PolyORB: a schizophrenic middleware to build versatile reliable distributed applications

Thomas Vergnaud[1], Jérôme Hugues[1], Laurent Pautet[1], and Fabrice Kordon[2]

[1] GET-Télécom Paris – LTCI-UMR 5141 CNRS
46, rue Barrault, F-75634 Paris CEDEX 13, France
thomas.vergnaud@enst.fr, jerome.hugues@enst.fr, laurent.pautet@enst.fr
[2] 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. The development of real-time distributed applications requires middleware providing both *reliability* and *performance*. Middleware must be *adaptable* to meet application requirements and integrate legacy components. Current middleware provides only partial solutions to these issues. Moreover, they newer address all of them. Thus, a new generation of middleware is required. We have introduced the *schizophrenic middleware* concept as an integrated solution to build versatile reliable distributed applications. PolyORB, our implementation of schizophrenic middleware, supports various distribution models: CORBA (Common Object Request Broker Architecture), SOAP (Simple Object Access Protocol), DSA (Ada 95 Distributed System Annex), Message Passing (an adaptation of Java Message Service to Ada 95) and Web Server paradigm (close to what AWS offers). In this paper, we describe the implementation of PolyORB and provide a summary of our experience regarding the issues mentioned above.

1 Introduction

The term "middleware" designates a piece of software that eases both development and deployment of distributed applications. It handles basic functions to distribute applications in heterogeneous environments. Offered services are for example transport protocol, remote execution, addressing, data marshalling, etc.

Distributed systems use classical distribution paradigms like message passing (MP), remote subprogram call (RPC), distributed objects (DOC) or shared memory (DSM). The community has defined several distribution models which propose a combination of these paradigms. DOC can be implemented using CORBA or RMI (Remote Method Invocation). DSA (Distributed Systems Annex for Ada 95) provides several paradigms like RPC, DOC and DSM. Distribution models also lead to many variations. CORBA or DSA define asynchronous remote method invocation. Middleware classically implements one distribution model.

The choice for a distribution model or for a list of distribution mechanisms is driven by the application requirements; a correct (or incorrect) choice may dramatically impact design and performances [Mul93]. The growing demand for distribution functions in a wide range of systems (embedded, mobile or real-time systems) increases the multiplication of distribution models with some specific variations.

Each application comes with its specific needs and takes advantage of one distribution model. But all these applications (possibly based on heterogeneous distribution models) have to interoperate one with another. Middleware usually handles interoperability between heterogeneous hardware or operating systems. In the context of heterogeneous components interactions, middleware has to propose a new form of interoperability on which this paper focuses e.g. an interoperability between distribution models. This property is summarized as *Middleware to Middleware* interaction (M2M) [Bak01].

Designing from scratch specific middleware for a variant of distribution model, driven by particular application requirements, would be too expensive. A better approach consists of designing general middleware that can be *tailored* to the specific application needs. This saves both time and money.

Configurable or generic middleware provides a first efficient solution to tailor these distribution platforms so that they meet the application needs: TAO [SC97] can be configured to address real-time concerns, and Jonathan [DHTS98] can be personalized for various distribution models. But such middleware does not provide interoperability between existing components based on different distribution models. Yet this issue cannot be discarded since distributed systems now commonly reuse legacy components.

We have introduced the *schizophrenic middleware* concept [QPK01] as a global solution to both configurability to meet application requirements, adaptability of distribution mechanisms and interoperability between distribution models. It also provides support for execution determinism and formal verification. PolyORB [PQK$^+$01], our implementation of such middleware, is a proof of concept.

The aim of this paper is to describe the schizophrenic concepts and the associated architecture. We study the PolyORB implementation and demonstrate the viability of such middleware. We first give an overview of configurable and generic middleware, and of the interoperability issues. Then we introduce the schizophrenic middleware concept and the current status of PolyORB, and the distribution models it supports. Finally we study the performances of PolyORB, compared to other middleware solutions.

2 Middleware adaptability and interoperability: an overview

Few projects directly focus on the design of middleware. They define architectures and building blocks to facilitate middleware modification and adaptation. Very few others focus on interaction between systems developed with different distribution models.

In this section, we analyze the design principles of *configurable* and *generic* middleware. We describe several architectures which provide these properties. We also present the difficulties to enforce interoperability between different distribution models.

2.1 Configurable middleware

Configurable middleware enables an application to select actual components and specific run-time policies that address its requirements for a single distribution model.

TAO: *The ACE ORB* (TAO) is a free configurable ORB based on the *ACE* (Adaptive Communication Environment) communication and synchronization library. TAO supports CORBA real-time features. It controls the scheduler policy to enforce real-time properties or to satisfy Quality of Service requirements. It is highly dedicated to avionics, multimedia or simulation applications. TAO architecture is based on design patterns [GHJV94]. It can be configured with the appropriate components to address a specific application domain and to ensure performance, determinism or scalability properties. TAO provides an IDL (Interface Description Language) compiler that optimizes the generated stubs and skeletons depending on user requirements.

2.2 Generic middleware

Generic middleware extends the configurability concept in order to adapt middleware to the distribution model required by the application. It defines canonical components and architecture which are extended or *personalized* to support the given distribution model. In other words, a generic middleware is instantiated according to a distribution model to create a *personality*.

Quarterware: This C++ middleware [SSC98] defines a restricted set of components. These components may be extended or specialized to implement a specific distribution model. This process has been successfully applied to the CORBA, RMI and MPI (Message Passing Interface) models. Quarterware's components embody typical middleware functions specified as design patterns. Their specialization are supposed to be fast and efficient. Unfortunately, this middleware is not an open source software and it is not possible to evaluate its reuse code ratio or its performances.

Jonathan: This architecture emphasizes on a core system, Jonathan, and its *personalities*. Jonathan [DHTS98] is a Java framework of configurable components and abstract interfaces. Current implementation provides a CORBA personality (*David*), a RMI personality (*Jeremie*) and specialized personalities for multimedia systems.

Generic middleware can be instantiated to meet the application needs; however, the development of a new personality implies the engineering of a significant amount of code. For instance, as Jonathan is mostly based on abstract interfaces, personalities like David and Jeremie reuse only 10% of the generic code.

2.3 Interoperability between distribution models

Legacy components usually rely on multiple heterogeneous technologies. This includes heterogeneous distribution models. Hence, software reusability may rise interoperability issues, leading to M2M concern described in section 1.

Typical solutions involve middleware nodes communicating one with another through ad hoc gateways. CIAO [Qui99] provides static gateways between CORBA and DSA nodes; CORBAWeb [MGG96] follows the same principles by providing dynamic gateways between CORBA and Web clients. This approach requires a gateway between each pair of middleware paradigm.

Other solutions like *Java/HPC++* [BDV$^+$98] rely on the choice of a *common exchange protocol*. Then only one translator between each middleware paradigm and this common protocol is necessary.

These solutions are interesting: they isolate the components interaction issue. This allows the developer to reuse legacy components. However, the use of intermediate entities requires a significant translation work; it impacts the performances of the whole distributed system. This may not offer sufficient performances.

3 Schizophrenic middleware

Configurable and generic middleware ease middleware adaptation. However they lack flexibility: configurable middleware provides adaptation hooks to abide to application constraints; generic middleware is a solution to tailor to a distribution model. Interoperability between different middleware types is often dealt as a separate issue.

We claim that an architecture combining configurability, genericity but also interoperability capabilities into a common middleware architecture is sufficient to address both application and distribution model requirements. This requires an architecture that emphasizes on separation of concerns. We now present this architecture.

3.1 Decoupling functionalities

Schizophrenic middleware refines the definition and role of personalities introduced by Jonathan. It proposes *application-level* and *protocol-level* personalities, and a *Neutral Core Middleware* (NCM). The latter allows for the interaction between multiple personalities, as shown on figure 1.

Application personalities: They constitute the adaptation layer between application components and middleware through a dedicated API or code generator. They register application components within the NCM; and they interact with it to enable the exchange of requests between entities at the application-level.

Protocol personalities: They handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages transmitted through a communication channel.

Requests handled by a protocol personality originate from three sources. Application entities produce requests to be sent to other nodes of the distributed application and convey them to the protocol personality through an application personality and the NCM. Requests also come from other application nodes and are received by the protocol personality. At last, they also come from another protocol personality through the NCM: in this case the application node acts as a proxy performing protocol translation between third-party nodes.

The Neutral Core Middleware (NCM): It 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: this enables the selection and interaction of any combination of application and protocol personalities.

Figure 2 summarizes the different ways various personalities can exchange requests through the neutral core middleware. This naturally leads to interoperability: entities registered to an application personality are available to any client using a middleware

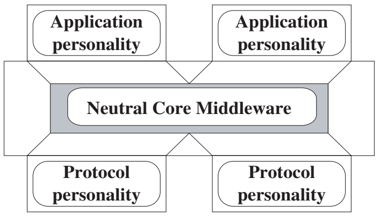

Fig. 1. Schizophrenic architecture

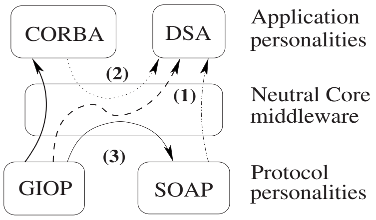

Fig. 2. PolyORB personalities interactions

for which the corresponding protocol personality exists (1), or to another collocated application or protocol personalities (2, 3); the NCM acts as a gateway between interoperating personalities. Hence, this architecture separates three main concerns in middleware: protocol-side, application-side and internals.

Personalities implement a specific aspect of a distribution model. The NCM enables the presence and interaction of multiple collocated application and protocol personalities within the same middleware instance, leading to "schizophrenia".

3.2 Services

Personalities and the NCM are built on top of seven basic services that embody key steps in client/server interactions (e.g. RPC or Distributed Objects). We now present these services through an example: the interaction between a DSA client and a CORBA server using the SOAP protocol (figure 3).

Each entity is given a unique identifier within the entire distributed application using the *addressing* service (1). This is used by the client to get a reference on a server entity. Then the NCM uses the *binding* service (2) and creates a binding object. It provides mechanisms to establish and maintain associations between interacting objects and the resources that support this interaction (e.g. a socket, a protocol stack).

Then request parameters are translated into a representation suitable for transmission over network, using the *representation* service (3). A *protocol* (4) is implemented for transmissions between the client and the server nodes, through the *transport* (5) service, which establishes a communication channel between the two nodes. Then the request is sent and unmarshalled by the server.

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

Middleware main loop handles the coordination of these different services and ensures correct propagation of data flow within the middleware node.

Middleware may also provide advanced services which are optional for some personalities. They implement facilities and high-level APIs that ease the development or deployment of a distributed application.

The *naming* service provides association between a reference on an entity and a symbolic name, e.g. CORBA COS Naming or the Ada Distributed System Annex inter-

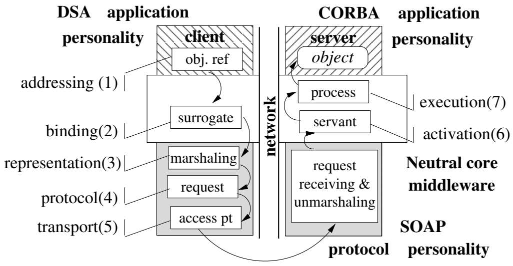

Fig. 3. Invocation request path

nal naming scheme. The *interface repository* services provide a metadata describing the interface of application entities and the types they define. This provides mechanisms to implement the CORBA Interface Repository. The *termination* service determines consensus on whether a distributed application has completed its task or not. Such a service is useful in a DSA implementation. The *shared data* service provides transparent access to data shared by different nodes in a distributed application. Such a service is also present in the DSA specification. The *synchronization* service provides mechanisms to coordinate actions of different nodes (e.g. distributed mutexes).

Then basic and advanced services are combined to build a middleware instance; this enables a precise adaptation to application requirements. We now describe our ongoing work on PolyORB, our implementation of schizophrenic middleware.

4 PolyORB: implementation of a schizophrenic middleware

PolyORB is our free software implementation of schizophrenic middleware, in Ada 95. It is released under the GNAT Modified GPL licence [PQK+01]. PolyORB is now stable software and entered industrialization process. In this section, we detail PolyORB implemented personalities and Neutral Core Middleware.

These implementations reuse components from GLADE [PT00], our implementation of the Distributed System Annex for GNAT; AdaBroker, our implementation of CORBA; AWS [Obr03] components for Web applications; and XML/Ada [Bri01].

4.1 Application personalities

PolyORB supports different distribution models, implemented as application personalities: Distributed Object with CORBA and DSA; Remote Procedure Call with DSA; Message Passing with MOMA; and Web applications with AWS.

CORBA: We reuse some elements implemented for the AdaBroker ORB to produce a CORBA-compliant application personality. This personality provides an IDL-to-Ada compiler that supports most IDL constructs, except for fixed-point numbers, abstract interfaces and object by value. It also provides a communication system that supports static and dynamic invocation and implements the Portable Object Adapter

(POA). Moreover, CORBA application personality provides some CORBA COS Services: COS Naming, COS Event, COS Time and the Interface Repository.

Distributed System Annex: The implementation of the DSA application personality was facilitated because of the know-how acquired during the development of GLADE. The implementation of this personality carried on the partial rewriting of the GNAT compiler so that PolyORB neutral core middleware and protocol personalities act as a communication system for the distributed system annex. The DSA personality supports Annex E specification. Some of the advanced services introduced in GLADE such as termination, bootstrap servers will be ported to PolyORB in a near future.

Message Passing: MOMA (Message Oriented Middleware for Ada) provides message passing mechanisms through an Ada 95 implementation of the well-known Sun's Java Message Service (JMS) [SUN99]. JMS is a standardized API for common message passing models; it keeps the distribution layer implementation-defined. We focused directly on MOM distribution logic and defined MOMA as an application personality. As for DSA, PolyORB NCM and protocol personalities act as a communication system. Our MOM architecture supports typical MOM primitives like 1-to-1 and 1-to-N message exchanges. Those are implemented using a *servant*-like pattern: clients invoke specific methods on MOMA objects to exchange messages.

Web Server: The Web Server personality provides the same API as the Ada Web Server project (AWS) [Obr03]. It allows for the implementation of web services, web server applications, or classical web pages. AWS-based servers allow the programmer to directly interact with incoming or outgoing HTTP (HyperText Transfer Protocol) and SOAP requests. We have implemented a *servant*-like pattern to enable servers creation; request manipulation is done through a wrapper on PolyORB's internals. The AWS personality provides the basic functionalities of the original AWS; other functionalities are facilities to ease data manipulation. They will be incorporated in future releases.

4.2 Protocol personalities

Protocol personalities provide support for data exchange. PolyORB supports GIOP (General Inter-Orb Protocol) for all-purpose requests, MIOP (Multicast Inter-Orb Protocol) for multicast and SOAP for Web Services.

GIOP: GIOP is the transport layer of the CORBA specifications. AdaBroker provided a basis for the implementation of GIOP. This personality implements GIOP versions from 1.0 to 1.2 along with the CDR (Common Data Representation) representation scheme to map data between the neutral core layer and CDR streams. GIOP is a generic protocol for which we provide several instances. IIOP (Internet Inter-Orb Protocol) supports synchronous request semantics over TCP/IP. MIOP [OMG03] instantiation of GIOP enables group communication over IP multicast. DIOP (Datagram Inter-Orb Protocol) relies on UDP/IP communications to transmit one-way requests only.

SOAP: SOAP protocol [W3C03] enables the exchange of structured and typed information between peers. It is a self-describing XML document [W3C00] that defines both its data and semantics. Basically, SOAP with HTTP bindings is used as a communication protocol for Web Services. PolyORB's SOAP personality reuses code from

AWS, and takes advantage from the XML self-describing properties to propose a direct mapping between SOAP and neutral requests.

4.3 Neutral Core Middleware

Neutral Core Middleware is the heart of PolyORB. It provides a basic implementation for most middleware services defined in section 3.2. Besides, it implements different APIs and patterns on top of which distribution facilities are built. Finally, it provides hooks to register and to configure application or protocol personalities.

These components are designed to maximize code reuse, performance and configurability. For instance, PolyORB tasking constructs can be adjusted to application requirements. It supports No Tasking, Full Tasking but also Ravenscar run-time [DB98]. Moreover, tasking policies may be defined to precisely control tasks allocation to process requests. It reuses notions introduced in GLADE and AdaBroker and define different policies: *thread pool*, *thread per session*, *thread per request*.

Moreover, the NCM eases to proof determinism of key middleware elements. Hence, the Ravenscar tasking run-time enables determinism of concurrency patterns. Static and dynamic perfect hash tables [DKM$^+$94] enable $O(1)$ lookup time in dictionaries.

5 Experiments and validation

This section illustrates PolyORB capabilities and efficiency to assess its usability as a distribution platform to develop complete applications. We first address its compliance to strict specifications; we then discuss code reuse and performance issues.

5.1 Interoperability & compliance to standards

Middleware specifications strictly prescribe the components on top of which a distributed application may be built, e.g. protocols, API. Strict compliance to these norms is required to ensure interoperability with other products.

We first verified compliance at the protocol level: transport of request must enable communication with other middleware.

CORBA/IIOP: We verified interoperability using the IIOP instantiation of GIOP with other CORBA-based middleware, C++ implementations: omniORB, TAO; Java implementations: Jonathan and OpenORB. Our tests verify data transfer consistency for various types as defined by the OMG IDL: simple, aggregate and complex ones.

CORBA/MIOP: We tested interoperability with TAO implementation of MIOP: messages were correctly broadcasted and then received by TAO and PolyORB nodes.

SOAP: We tested interoperability with AWS clients and servers built on top of AWS original implementation. We verified data transfer consistency.

Then, we verified compliance at the application level: implemented API or generated code must verify a defined semantics and run-time properties:

CORBA: The OMG defined in [OMG01] how CORBA specifications are to be translated into Ada 95. We implemented these recommendations and verified correctness on typical CORBA examples. Besides, the implementation of the COS[3] Naming,

[3] COS: Common Object Service

COS Event and the Interface Repository stressed our implementation. Let us note that this mapping is currently under revision; adjustments may be required.

DSA: We used the *Ada Conformity Assessment Authority* test suite to validate our implementation of the Distributed System Annex. We removed the conformity tests of obsoleted features like the compliance to `System.RPC` which is no longer normative in Ada0Y (paragraph E.4.20). We pass all tests but two. The missing tests are related to variable-length types, and will be added in a near future.

AWS & MOMA: These two personalities rely on specific and not normalized interfaces. We verified on selected examples that the application nodes behaved correctly.

These different tests demonstrate that PolyORB provides a stable implementation of various distribution models, compliant to industrial standards.

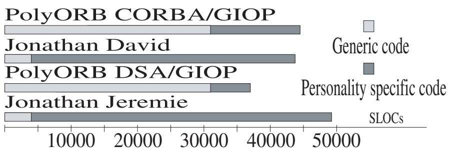

Fig. 4. Code reuse in PolyORB and Jonathan

Server \ Client	omniORB	TAO	PolyORB
omniORB	$1.382s$	$3.310s$	$1.869s$
TAO	N/A	$2.986s$	$2.498s$
TAO TP	N/A	$3.361s$	$3.865s$
PolyORB	$1.785s$	$3.576s$	$2.056s$
PolyORB TP	$3.340s$	$5.053s$	$3.602s$

Fig. 5. Execution time for $10,000$ requests

5.2 Code reuse

Code reuse ratio provides a measure of the reusable functional key components provided by middleware. Figure 4 shows a measure of the SLOCs present in comparable applications. We computed Source Lines Of Code (SLOCs) of the generic core and personalization specific code between PolyORB and Jonathan, for which source code is freely available. The measures have been done using SLOCCount [4].

We considered two distinct configurations: "ORB", using CORBA/GIOP personalities; and "RPC" comparing DSA/GIOP for PolyORB and RMI for Jonathan.

A raw analysis demonstrates that PolyORB NCM represents a significant part — more than 66 % — of the distribution infrastructure; Jonathan core and sets of abstract interfaces represent around 10 %. PolyORB's numbers demonstrate that the NCM provides key building blocks to implement middleware. This clearly reduces the need to write large portions of code when adapting PolyORB to meet the needs of an application. Hence, PolyORB facilitates the development of new personalities. This has been verified when implementing AWS, MIOP and MOMA: PolyORB provides generic abstractions to build different distribution models.

5.3 Code efficiency

PolyORB novel architecture enables great configurability, genericity and interoperability. These are interesting properties when developing or deploying a distributed application. However, middleware implementations should have acceptable performance as

[4] from David A. Wheeler, `http://www.dwheeler.com/sloccount/`

well. Thus, we benchmarked PolyORB against typical and configurable middleware to assess impact of its architecture on performances.

We compared the execution time for PolyORB configured with CORBA/GIOP personalities; and two very distinct C++ CORBA ORB: omniORB and TAO. omniORB (from AT&T Labs) is known for its strict compliance to CORBA specifications and its efficiency. TAO advertises its configurability and reliability properties.

PolyORB relies on dynamic mechanisms. In order to have coherent measurements sets, we tested against other dynamic clients and servers, i.e. CORBA Dynamic Interface Invocation (DII) clients and CORBA Dynamic Skeleton Interface (DSI) servers. We measure the time required to execute $10,000$ requests. Each request "echoes" an unsigned long parameter.

Figure 5 summarizes our measurements. omniORB uses its default settings, TAO and PolyORB denotes no tasking configuration of middleware, TAO Thread Pool (TP) and PolyORB TP are full tasking configurations of middleware, using a thread pool of four threads.

We note that TAO DSI cannot interoperate with omniORB DII clients, indicated by "N/A" in the table; this is a known bug. PolyORB competes with these two platforms. In thread pool mode, PolyORB has similar performance compared to TAO, better performance in mono tasking — nearly 30 %. omniORB is the fastest ORB, performance loss factor ranges from 1.3 to 2.4. The main explanation to this significant difference with omniORB is its over-optimized architecture that in return limits configurability.

5.4 Towards reliability of distribution middleware

Building reliable software is a complex task, more specifically when it comes to distributed applications. Middleware tends to be used as COTS[5] components in mission critical applications. However, the integration of COTS components raises new issues to ensure reliability or to certify applications [Bud03]. Hence, developer requires precise knowledge on component behavior and properties.

At the implementation level, PolyORB enforces determinism. The use of selected algorithms and patterns as well as of the Ravenscar profile (section 4.3) ensure determinism of the basic elements of the implementation.

Then we contemplated the formal verification of our architecture. Schizophrenic middleware provides a comprehensive description of its architecture built around different well-defined services. Middleware functions are delegated to specific services. This enables separate verification of its components.

In [HPK03], we proposed a roadmap to middleware verification. The middleware main loop coordinates all functions within a middleware node. We identified this loop as the most critical component of middleware. Then, we defined and modelled it using Petri Nets. This allows us to formally verify qualitative properties on the behavior of this component such as the bounds of buffers or the appropriate use of critical sections.

The other middleware services and components are less complex entities. They implement well known patterns, or simply manipulate data: they do not directly rely on constructions that may lead to faulty execution. We plan to gradually verify them. The

[5] COTS: Commercial Off-The-Shelf

combination of these different results is a first step towards formal verification of middleware. This provides inputs to assess properties and then to prove reliability.

PolyORB properties and performances make it a possible choice to develop distributed applications. PolyORB provides configurability and genericity which does not impede performance. Besides, PolyORB relies on strong engineering methods. The formal verifications of qualitative properties ensure middleware reliability. These are first steps to ensure the usability of PolyORB to build reliable distributed applications.

6 Conclusion

The diversity of application requirements leads to different distribution models, and then middleware implementations adapted to a specific context. Thus, distributed application engineering requires adaptable middleware to fit a wide range of applications. This paper discussed about adaptable middleware design and implementations.

First, we identified several requirements on middleware architecture: adaptability to application needs, interoperability between heterogeneous components, reliability.

Then, we discussed existing solutions. Configurable middleware allows for fine adaptation to application needs, but is restricted to a given distribution model. Generic middleware defines abstract interfaces that are to be instantiated to meet the specific needs of the application; however this implies engineering of a large portion of code for each instantiation. Interoperability between heterogeneous middleware is often addressed separately. This slows down the performances of the whole system.

We introduced *schizophrenic middleware* as a comprehensive solution to distributed applications engineering. Schizophrenic middleware solves both configurability, genericity and interoperability concerns. Schizophrenic middleware extends the concept of middleware personalities to permit several collocated personalities to inter-operate within the same middleware instance. We defined *application* personalities and *protocol* personalities, all bound to a *Neutral Core Middleware*.

Our implementation PolyORB, which is available under a free software license, demonstrates the validity of the schizophrenic concepts. We developed personalities for SOAP and protocols associated to GIOP. We also developed personalities for CORBA, DSA, Message Passing (MOMA) and Web (AWS) middleware types.

Finally, we analyzed PolyORB implementation. We detailed its compliance to industrial specifications, and discussed its interoperability with other middleware implementations. We discussed code reuse, and indicated how PolyORB design eases the implementation of new distribution models. PolyORB's benchmarks show that it competes with other well-known implementations, yet it offers more adaptation capabilities. We also discussed reliability issues: PolyORB relies on strict engineering and design methods that help to prove its reliability. We are currently modelling key elements of middleware to formally prove their reliability in specific use cases. This will provide more information on middleware reliability.

PolyORB current development focuses on stringent determinism and real-time properties of distribution middleware, as well as the implementation of real-time middleware specifications such as RT-CORBA.

References

[Bak01] S. Baker. Middleware to middleware. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, September 2001.

[BDV+98] F. Breg, S. Diwan, J. Villacis, J. Balasubramanian, E. Akman, and D. Gannon. Java/RMI performance and object model interoperability: Experiments with Java/HPC++ distributed components. In *Proceedings of Workshop on Java for High-Performance Network Computing*, pages 91–100, May 1998.

[Bri01] E. Briot. *XML/Ada: a full XML suite*, 2001.

[Bud03] T. J. Budden. Decision Point: Will Using a COTS Component Help or Hinder Your DO-178B Certification Effort. *STSC CrossTalk, The Journal of Defense Software Engineering*, November 2003.

[DB98] B. Dobbing and A. Burns. The Ravenscar tasking profile for high integrity real-time programs. In *Proceedings of SigAda'98*, Washington, DC, USA, November 1998.

[DHTS98] B. Dumant, F. Horn, F. Dang Tran, and J-B. Stefani. Jonathan: an open distributed processing environment in java. In *Proceedings of the IFIP International Conference on Distribut ed Systems Platforms and Open Distributed Processing*, 1998.

[DKM+94] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[HPK03] J. Hugues, L. Pautet, and F. Kordon. Refining middleware functions for verification purpose. In *Monterey Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation*, Chicago, IL, USA, September 2003.

[MGG96] P. Merle, C. Gransart, and J-M. Geib. CORBAWeb: A generic object navigator. *Computer Networks and ISDN Systems*, 28(7-11):1269–1281, 1996.

[Mul93] S. Mullender. *Distributed Systems*. ACM, 1993.

[Obr03] P. Obry. Ada Web Server (AWS) 1.3, 2003.

[OMG01] OMG. *Ada Language Mapping Specification, v1.2*. OMG, October 2001. OMG Technical Document formal/2001-10-42.

[OMG03] OMG. *unreliable Multicast InterORB Protocol specification*. OMG, January 2003. OMG Technical Document ptc/03-01-11.

[PQK+01] L. Pautet, T. Quinot, F. Kordon, J. Hugues, and T. Vergnaud et al. Polyorb, 2001. `http://libre.act-europe.fr`.

[PT00] Laurent Pautet and Samuel Tardieu. GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems. In *Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00)*, Newport Beach, California, USA, June 2000.

[QPK01] T. Quinot, L. Pautet, and F. Kordon. Architecture for a reuseable object-oriented polymorphic middleware. In *Proceedings of PDPTA'01*, Las Vegas, USA, June 2001.

[Qui99] T. Quinot. CIAO: Opening the Ada 95 distributed systems annex to CORBA clients. In *Ada France 1999*, Brest, France, September 1999.

[SC97] D. Schmidt and Christ Cleeland. Applying patterns to develop extensible and maintainable ORB midd leware. *Communications of the ACM, CACM*, 40(12), 1997.

[SSC98] A. Singhai, A. Sane, and R. Campbell. Quarterware for Middleware. In *Proceedings of ICDCS'98*. IEEE, May 1998.

[SUN99] SUN. Java Message Service (JMS), 1999.

[W3C00] W3C. *Extensible Markup Language (XML) 1.0*, 2000. W3C recommandation.

[W3C03] W3C. *Simple Object Access Protocol (SOAP) 1.2: primer*, june 2003. W3C recommandation.