Contributions to middleware architectures to prototype distribution infrastructures

Jérôme Hugues, Laurent Pautet
{hugues, pautet}@enst.fr
École Nationale Supérieure des Télécommunications
CS & Networks Department
46, rue Barrault
F-75634 Paris CEDEX 13, France

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

***Abstract*— Distributed applications require specific middleware support for semantics and run-time constraints for a wide range of hardware or software configurations. However, their full specifications and existing implementations show they share functional notions and run-time mechanisms. Thus, distribution infrastructures could be prototyped from a given set of middleware components. Generic middleware proposes patterns to describe distribution models; configurable middleware constructs to abide to run-time constraints. We have introduced the *schizophrenic middleware* concept as a comprehensive solution to rapidly implement different distribution models. PolyORB, our implementation of a schizophrenic middleware, supports CORBA, SOAP, the Ada 95 Distributed System Annex and Message Oriented Middleware distribution models. In this paper, we describe existing generic and configurable middleware; we introduce PolyORB's key concepts and design; then we compare our platform design to existing generic and configurable middleware, and discuss their respective use to prototype specific distribution models.**

I. Introduction

Middleware provides implementation guidelines as well as frameworks to ease the development of large heterogeneous distributed systems; thus they are commonly accepted development platforms. Since there is no "one size fits all" architecture, choosing the adequate middleware in its most appropriate configuration is a key design point that may dramatically impact the design and performance of an application.

Consequently applications need to rapidly tailor middleware to the specific *distribution model* they require. A distribution model is defined by the combination of distribution mechanisms made available to the application. Common examples of such mechanisms are Remote Procedure Call (RPC), Distributed Objects or Message Passing. Thereafter a distribution infrastructure or middleware refers to software that supports one (or several) distribution model.

Besides, multiplicity in distribution models, each of which introduces its own semantics, API or protocol, creates a new layer of heterogeneity among distribution infrastructures. This leads to the *Middleware to Middleware* problematic [1]: the limitation on interactions between application components built around heterogeneous middleware and their reuse. Thus prototyping processes should also address the construction of new applications from various existing components.

Rapid System Prototyping [2] provides a definition canvas to express implementation and maintenance processes of complex applications. In this paper we show how various architectures may address the emerging need to rapidly prototype and deploy dedicated middleware.

A first solution is to reuse or adapt existing software components: recurring middleware functionalities can be factored out to define a general distribution framework or generic middleware. Their complete instantiation provides a single distribution model, called a *personality*. Thus, interoperability issues are not addressed by this approach. This restricts prototyping capabilities to a limited set of distribution models and reduce components reuse.

We have introduced the *schizophrenic middleware* concept [3] as a global solution to both distribution models interoperability and prototyping. Schizophrenic middleware extends generic middleware to allow the simultaneous support of multiple interacting personalities within the same middleware instance. Its modular design expresses both concerns for genericity of middleware components and their interaction, enabling interoperability between distribution models. PolyORB[1], our implementation of a schizophrenic middleware, is a proof of concept. It provides CORBA [4], SOAP [5], a Message Oriented Middleware (MOMA) and the Ada 95 Distributed System Annex [6] personalities. Current implementation status demonstrates interoperability between these heterogeneous distribution models (see [7] and [8]).

In this paper, we assess the use of our middleware PolyORB to prototype distribution infrastructures and the effective results we achieved. We propose a new architecture expressing genericity, configurability and interoperability requirements. The implementation of several personalities confirms our design choice: schizophrenic middleware, and more specifically

[1] available at *http://libre.act-europe.fr*

PolyORB, helps in the rapid design and implementation of distribution models.

We first give an overview of generic and configurable middleware. We then introduce the schizophrenic middleware concept and the current status of PolyORB; and demonstrate how we prototyped one PolyORB personality. We then compare the design of significant middleware architectures, focusing on their capabilities to prototype a specific distribution model.

II. Configurable and Generic middleware

An analysis of middleware implementations shows they rely on similar abstractions and run-time mechanisms to provide their services. Several projects focused on distribution infrastructure design, leading to the definition of *configurable* and *generic* middleware: middleware that expose constructs and concepts to enable their modification and adaptation.

A. Configurable middleware

Configurable middleware enables an application to select the actual components and specific run-time policies to address its requirements. We present two of them: TAO, The ACE ORB, and GLADE, the first validated implementation of the Distributed Systems Annex of Ada 95.

1) TAO: The ACE ORB (TAO) [9] is a free, scalable and configurable ORB based on the *ACE* communication and synchronization library. TAO is highly dedicated to avionic, multimedia or simulation applications. TAO architecture is based on design patterns; and can therefore be configured with the appropriate components to address a specific application domain and to ensure performance, determinism or scalability properties. Moreover, TAO provides an IDL compiler that optimize the generated stubs and skeletons depending on user requirements; but also control on the scheduler policy in order to enforce real-time properties or to satisfy Quality of Service (QoS) requirements. A typical example is the use of minimal perfect hashing functions to allow the ORB to handle incoming requests in a constant time [10].

2) GLADE: The normalized Ada 95 distribution model is a subset of the classical language entities. It provides distribution capabilities to some language constructs and adds support for remote subprograms, as well as remote or shared objects. As the language semantics is preserved, DSA users can design, implement and test their applications in a non-distributed environment, and then smoothly switch to a distributed deployment.

GLADE includes the deployment tool GNATDIST [11]. It provides a comprehensive framework integrated to the GNAT Ada 95 front-end for GCC and its compilation tool set. This enables the end user to control middleware configuration. GNATDIST description language eases the mapping of the distributed application components onto logical nodes; the parameterization of the communication subsystem or the resource allocation and tasking policies.

B. Generic middleware

Generic middleware extends the configurability concept. Middleware implementations have similar design, thus a specific distribution model may be built around canonical elements using a functionality-oriented approach; these elements are then adapted to conform to a specific distribution model during a *personalization* process.

Several projects demonstrate that middleware functionalities can be described as a set of generic services, independent from any distribution model; and propose architectures based on a set of abstract interfaces. A personality is then defined as the combination of concrete modules that implement these interfaces and provide access to generic middleware services.

1) UIC: The Universal Interoperable Core [12] provides only abstract interfaces. Their implementation provides control over the distribution model; specific implementations of CORBA client-side mechanisms have been proposed for hand-held devices, with hard constraints on memory resources.

2) Quarterware: This middleware [13] proposes a similar approach, extended to the CORBA, RMI and MPI models. These models have been implemented using a restricted set of components that can be extended to implement a specific model; or specialized for optimization and high-performance.

3) Jonathan: Its architecture [14] emphasizes on *personalities* that are adaptations of the core system Jonathan: a framework of configurable components and abstract interfaces. Dedicated instantiations of Jonathan provides a CORBA personality (*David*), a RMI personality (*Jeremie*) or specialized personalities for multimedia systems.

C. Needs for prototyping

These various architectures demonstrate that middleware can be described by a set of canonical elements: design patterns or abstract interfaces. This allows for the adaptation of their distribution model to user requirements. Hence they are solutions to prototype specific distribution models.

However the underlying prototyping process is not clearly specified and remain complex: it requires the implementation of most of the middleware functions. This process is also expensive in time or complexity. Moreover, the result of the personalization process is a dedicated monolithic middleware; this limits reuse of the prototyped elements. So, such approaches are only partial solutions to prototyping of distribution infrastructure. This point is discussed further in section V-C.

III. SCHIZOPHRENIC MIDDLEWARE

Configurable and generic middleware ease middleware adaptation; they make one step towards distribution infrastructures prototyping. However they lack flexibility: configurable middleware provides adaptation hooks to abide to application constraints; generic middleware a solution to tailor a distribution model.

We claim that a flexible architecture combining configurability, genericity but also interoperability capabilities into a common middleware architecture is required to fully prototype a distribution infrastructure and to address both application and distribution model requirements. This requires an architecture that emphasizes on separation of concerns.

Compilers theory describes a flexible architecture, separating machine code generation from source code: a front-end module analyzes source code; a back-end assembles machine code; both of them interact using different neutral representations of the code and the interface of an intermediate layer. Projects like the free software GNU Compiler Collection (GCC)[2] clearly demonstrates component reuse capabilities while providing support for multiple languages and targets.

Similarly, we separate concerns between a distribution model API and protocol, and their implementation related mechanisms; and propose an original middleware architecture [7]. We now present this architecture, and the design of our implementation: PolyORB.

A. Decoupling middleware set functionalities

Schizophrenic middleware refines the definition and role of personalities, and introduces *application-level*, *protocol-level* personalities and a Neutral Core Middleware. The latter provides support for interaction between multiple personalities (figure 1).

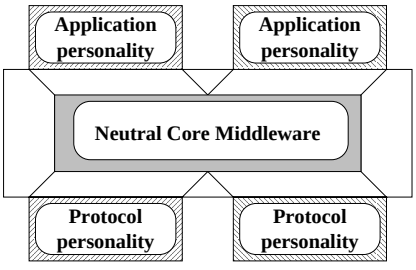

Fig. 1. Schizophrenic architecture

[2]Free software compiler front-ends and back-ends available at *http://gcc.gnu.org*

1) Application personalities: they constitute the adaptation layer between application components and middleware through a dedicated API or code generator; they provide services similar to those provided by a compiler front-end: translation of high-level constructs into simpler one. They provide APIs to register application components with the core middleware; they interact with the core layer in order to allow the exchange of requests between entities.

- On the client side, they map requests made by client components from their personality-specific representation to a personality-independent one. This neutral representation is then processed by the Neutral Core Middleware; results are translated back from neutral to personality-specific form.
- On the servant side, they receive requests for local objects from the core middleware, assign them to actual objects for evaluation, and return results.

2) Protocol personalities: they handle the mapping of personality-neutral requests (representing interactions between application entities) onto messages exchanged through a chosen communication network and protocol; similarly to a compiler back-end which transforms intermediate code representation into low level mnemonics. Requests can be received either from application entities (through an application personality and the neutral core) or from another node of the distributed application. They can also be received from another protocol personality: in this case the application node acts as a proxy performing protocol translation between third-party nodes.

3) 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 of any combination of application and/or protocol personalities; as the GCC compiler allows the selection of any given pair of front-end/back-end.

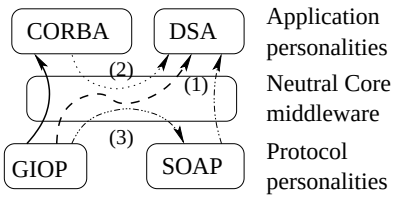

Fig. 2. Interaction between PolyORB's personalities

Figure 2 summarizes the different ways various PolyORB's 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 for which the corresponding protocol personality exists (1), or to another coexisting application or protocol personalities (2, 3); the NCM acts as a gateway between inter-operating personalities.

B. PolyORB building blocks

Schizophrenic middleware requires a flexible implementation and the identification of the functionalities involved in request processing to ease the prototyping of new personalities and their interaction.

1) Key patterns: Design patterns are used to facilitate code maintenance and readability. TAO has demonstrated how design patterns can be powerful tools to implement middleware [9]. We use patterns `Facade`, `Reactor` and configuration patterns for the real-time profile defined in our previous projects GLADE [15]. We also defined specific patterns to allow dynamic extension and specialization of objects: `Annotation` adding external informations to objects; `Component` adding new method to objects.

To allow a greater flexibility, Neutral Core Middleware (NCM) is built around dynamic constructs: invocation and implementation of the registered servants are based on CORBA Dynamic Invocation Interface (DII) and Dynamic Skeleton Invocation (DSI) models; exchanged data are mapped onto an all-purpose self-defined type, analog to CORBA Any's.

2) Basic services: PolyORB personalities and NCM are built on top of seven basic services embodying client/server interactions found in the RPC or ORB distribution models. A more complete description can be found in [3].

- **Addressing** Each entity is given a unique identifier within the entire distributed application.
- **Binding** Middleware establishes and maintains associations between interacting objects and resources allowing this interaction (e.g. a socket, a protocol stack). This service is inspired in part by the ODP binding [16].
- **Representation** Request parameters must be translated into a representation suitable for transmission over network.
- **Protocol** Middleware implements a protocol for the transmission of requests amongst nodes.
- **Transport** A communication channel is established between a node and an object to transmit requests.
- **Activation** Middleware ensures that a concrete entity implementing objects is available to execute the request.
- **Dispatching** Middleware assigns execution resource to process every incoming request.

Figure 3 illustrates how these different services cooperate to transmit one request from one personality to another when a DSA application interacts with a CORBA object using SOAP protocol. The client gets a reference on the object (1); the core middleware creates a binding object (2); a dynamic gateway to the CORBA object through which the client communicates; the request is then formatted and sent to the remote node (3, 4 and 5); upon reception, remote node middleware ensures that a concrete entity implementing the object is available to execute the request (6) and assigns execution resources (7).

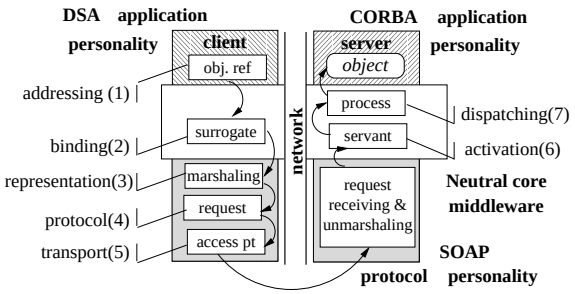

Fig. 3. Invocation request path

3) Advanced services: They provide solutions to distribution higher level problems and abstractions on top of which personalities mandatory functionalities can be implemented.

- **Naming** services provide association between entities references and symbolic names, e.g. JNDI API using an external naming service, or the Ada 95 Distributed System Annex (DSA) internal naming scheme.
- **Termination** services determine consensus on whether a distributed application has completed its task or not. Such a service is useful in a DSA implementation.
- **Shared Data** services provide transparent access to data shared by different nodes in a distributed application. Such a service is also present in the DSA specification.
- **Synchronization** services provide mechanisms to coordinate actions of different nodes (e.g. distributed mutexes).
- **Interface repository** services provide a database describing the interface of application entities (e.g. the set of interactions that they support), and the types of their parameters. It thus provides mechanisms to implement the CORBA Interface Repository.

The composition of these basic and advanced services allows for the implementation of different distribution models. We now detail how the use of these services in PolyORB's architecture enables prototyping capabilities.

IV. Prototyping a distribution model

We identified different services that provide foundations to define a middleware architecture geared towards distribution prototyping. In this section we highlight : facilities PolyORB provides to ease components reuse; its services combination to prototype MOMA, the Message Oriented Middleware personality for PolyORB, and an assessment of components reuse in implemented personalities.

A. PolyORB services

PolyORB's services propose a canonical view of the mechanisms within distributed applications and an implementation canvas for distribution models.

Neutral Core Middleware (NCM) provides an implementation of all services, which can be reused or adapted when implementing personalities. Moreover, the addition or specialization of services directly within NCM is encouraged to increase code reuse and to provide facilities for personalities. Hence, we propose either minimal services implementation as for protocol-related services, or complete and detailed for key mechanisms (PolyORB proposes different tasking runtime and object activation policies).

Besides, composition of personalities within the same middleware instance allows interoperability between different distribution models: the NCM acts as a dynamic gateway (see [7] for more details). This enables the reuse of existing legacy applications into new distributed applications at a limited cost. PolyORB directly provides gateway mechanisms between the various personalities. The personalities we implemented demonstrate full interoperability and can be deployed in the various scenarios presented in figure 2.

B. Prototyping a PolyORB's personality: MOMA

Message Oriented Middleware (MOM) provides message passing mechanisms between client nodes similar to e-mail or newsgroups; they support different delivery policies, persistence capabilities ... Typical MOM such as WebSphere MQ are widely used in large scale information systems [17].

We propose Message Oriented Middleware for Ada (MOMA) as a new PolyORB personality, it is an Ada 95 implementation of the Sun's Java Message Service API (JMS) [18]. JMS provides a standardized API for common message passing models; and a complete MOM specification for the Java platform. It precisely describes the different steps involved in MOM messages life-cycle when creating, sending, receiving, reading or destroying messages. Yet JMS only specifies an API, it does not address the underlying required layers of any distributed infrastructure such as transport protocols, data representation, etc. This is delegated to the *JMS Provider:* a distribution system on top of which the API is implemented. This decoupling between API and distribution infrastructure enhances separation of concerns and allow the prototyping of a wide range of provider.

JMS model is organized around different collaborating objects to exchange messages: producers, consumers, pools, and configuration objects: connections and sessions. We now describe the mapping this model to PolyORB's architecture.

Mapping MOMA functionalities to PolyORB: JMS API provides object primitives to allow clients to interact with message pools through requests (e.g. to post and receive messages); a provider supports distribution functions for client/server communications, and implementation of message pools. The MOMA provider is the combination of of two personalities and the core middleware:

- *application personality* implements MOMA objects (producers, consumers, ...) and client API.
- *protocol personality* addresses transport mechanisms.

Application personality: Interactions between a client and a message pool are similar to method calls on remote objects in a distributed application. We thus defined an application personality implementing message transport mechanisms based on a *servant*-like pattern: clients invoke specific methods to exchange messages on MOMA objects, which are implemented as PolyORB's servant remotely available.

Protocol personality: We focus on the application level MOM functionalities. Thus we only require a protocol personality that provides the mechanisms required to transport requests between a MOM client and a message pool. Any of PolyORB's existing protocol personalities is one candidate; we reuse the GIOP protocol personality in our test configurations.

Our MOM architecture built on top of PolyORB's Neutral Core Middleware (NCM) provides the minimal set of functionalities to support MOM primitives. The core layer and existing personalities provided support for distribution mechanisms, from protocol to request broker and object activation. Hence, we focused directly on MOM distribution logic and the implementation of the different required servants. Our design and the previous work done on PolyORB infrastructure clearly eased and accelerated the prototyping of this personality.

C. Prototyping middleware with PolyORB

In this section we summarize the lessons learnt when prototyping different distribution models as PolyORB personalities.

1) Application personalities: They provide support for a distribution model API or mechanism to applications components; PolyORB now proposes three different application personalities based either on code generation only (DSA), an API only (MOMA) or both (CORBA).

- DSA: The Ada 95 Distributed system annex proposes a RPC-like distribution model; thus it can be mapped onto PolyORB's interaction model and basic services. As for the CORBA application personality, most of its entities are directly mapped onto PolyORB entities. Code generation creates the appropriate calls to PolyORB NCM.
- MOMA: We showed in the previous section how this personality reuses all of NCM and protocol personality services to provide a MOM API.
- CORBA: PolyORB invocation model is notionaly equivalent to CORBA DSI/DII mechanisms, the *activation* service and the CORBA Portable Object Adapter have the same role. Hence, CORBA mapping to PolyORB internal structure is straight forward. Most of CORBA entities such as Objects, POA were directly wrapped on PolyORB own entities in both the generated code from an IDL and the CORBA API.

Creating an application personality implies to map a specific API and semantics onto PolyORB's invocation model presented in III-B.2. This model is versatile enough to express the various distribution models we considered. Basic services allow a rapid prototyping of distribution model internals, advanced services provide a canvas to design higher level functionalities that implements advanced features of a distribution model. This clearly eases the rapid prototyping process of infrastructure that supports a given distribution model.

2) Protocol personalities: They provide support for a given data exchange mechanism. PolyORB proposes GIOP and SOAP protocols; and Simple Request Protocol (SRP), a protocol devised to test simple invocations on servants. SRP allows the developer to test application personalities request handling: the user can forge its own request by hand with an HTTP-like syntax and send them to the servant directly using a `TELNET` connection.

Each protocol strictly specifies its semantics and representation data stream, hence the corresponding PolyORB's protocol personalities have to provide dedicated implementation of the *protocol*, *representation* and *binding* services. Code reuse possibilities are reduced; but one can notice that a representation or protocol are described by a normalized description or state machine. A protocol personality implementation is guided by these precise descriptions.

V. Comparing middleware architectures

We have presented different middleware architectures, putting the emphasis on their potential use as prototyping architecture to build distributed applications. In this section we provide information on PolyORB effective code reuse and efficiency compared to other significant architectures. We then discuss the use of configurable, generic and schizophrenic middleware to prototype distribution models.

A. Code reuse

Code reuse ratio provides a measure of the reusable functional key components provided by middleware. We compared Source Lines Of Code (SLOCs) of the generic core and personalization specific code between PolyORB and Jonathan, for which source code is freely available.

Figure 4 provides a measure of the SLOCs[3] present in comparable applications built using PolyORB and Jonathan in two distinct configuration: "ORB", using CORBA/GIOP personalities; and "RPC" comparing DSA and RMI personalities. A raw analysis demonstrates that PolyORB's neutral core layer represent a significant part - more than 75 % - of the distribution infrastructure; Jonathan core and sets of abstract interfaces represents around 10 %. Code reuse ratio

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

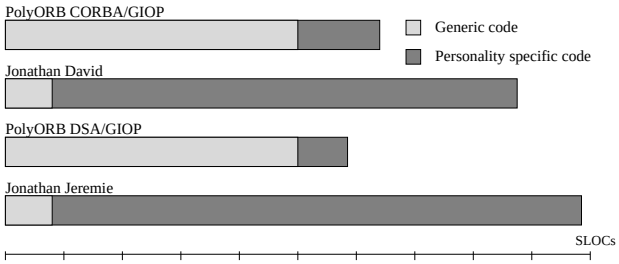

Fig. 4. Code reuse in PolyORB and Jonathan

demonstrate that PolyORB Neutral Core Middleware provides key building blocks, which clearly reduces the need to write large portions of code when prototyping a distribution model.

B. Code efficiency

Prototypes may serve as a first-step implementation towards production code; hence they should have correct performance. We compared time execution for the PolyORB's CORBA/GIOP configuration to the configurable CORBA ORB omniORB from AT&T Labs. omniORB is known for its strict compliance to the CORBA specification and its efficiency. As PolyORB relies on dynamic invocation, we tested against DII clients and DSI servers to have coherent measurements sets.

First measurements demonstrate that PolyORB and omniORB DII clients have similar performances; PolyORB servers are less efficient of a factor from 1.5 to 3 given its tasking policy. The current implementation has not been designed with optimization in mind, but these results are interesting. They show that distribution models prototyped with PolyORB have acceptable performance compared to "off-the-shelf" optimized middleware.

C. Prototyping facilities

Configurable, generic and schizophrenic middleware architectures rely on different architectural models that enable their adaptation. They have expected properties that can help in the choice of middleware to prototype distribution infrastructures. Thus they provide different levels of functionalities to prototype a distribution model using their architecture:

1) configurable middleware: they provide a predefined set of configuration hooks. They are thus easy to deploy. However the user has only access to a limited range of configurations, limiting prototyping capabilities. Moreover the addition of new configurable components is not directly possible and requires modifications to the middleware architecture.

2) generic middleware: they provide complete control over their architecture to allow dedicated implementations of a distribution model when concretizing their interfaces. However the personalization process may be complex and time consuming. The steps to define an early prototype up to a complete distribution model are not specified.

3) schizophrenic middleware: as for generic middleware, they provide complete control over their architecture. Personalities are implemented using either existing services, or implementing specific ones. This reutilization capability eases development process of early prototype, without significantly compromising performance. This prototype can then be refined rewriting or adapting some services in a later stage.

Schizophrenic middleware take advantage of both configurable and generic middleware to propose a global canvas to prototype distribution models at different stages of the application development process: from early prototypes up to the final release.

VI. Conclusion

Diversity in distribution models, each of which has specific configuration or heterogeneous distribution mechanisms, points the needs for rapid prototyping processes of distribution models. We identified key elements in this process: the adaptation of middleware to user and specific distribution model requirements; interoperability to enable components reuse. However existing adaptable middleware architectures do not provide these elements.

Configurable middleware enables the developer to choose components to tailor its distribution model for user requirements, but provides no control over the distribution model.

Generic middleware allows for the complete prototyping of distribution models from a set of abstract interfaces during the *personalization* process. However this implies the complete implementation of large components, which facilitate neither code reuse nor interoperability.

We introduced *Schizophrenic middleware* to address these key elements. It encapsulates configurability, genericity and interoperability concerns in a simple yet powerful way; allowing both code reuse and tailoring given developer needs. Schizophrenic middleware extends the concept of middleware personalities to allow several personalities to simultaneously operate within the same middleware instance. It also enables interoperability between distribution models.

Our implementation PolyORB, which is available under a free software license, demonstrates how the separation of concerns in the definition and implementation of a given set of generic services can provide implementation blocks. Then these blocks can directly be reused or specialized to implement 'off the shell' distribution models, providing a canvas to develop early prototypes up to complete distribution models.

The main characteristic of a schizophrenic middleware is to separate specific code in two separate parts: *application* personalities and *protocol* personalities which implementation relies on a Neutral Core Middleware. In terms of prototyping, this has two majors advantages: *1)* only a limited amount of code has to be written when implementing a new distribution infrastructure, *2)* any protocol personality may be associated with any application personality to create an original configuration.

So, PolyORB provides a solution for the rapid prototyping of distribution infrastructure, focusing only on the high-level services they require. The performances of the resulting middleware demonstrates that this solution is acceptable, though it requires optimizations at later stages.

We now contemplate investigating deeper middleware services configuration and performance. First we want to allow the user to control efficiently the code actually executed and adapt it to runtime constraints such as real time or mobility. Then we plan to devise a full prototyping process of distribution infrastructures using our architecture.

References

[1] S. Baker, "Middleware to middleware," in *Proceedings of the 3rd Int'l Symposium on Distributed Objects and Applications (DOA'01)*, Sept. 2001.

[2] F. Kordon and Luqi, "An introduction to rapid system prototyping," in *IEEE Transaction on Software Engineering Engineering*, Sept. 2002.

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

[4] OMG, *The Common Object Request Broker: Architecture and Specification, revision 2.2*. OMG, Feb. 1998.

[5] *Simple Object Access Protocol (SOAP) 1.1* , W3C, May 2000.

[6] ISO, *Information Technology – Programming Languages – Ada*. ISO, Feb. 1995, ISO/IEC/ANSI 8652:1995.

[7] T. Quinot, F. Kordon, and L. Pautet, "From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models," in *Proceedings of the 3rd Int'l Symposium on Distributed Objects and Applications (DOA'01)*, Sept. 2001.

[8] J. Hugues, F. Kordon, L. Pautet, and T. Quinot, "A case study of middleware to middleware: Mom and orb interoperability," in *Proceedings of the 4th International Symposium on Distributed Objects and Applications (DOA'02)*. Irvine, CA, USA: University of California, Irvine, Oct. 2002.

[9] D. Schmidt and C. Cleeland, "Applying patterns to develop extensible and maintainable ORB midd leware," *Communications of the ACM, CACM*, vol. 40, no. 12, 1997.

[10] D. C. Schmidt, "Gperf: A perfect hash function generator," in *Proceedings of the 2 nd C++ Conference*, San Francisco, California, Apr. 1990.

[11] Y. Kermarrec, L. Nana, and L. Pautet, "GNATDIST: A configuration language for distributed ada 95 applications," in *Proceedings of TRI-Ada'96*. ACM Press, 1996, pp. 63–72. [Online]. Available: http://www.infres.enst.fr/~pautet/papers/pautet96gnatdist.ps

[12] M. Román, F. Kon, and R. H. Campbell, "Reflective middleware: From your desk to your hand," *IEEE Distributed Systems Online*, vol. 2, no. 5, 2001.

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

[14] B. Dumant, F. Horn, F. D. 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.

[15] L. Pautet and S. Tardieu, "GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems," in *Proceedings of the 3rd IEEE Int'l Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'00)*. Newport Beach, California, USA: IEEE Press, June 2000.

[16] ODP, "ODP Reference Model: overview," 1995, ITU-T -- ISO/IEC Recommendation X.901 -- International Standard 10746-1.

[17] G. Banavar, T. Chandra, R. Strom, and D. Sturman, “A case for message oriented middleware,” in *International Symposium on Distributed Computing*, 1999, pp. 1–18.
[18] SUN, “Java Message Service (JMS),” 1999.