Architecture for a reuseable object-oriented polymorphic middleware

Thomas QUINOT, Fabrice KORDON, Laurent PAUTET

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

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

Using a middleware when designing a new distributed application allows portability across numerous software and hardware architectures, but introduces a new layer of potential incompatibilities due to the existence of various middleware standards.

This paper presents the architecture of DROOPI *(Distributed Reusable Object-Oriented Polymorphic Infrastructure). This project aims at defining and implementing a modular middleware emphasizing software reuse, customizability towards environment and application requirements, and transparent interoperability through the support of multiple middleware personalities. Some of these capabilities may be found in similar projects. Our objective is to bring all these properties in a single environment, enabling communication between various components using different middleware standards.*

1 Introduction

The emergence of distributed applications has raised the need for portability across numerous software and hardware architectures. A way to address this problem is to use a middleware when designing a new distributed application. However, this solution introduces a new layer of potential incompatibilities: various middleware standards have been defined, which use different communication protocols. Examples of such standards include CORBA, the Ada 95 DSA (Distributed Systems Annex) and Java/RMI. These incompatibilities seriously impair the reusability of distributed application components, which remain isolated in a given environment.

This interoperability problem may be solve case by case using specific gateways between two components. However, this approach does not promote reusability of middleware code, and thus fails to scale to applications in the large. Multi-personality middlewares have emerged from the observation that much of a middleware's behaviour is independent of the personality it assumes. For example, the generic middleware components of the Jonathan project [5], can be customized for a given middleware strategy (CORBA or Java/RMI). Similarly, Quarterware [11] can endorse several personalities and provides the core facilities of other existing middlewares: CORBA, Java/RMI and MPI (a middleware based on the message-passing distribution paradigm). To resolve the problem of interoperability, which is not addressed by the two listed projects, we propose to extend the concept of a generic middleware to support multiple personalities *simultaneously*, within a single executing middleware instance.

This requires a very flexible and customizable design. Such a design has other beneficial effects on middleware properties. To ensure predictability of a real-time, high-integrity system, a programmer may want to restrict the computational model. For this reason, Ada 95 allows the

definition of profiles, which are proper subsets of Ada 95 guaranteeing time efficiency and determinism at compile time. Existing profiles do not address distribution issues, but they provide pertinent restrictions on language features that one can use to implement lightweight configurations of a given middleware.

Our objective is to design and implement DROOPI, a Distributed Reusable Object-Oriented Polymorphic Infrastructure. DROOPI aims to build, using our experience as middleware implementors, a modular middleware design emphasizing software reuse, customizability towards environment and application requirements, and including the support of transparent interoperability through the support of multiple middleware personalities.

This paper presents the DROOPI architecture. Section 2 presents similar projects and sketches the main features we want to implement. Then, section 3 describes the architecture we have elaborated for DROOPI before concluding remarks.

2 State of the art

2.1 Implementation of platform-specific middlewares

These past years, we have developed two middlewares implementing two different distributed object-oriented models. Both middlewares are targeted to work with the programming language Ada 95. Ada 95 is the first standardized object-oriented language (ANSI/ISO/IEC-8652:1995).

It is also the first standardized language including distribution features. The Ada 95 Distributed Systems Annex (DSA) provides an interesting approach to distribution: existing constructs for separation of interface and implementation are extended with distribution semantics; all existing properties of the language are preserved.

A great effort was led by Ada Core Technologies (ACT) to provide the Ada community with a free high-quality Ada 95 compiler called GNAT. This compiler, which implements the core Ada 95 language as well as all the optional annexes, belongs to the GCC family and shares its back end with C and C++ compilers. As other compilers from the GCC suite, GNAT supports many native and cross configurations.

In collaboration with ACT, we developed GLADE, the only certified conforming implementation of DSA. GLADE is available under the same free license as GNAT. It has been designed for this particular compiler, but should be portable to any Ada 95 compilation environment with minimal efforts.

Then, the OMG CORBA model becomes popular. This model has an outstanding interoperability architecture; it addresses the need for interoperability between software components developed using various languages on discrete machine architectures.

Thanks to our previous experience in developing the GLADE middleware, we have developed AdaBroker, the unique free CORBA environment providing an Ada 95 mapping. This project provides a translator with a full Ada 95 ORB. This includes a Portable Object Adapter (giving application developers a tool-independent, fine-grained control over the management of objects within a server) and all the basic OMG Common Object Services (COS) as well.

Using a Java Virtual Machine (JVM) creates innovative possibilities for distribution (code migration, stubs download). The JVM can also be easily programmed in Ada 95, for example with the JGNAT compiler [1].

We thus also considered Sun's Java/RMI (Remote Method Invocation). However, its approach to distribution is very classical and does not introduce new ideas beyond features already present in CORBA or DSA. Work is in progress within the OMG to integrate RMI as a subset of CORBA [7]. Furthermore, the RMI distribution model exhibits severe shortcomings: although the intent of the RMI specification is to provide distribution-transparent remote invocation, Java language semantics cannot be preserved correctly in a distributed context [2].

2.2 Aspects of a generic middleware

From our experience as implementors of GLADE and AdaBroker, we have identified common architectural characteristics of the DSA and CORBA distribution platforms. Both fit in a much more general model encompassing many object-oriented distribution platforms. We therefore endeavored to formally define a generic middleware architecture based on that model, with an emphasis on the following essential features:

genericity The middleware shall factor out common components that can be shared among different personalities. Examples of such components include the management of environment resources such as control threads and data transport endpoints.

patterns Aspects of common or personality-specific components that correspond to well-known design patterns shall be identified, and their implementations shall reflect their affiliation to a generic design.

customizability The middleware shall enable users to control and tailor its functionality according to his specific purposes and application requirements. They will thus be able to control the resource usage as well as aspects of middleware operations (such as the choice of a particular communication protocol) to fit their specific requirements.

environment profiles Proper subsets of middleware functionalities shall be defined to satisfy real-time properties and exhibit predictable behaviours. For example, we plan to provide a minimal implementation with no use of non-deterministic tasking primitives as identified by the Ravenscar profile for real-time, high-integrity systems [4]. A subset of CORBA for embedded systems has already been defined with similar objectives: *minimumCORBA* [6].

interoperability The middleware shall allow objects to interoperate regardless of whether they are implemented using the same distribution platform, or discrete ones. A server object created in a given platform shall thus be accessible by clients using any other platform.

2.3 Related projects

The following projects exemplify the identified features:

Jonathan is a distributed processing environment for the Java Virtual Machine which offers the possibility to integrate various types of bindings between objects in an application [5]. Jonathan supports a CORBA and a Java/RMI personality using a generic distribution kernel.

TAO is a modular ORB whose design is based on the pervasive use of patterns [10]. Patterns help to reduce the complexity and improve the maintainability of the ORB. An ORB with a modular pattern-based design such as TAO also proves to be more extensible and evolvable than a conventional ORB.

Quarterware adopts a CORBA-like model to offer a component-based middleware [11, 3]. This design allows specialization and optimization to fit requirements of a particular application. Quarterware can endorse several personalities, which provide the core facilities of other existing middlewares: CORBA, Java/RMI and MPI.

GLADE comprises a configurable inter-task synchronization library that can transparently offer no-operation placeholder implementations for synchronization primitives when there is only one task in a partition. Avoiding the use of language tasking primitives improves the predictability and real-time properties of the application [8] The runtime library's memory footprint is also reduced.

CIAO is a code generation tool developed to let GLADE and AdaBroker interoperate [9]. It allows services developed using Ada 95 DSA to be accessed by any CORBA-compliant client. This constitutes a first step towards full interoperability between DSA and CORBA.

DROOPI's objective is to define and implement a middleware able to provide various personalities such as DSA, CORBA or RMI. We aim to integrate all the essential features listed in section 2.2.

Unlike other middlewares that can be tailored to support various personalities (such as Jonathan or Quarterware), DROOPI's goal is to support multiple personalities within the same instance of a running middleware, so as to provide interoperability between object-oriented distributed applications.

3 Architecture of a generic middleware

DROOPI's architecture is the result of a research process driven by the problem of interoperating different distribu-

tion platforms. The CIAO project showed that automatic generation of gateway components could address this issue to a certain point. It also outlined that the offered integration was not as seamless as we would have expected. We thus decided to investigate appropriate mechanisms to bridge the gap between existing middleware platforms.

3.1 Lessons learned from the CIAO project

CIAO has successfully allowed the generation of gateways between distribution platforms. However, this approach has a number of drawbacks. Some are related to the current implementation, but others are essential to the approach.

Using a gateway to convert requests from one platform to the other means that all requests will have to go through a bottleneck. Depending on the size of the application, this may be a major drawback in terms of performance. Reliability is also at stake, because the computing node executing the gateway becomes a single point of failure. Both issues may be addressed by establishing several gateways. This introduces another concern: consistence of the information used by all gateways to translate requests. It becomes difficult to ensure that translated object references designating the same DSA object seen through different gateways are identical CORBA references, i. e. to preserve object identity across gateways.

The CIAO approach also requires application developers to perform specific steps to make distributed (DSA) application open to CORBA clients. While most of this process is automated, it requires a modification to the application to include the proxy units in one of its partitions.

Our current interoperability approach provides a more convenient way to make objects defined within one environment available in another. Our objective is to enable users to define middleware personalities both at the "application level" and at the "ORB protocol level".

The application level is the interface between an application object (client or server) in a distributed system, and its host middleware. The "ORB protocol level" is the interface where our generic middleware communicates with other middleware using their specific communication protocols.

3.2 Recurring services in distributed object-oriented applications

In this section, we present a generic description of the functions that one can expect to find in a distributed OO platform; we thus provide an architectural model for such a platform, and we show how CORBA and DSA can be interpreted as instances of this model. This description captures the common structural and functional elements of distribution middlewares, in order to help structuring a generic middleware implementation supporting each of these elements under various personalities.

In this paper we restrict ourselves to middlewares supporting the distributed object paradigm. Other distribution models, such as plain RPC or Message Passing, can be implemented in terms of distributed object oriented primitives. An abstract model of the distributed OO architecture thus notionally encompasses those other architectures.

Basic services offered by distribution middlewares constitute an abstraction layer over Operating System communication facilities. This layer has to preserve the fundamental properties of the OO model: addressing (embodying *identity of objects*), exchange of requests (transport, data representation, request protocol, embodying *method calls*), and resource management (activation of objects, requests dispatching, embodying *objects life cycle*).

We now describe these functional areas:

Addressing

Definition Each object needs to be assigned an address (or *reference*). An address is an piece of information which can be passed from node to node, and denotes one particular object unambiguously all over its existence.

Properties An address must be a *global* identifier for an object. Addresses need to designate an object unambiguously: they must be globally *unique*.

Transport

Definition A node must be able to establish a communication link to an object. This link is used to transmit requests for the execution of object methods.

Properties Messages must be reliably delivered.

Marshalling

Definition Request data must be translated into a representation suitable for transmission over a network.

Properties In order to build interoperable applications, this representation has to be previously standardized. For instance, the middleware at the calling and receiving ends of the communication channel have to agree on the size and endianness of integers.

Protocol

Definition The middleware implements a protocol for the transmission of requests among instances of distribution runtimes.

Properties The fundamental primitives of a distributed objects protocol are *Request* and *Reply*, which are used to implement remote invocation of subprograms and methods. Abortion of pending requests can also be provided by protocol primitives. Further primitives may be found in particular platforms; these are used to perform various high-level functions, and can be functionally understood as requests sent to objects that are part of the platform implementation.

Activation

Definition The middleware has to activate and deactivate the concrete entities implementing objects. When a request is received, it ensures that the object reference is mapped onto an actual object implementation. The middleware may have to create an object implementation on-the-fly, or to retrieve an object state from persistent storage.

Properties Each request must be processed within the correct context (including object state and identity, history of preceding requests, etc.)

Dispatching

Definition When a request reaches a node, it has to be assigned onto an execution resource (a thread of control) for processing. The distribution runtime has to find a suitable thread on the node, or create one if necessary.

Properties The dispatching mechanism should not introduce dead locks. If possible, it should process requests fairly; request prioritization support may also be provided.

The previously listed services are sufficient to provide the core functionalities of a distribution middleware. They are built atop the operating system communication facilities and provide distributed object-oriented abstractions.

However, applications often require support for more sophisticated functions related to distribution:

Naming

Definition The Naming service allows object references to be associated with symbolic names, and nodes to query the service for any reference associated with a name.

Synchronization

Definition A synchronization service may be provided to synchronize the operation of nodes executing actions in parallel.

Interface Repository

Definition An application may need to invoke operations on objects whose interface was not known when the application was created. For this invocation to be performed correctly, a description of the interface must be available. The purpose of Interface Repository Services is to propagate such information: they distribute service descriptions to interested nodes.

Properties The abstractions manipulated by the Interface Repository service must reflect the platform's service model.

A dynamic invocation mechanism is necessary to use the information provided by an interface repository. This facility enables nodes to invoke dynamically discovered methods. A dynamic invocation interface reifies the process

of creating a remote method invocation request. After the request object is constructed, it can be processed like any ordinary method invocation on a remote object; the receiving object cannot distinguish dynamically-constructed requests from statically-created ones.

Termination

Definition A distributed application consists of a set of computing nodes that cooperate to reach a goal. If, at some point in time, all nodes agree that this goal has been reached, they may decide to globally terminate processing and perform an orderly shutdown. A Termination service can be implemented to establish this decision.

Properties Such a service must determine a consensus among participating nodes on whether to end the application. The Termination service provides a support for applications to make this decision. Implementation of this service requires specific support from the ORB in addition to the low-level services listed above. Specifically, the implementation of the Termination service needs some way to inspect the state of the ORB's internal threads of control.

Shared data

Definition Nodes in a distributed application may want to share part of their data. A shared data service provides shared storage transparently for the application developer: distributed shared data appears like normal local data; the distribution runtime takes care of propagating the effects of data access between nodes.

Properties A shared storage service must provide all nodes with a consistent view of the shared variables. Several models of consistency can be defined.

3.3 Architecture summary and rationale

We propose to make use of a layered architecture that decouples application-level personalities from ORB protocol-level personalities through a generic ORB core layer. The application-layer personalities are interfaces presented by the middleware to the application components using it: client and server objects. These personalities depend on code generated from user-provided service descriptions to map personality-specific requests sent by clients objects to server objects from a personality-dependent representation to a personality-independent one.

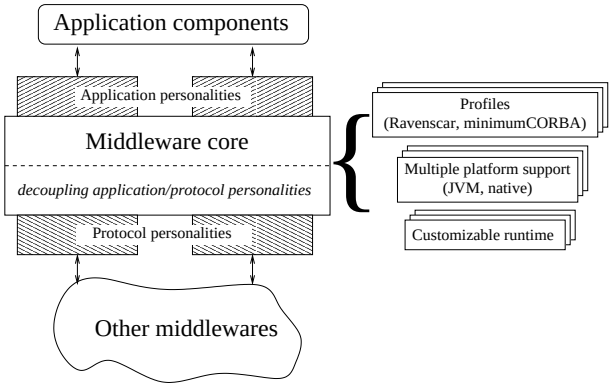

Figure 1: Overview of the DROOPI architecture

This representation is then passed to an ORB protocol personality for transmission to the target object. It should be noted that our architecture does not impose any constraint on the protocol personality to be used for a given application personality; both layers are entirely decoupled. This allows the architecture to offer interoperability between distribution platforms as seen by application developers.

This personality-independent, reified representation of method invocation requests is very similar to the Request object defined by the CORBA Dynamic Invocation Interface. It is also the key point in avoiding a combinatorial explosion when adding new personalities to the generic middleware. Adding a new set of application and protocol personalities only requires the development of:

- an application personality library;
- a protocol personality library.

Adding a new personality is thus a task of constant complexity; this could not be achieved with the gateway

generation approach, where the complexity of supporting a new distribution platform would require the development of as many gateways as existing supported platforms.

Figure 1 summarizes the proposed architecture. The key-points we identified in 2.2 are addressed as follows:

genericity This design captures the common behaviours that are represented in various distribution platforms. These behaviours are implemented as commodity components providing services to other middleware modules. An typical example is the elaboration of a common generic representation for requests.

patterns We use generic, reusable modules extensively. When a well-known design pattern is encountered in the design of DROOPI, we reflect it the implementation by the use of Ada 95 facilities such as generic units and abstract tagged types.

customizability Our architecture delineates abstract interfaces between components, and allows several concrete implementations of these interfaces to allow compile-time and run-time customization of its behaviour. This will also allow support of multiple platforms (native compiler, Java virtual machine).

environment profiles Multiple application profiles will be supported through customization of key resource management components, such as the memory allocation subsystem, or the task creation/destruction subsystem. This will allow the definition of an ORB configuration compatible with the Ravenscar profile. A minimal subset of CORBA, *minimumcORBA* [6], has been defined by the OMG to run in such "light" environments.

interoperability This architecture decouples the personalities presented to the local application and to other middlewares (and hence other computing nodes): objects created within one personality of the generic middleware become transparently available to any client that uses a distribution platform for which a corresponding protocol personality exists.

4 Conclusions and future works

DROOPI is a novel middleware that will allow interoperability of distributed object-oriented applications across distribution platforms. We have completed the first step of this project, which consists in the definition of a generic middleware architecture. This architecture integrates and extends several aspects of existing middlewares.

The implementation of this design has started quickly, thanks to the re-use of several modules from previous projects. We have integrated from the very beginning of the project the port to the Java virtual machine by developing concurrently a native version and a JVM version of DROOPI. We take also care of using a restrictive subset of Ada 95 to ensure the compatibility with common real-time environment profiles.

To implement services that make use of the generic middleware, a code generator translating service descriptions to the middleware's internal, personality-independent representation is necessary. This phase of application development is currently addressed by ad-hoc tools based on the `GLADE` and AdaBroker code generators. In the future, we will investigate new solutions to provide generic code generation facilities, thus extending the notion of a generic middleware to a complete generic distributed application development tool suite.

Like `GLADE` and AdaBroker, software developed within the DROOPI project will be made available to the community as a *free software* product.

References

[1] E. Briot. JGNAT: The GNAT Ada 95 environment for the JVM. In *Ada France 1999*, Sept. 1999.

[2] G. Brose, K.-P. Löhr, and A. Spiegel. Java Does not Distribute. In *TOOLS Pacific '97*, Nov. 1997.

[3] R. H. Campbell, J. W. Coomes, A. Dave, N. Islam, Y. Li, W. S. Liao, S. Lim, T. Qian, D. K. Raila, E. Roush, A. Sane, M. Sefika, A. Singhai, and S. T. Tan. Customizable Object-Oriented Operating Systems. Submitted for *Communications of the ACM* Special Issue on Recent Developments in Operating Systems, Sept. 1996.

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

[5] B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stéfani. Jonathan: an Open Distributed Processing Environment

in Java. In *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, Sept. 1998.

[6] Object Management Group. *minimumCORBA*. Object Management Group, Aug. 1998. OMG orbos/98-08-04.

[7] Object Management Group. *Java Language to IDL Mapping*. Object Management Group, Jan. 2000. OMG ptc/00-01-06.

[8] L. Pautet and S. 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.

[9] T. Quinot. CIAO: Opening the Ada 95 Distributed Systems Annex to CORBA Clients. In *Ada France 1999*, Sept. 1999.

[10] D. C. Schmidt and C. Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine Special Issue on Design Patterns*, 1998.

[11] A. Singhai. *QuarterWare: A middleware toolkit of software RISC components.* PhD thesis, University of Illinois at Urbana-Champaign, 1999.