g. CORBA's OMA or DDS QoS policies.

Hence, the Neutral Core can be configured to manage MOMA entities with QoS policies adapted to DDS needs.

Protocol: DDS does not define a protocol stack to exchange data, only provision for efficiency.

Without loss of generality, we chose the GIOP personality as an element for our first-step prototype, it provides versatile mechanisms to exchange data and QoS information between nodes. GIOP proposes `oneway` request propagation mechanisms, the MIOP protocol also proposes group communication. This provides sufficient features for MOM.

Let us note that the versatility of PolyORB allow us to change the protocol at a limited cost, focusing more precisely on application needs for efficient data transfer.

4.3 Defining the DDS personality

We presented the elements provided by PolyORB to support DDS. DDS focuses on the definition of an efficient publish/subscribe middleware for DRE systems. Compared to MOMA concepts, it "simply" adds 21 QoS policies.

A careful review of these policies show that most of them are related to the message life-cycle (ordering, historic, limits on resource, etc.). MOMA entities simply provide message passing capabilities. The DDS personality will rely on this feature and enforce specific behavior. They impose the specific implementation of the components managing access to message in MOMA entities, and is directly under the responsibility of the DDS personality.

Time and priority oriented policies are already supported by the neutral core and the protocol personality as part of the work done for the RT-CORBA implementation.

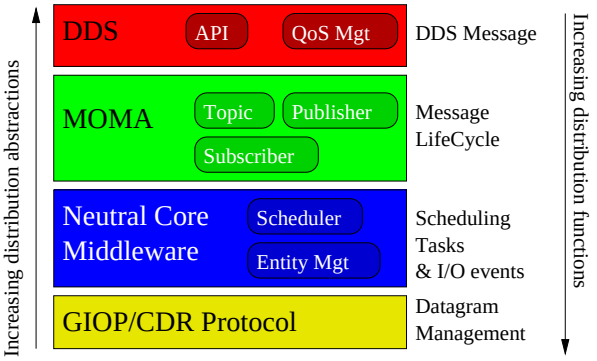

Figure 5. Abstractions/Functions in PolyORB

Figure 5 summarizes this adaptation work. The mapping from DDS PSM onto the application personality API is a direct mapping from the different objects onto Ada 95 objects and types. Their implementation code uses MOMA entities to manage DDS message lifecycle. They are configured using QoS policies and functions to support priority-aware request dispatching that are provided by the core and protocol components. It is an enrichment of the semantics of the message passing model.

This adaptation work focused on a specific area of the middleware: data management; other distribution issues are already addressed when reusing PolyORB's components.

In this section, we presented our solution to implement DDS. It focuses on high-code reuse through an intensive separation of concerns and the reuse of many middleware functions, allowing for a rapid prototyping of a DDS personality. In the next section, we assess our solution.

5 Assessment of our solution

In this section, we assess our implementation and compare our work with existing implementations of DDS.

5.1 Code reuse and benchmarks

We indicated in section 4.2 that DDS can reuse many functions inherited from existing components. We analyzed the final implementation code and evaluated the number of SLOCs (Significant Lines of Code, i.e. code statements only) of the components[3] for one non-optimized assembly.

[3]The measures have been done using SLOCCount from David A. Wheeler, `http://www.dwheeler.com/sloccount/`

This assembly directly reused MOMA, GIOP, the neutral core layer configured to support priority-based request processing. Table 1 summarizes our measurements.

Component	SLOCs	
*N*eutral Core	30417	PolyORB entities reused = 83 %
*G*IOP	8248	
*M*OMA	4503	
*D*DS	7009	Specific code = 17 %

Table 1. PolyORB's components used

We note that DDS personality represents less than 17% of code, this denotes an intensive code reuse. Besides, building this prototype of this new complex standard required less than four man-months.

This code reuse ratio demonstrates that PolyORB middleware components provide key building blocks, which clearly reduces the need to write large portions of code when implementing a new distribution model. This allows developers to quickly implement new distributions mechanisms and experiment with their semantics, and then focus on performance metrics to optimize memory footprint, time property of the middleware.

5.2 Usability assessment

In the previous section, we presented an assembly of components that form one DDS middleware. It reuses many components written in the context of other distribution models: RT-CORBA policies and protocol, MOMA messaging entities. The DDS personality combines these elements to support the DCPS layer. It provides the first open source implementation of DDS in Ada 95.

This code is not as-is optimized for any resource constrained DRE systems. From a performance point-of view, the behavior of this implementation is notionally equivalent to a RT-CORBA application for the request processing part, reusing its protocol and priority mechanisms, with application entities implementing the messaging distribution model. Message throughput is 2400 messages per seconds on average workstations running GNU/Linux.

PolyORB's middleware components can be tailored to reduce their semantics to the only required elements, e.g. protocol, type managements, memory policies, etc. Hence, the versatility of PolyORB enables the user to precisely adapt any of its core elements, from transport, protocol, core and messaging entities, without modifying the DDS personality. This provides a high level of tailorability.

Therefore, we propose a solution to the portability and adaptability issues that arise when using DDS in many heterogeneous application domains, enhancing the value added of the DDS development approach. The developer can meet specific deployment and resources constraints by adapting the middleware components while preserving the properties of its application based on one uniform API.

5.3 Related Work

DDS is a recent OMG specifications. Yet, it originates from long-term industrial projects that built a de facto standard for which many implementations exist.

nDDS by RTI is the originator of the DDS specifications, they propose nDDS as a real-time publish/subscribe middleware for many years. They contributed to the DDS specifications. nDDS is a closed-source implementation that supports multiple languages, it is field-proven and deployed in many industrial and military projects such as [8].

SPLICE DDS is another DDS middleware done by Thales and Prismtech. It is deployed in military projects, with a strong emphasis on performance and scalability [5].

It is difficult to evaluate these closed-sources projects. The main question is the level of code reuse and ease of adaptability provided compared to the footprint and efficient data throughput advertised. The latter are important for the particular application to builds, but may impede middleware usability on the long run.

TAO/DDS is a recent open source project from OCIthat proposes a C++ implementation of DDS 1.0 built around the ACE and TAO frameworks [15].

This implementation leverages existing elements from the ACE frameworks for low-level I/O routines : CDR marshallers, pluggable transport framework, etc.

It also reuses some elements from TAO to avoid code duplication, and also enables some cooperation between CORBA and DDS domains. This feature is very similar to the notion of personalities advertised by our architecture : PolyORB can support simultaneously CORBA and DDS personalities, and enable interoperability between them.

Release 0.6 of TAO/DDS covers DCPS features, it incorporates $21,583$ SLOCS. This indicates TAO/DDS factors middleware functions at a lower conceptual level (e.g. protocol, request dispatching) than PolyORB which focuses on distribution mechanisms as a whole.

In this section, we assessed our solution, and compared it with other projects. PolyORB and its personalities demonstrate how separation of concerns enable the rapid prototyping of middleware, leveraging existing middleware components. This allows for optimizations at a latter stage. Other projects focus on features to be implemented instead of code reusability and tailorability. TAO provides a first view on code reuse using ACE, but is still at an early stage.

6 Conclusion

The OMG published the Data Distribution Service (DDS), a high-level description of a data-centric middle-

ware for DRE systems. Following, the MDA approach, A platform independent model defines DDS semantics. Considering the versatility of application requirements, we claim that DDS implementations should allow high tailorability while allowing application reuse.

PolyORB, our implementation of a schizophrenic middleware, provides an implementation framework for DDS, and a repository of middleware components ready for use. Its modular architecture built around the μBroker and the identification of canonical middleware functions allows the definition and implementation of a DDS API based on one highly tailorable architecture. This approach brings major benefits to middleware engineering.

First, the developer can carefully select, adapt or create middleware components and integrate them in the architecture and preserving the DDS API. This allow for a high-level of tailorability, without impeding application reusability in different contexts (hardware, deployment, etc.).

Second, this approach factors code across multiple distribution technologies. PolyORB supports DDS, but also CORBA, RT-CORBA, a MOM on top of a generic middleware core. This core can be formally modeled and verified, and has actually been verified for some configurations. This increases confidence in the middleware components reused in another context: functional properties are kept.

Hence, the adaptation work for a new technology can then focus on specific aspects, such as non-functional QoS policies. This reduces development cost while increasing the confidence in the infrastructure.

The construction of application-tailored middleware using DDS and PolyORB is iterative. The middleware is first drafted for the first tests of the application and then refined by selecting and extending the implementation capabilities. This approach allows the tuning of both the application and the middleware it relies on. *Application/Middleware co-design* is then possible. This is useful for developers to meet specific needs of a given DRE system.

Later work will consider the automation of the middleware adaptation work, defining the tools and methodology to assist the user in selecting or extending middleware components. The AADL ("Architecture, Analysis and Design Language", [3]) has been selected to describe the middleware's structure. We aim to use this description, combined with a set predefined routines and service libraries, to weave the configured middleware dedicated to a DRE system.

Acknowledgement The authors thank Suha Demir CAN for his valuable work when implementing the first release of the DDS personality for PolyORB.

References

[1] S. Baker. Middleware To Middleware. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, Sept. 2001.

[2] B. Dumant, F. Horn, F. D. Tran, and J.-B. Stefani. Jonathan: an open distributed processing environment in java. In *IFIP Int'l Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 175–190, Londres, 1998. Springer Verlag.

[3] H. Feiler, B. Lewis, and S. Vestal. The SAE Avionics Architecture Description Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering. In *RTAS 2003 Workshop on Model-Driven Embedded Systems*, May 2003.

[4] S. Gorappa, J. A. Colmenares, H. Jafarpour, and R. Klefstad. Tool-based Configuration of Real-time CORBA Middleware for Embedded Systems. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC'05)*, Seattle, USA, May 2005.

[5] Hans van't Hag. OMG - DDS Exploiting the Potential - Proven Suitability in the Naval Combat System Domain. In *Proceedings of the OMG Real-Time Embedded Systems Workshop*, Arlington, VA USA, July 2005.

[6] J. Hugues, L. Pautet, and F. Kordon. Contributions to middleware architectures to prototype distribution infrastructures. In *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03)*, San Diego, CA, USA, June 2003.

[7] J. Hugues, Y. Thierry-Mieg, F. Kordon, L. Pautet, S. Baarir, and T. Vergnaud. On the Formal Verification of Middleware Behavioral Properties. In *Proceedings of the 9th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'04)*, Linz, Austria, Sept. 2004. To be published in Electronic Notes in Computer Science.

[8] R. T. I. Inc. Customer Success : Lockheed Martin, 2005.

[9] F. Kordon and L. Pautet. Toward next-generation toward next-generation middleware? *IEEE Distributed Systems Online*, 5(1), 2005.

[10] OMG. Model Driven Architecture (MDA), Document number ormsc/2001-07-01. Technical report, OMG, 2001.

[11] OMG. *Overview and guide to OMG's architecture*. OMG, June 2003. OMG Technical Document formal/03-06-01.

[12] OMG. *Data Distribution Service for Real-time Systems Specification version 1.0*. OMG, Dec. 2004. OMG Technical Document.

[13] R. Rajkumar, M. Gagliardi, and L. Sha. The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation. In *Proceeding of the 1st IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, USA, May 1995.

[14] D. Schmidt, D. Levine, and S. Mungee. The design and performance of real-time object request brokers, 1998.

[15] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, Apr. 1998.

[16] SUN. Java Message Service (JMS), 1999.

[17] T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004 (RST'04)*, Palma de Mallorca, Spain, June 2004.