From functional to architectural analysis of a middleware supporting interoperability across heterogeneous distribution models

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.

POLYORB*, our generic middleware, directly addresses this problem: interoperability between distribution models. It can be customized into several "personalities" (e.g. CORBA, or RMI, or Ada 95 DSA), such as Jonathan and Quarterware. It also provides* simultaneous *support of multiple personalities within a single middleware instance. Software component implemented using a given distribution model may thus interact with remote entities from a different distribution model. This paper identifies the functions required to implement such a middleware and maps them on an operational design.*

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. Each of them offers slightly different flavors of the distributed objects paradigm, with various levels of services. The interface presented by these middleware platforms to application developers are also different, as are the underlying mechanisms used in implementation. These incompatibilities seriously impair the reusability of distributed application components, out of the environment they were built for.

In the remainder of this paper, the term "interoperability", or interoperability between distribution models, stands for the capability of a middleware to allow objects defined within a given distribution model to interoperate with remote objects from a different distribution model.

Interoperability can be achieved case by case using specific gateways between components. However, this approach requires new gateway development for each possible pair of middlewares. It therefore does not promote reusability of middleware code, nor scales to applications in the large.

Multi-personality middlewares have emerged from the observation that much of a middleware's behavior is independent of the personality, or façade, it assumes. For example, the generic middleware components of the Jonathan project [4], can be customized for a given middleware strategy (CORBA or Java/RMI). Similarly, Quarterware [20, 19] 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 [10]). More generally, most middleware functions can be described as a set of services that are independent of the any particular personality.

Figure 1 illustrates various services required to implement a distribution model. The vertical axis corresponds to functionality level and the horizontal axis to distribution model requirements. Obviously, CORBA offers limited services compared to DSA or CORBA enriched with Common Object Services (COS). In our architecture, *fundamental services* interconnect existing middlewares and allow to implement canonical distribution models. Then, *advanced services* offer sophisticated behaviours and enable the implementation of richer distribution models.

Although the mentioned projects, Jonathan and Quarterware, do emphasize the separation of middleware services and middleware personalities, they do not address the issue

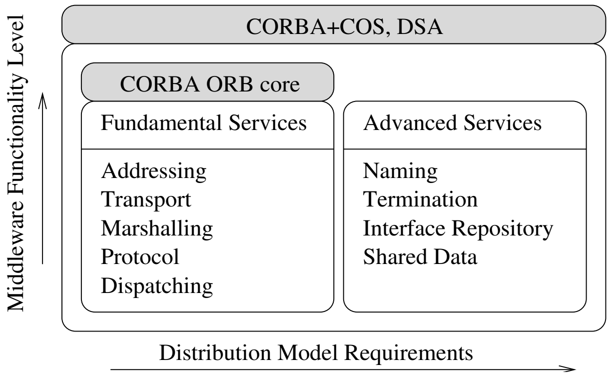

Figure 1: Middleware services and functionality level

of interoperability. Unlike these projects, we propose to extend the concept of a generic middleware to *simultaneously* support multiple personalities, within a single executing middleware instance.

This requires a very flexible and customizable design, thus also allowing adjustment of middleware properties towards application requirements. For example, to ensure predictability of a real-time, high-integrity system, a programmer may want to restrict the computational model.

Our objective is to design and implement POLYORB, a Distributed Reusable Object-Oriented Polymorphic Infrastructure. POLYORB aims to build, using our experience as middleware implementors, a modular middleware emphasizing:

- software reuse,
- customizability towards environment and application requirements,
- transparent interoperability between components designed for different personalities.

This paper details the POLYORB architecture. Section 2 presents similar projects and structures our objectives. Section 3 explicits the fundamental services offered by POLYORB. In section 4, we discuss their implementation in an architecture. Then, section 5 proposes a detailed design using UML class diagrams. Finally, section 6 details advanced services in POLYORB and sketches how they can be supported.

2 State of the art

In this section, we first draw lessons from our experience in implementing two platform-specific middlewares: *GLADE*, the only certified conforming implementation of the Ada 95 distributed systems annex, and *AdaBroker*, the free-software CORBA suite for Ada 95.

From this experience, we show that these object-oriented middlewares share several essential features, and we claim that these common features could be factored out into generic middleware components. We then define several aspects to be fulfilled by such generic components and relate them to existing generic middlewares. We conclude that existing systems do not simultaneously fulfill all these aspects.

2.1 Our experience in 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 language including distribution features [7]. The Ada 95 Distributed Systems Annex (DSA) provides an interesting approach to distribution: distributed subprogram invocation (either as statically or dynamically bound remote procedure calls [1], or as distributed object method invocations) is offered as a natural extension of the language's standard subprogram call constructs. Shared objects are also provided, which can be mapped on any shared memory support like a multiprocessor memory or a distributed shared memory. High-level semantics of the language are preserved when using the DSA; for example, exception propagation works as usual and timed subprogram calls can be used as if the program was not distributed. In collaboration with Ada Core Technologies, we have developed GLADE, a free software, certified conforming implementation of DSA for the GNAT compiler.

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 and machine architectures.

Thanks to our previous experience in developing the GLADE middleware, we have developed AdaBroker, the only 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 [2].

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 [13]. 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 [3].

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 aspects:

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 allow 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.

interoperability The middleware shall allow objects to interoperate regardless of whether or not they are implemented using the same distribution platform. 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 aspects:

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 [4]. 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 [17]. 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 [19]. 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 [14] The runtime library's memory footprint is also reduced.

CIAO is a code generation tool developed to let GLADE and AdaBroker interoperate [16]. 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.

POLYORB'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), POLYORB'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 Fundamental middleware services

POLYORB's architecture is the result of a research process driven by the problem of interoperating different distribution 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 raises several issues, including performance (the gateway is a bottleneck), and reliability (it is also a single point of failure).

Moreover, duplicating gateways to address this issue would automatically introduce another concern: the consistency of the information used by all gateways to translate requests. It would become infeasible 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 approach to the problem of interoperability 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 from two separate points of view:

- the aspect presented to application components;
- the interface to other middlewares.

3.2 Recurring services in distributed object-oriented applications

In this section, we present a generic description of the services that one can expect to be provided by a distributed objects middleware. We therefore provide a functional model for such a middleware, and show how CORBA and DSA can be interpreted as instances of it. This models captures common middleware functions.

In the following sections, this functional structure is used to derive an architecture for the implementation of a generic, supporting each of these elements under various appearances, or personalities.

This paper is primarily focused on middlewares supporting the distributed object paradigm. Such middlewares use *message passing* mechanisms — a more basic distribution paradigm — to transmit method invocation requests from callers to callees. Various implementations of message passing can be used: simple network sockets, or more sophisticated systems such as MPI [10].

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 (object oriented) 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 give a more detailed description of 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*.

Rationale One of the fundamental properties of objects is *identity*. Addresses embody that property by allowing nodes to manipulate these object identities.

Example An open addressing scheme that satisfies these properties can be established using TSAP (Transport Service Access Point [6]) addresses from the underlying networking environment, associated with a locally unique identifier attributed by the middleware. This is the solution retained in CORBA.

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.

Rationale Invoking an object method in the object-oriented programming model consists in *sending a message* to the object [5]. Distribution fits nicely in this model: the corresponding message goes through a communication network.

Example Existing message-passing libraries can be reused. This method has already been successfully applied to reimplement the Java/RMI platform over existing distributed processing libraries. [11] presents KaRMI, a drop-in replacement for Sun's RMI, which provides an abstraction for the message passing layer. This abstraction can make use of TCP/IP sockets as well as specialized message-passing hardware in a parallel environment; KaRMI also has the ability to create bridge objects between different technology domains on-the-fly.

Marshaling

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.

Rationale System representation choices are a tradeoff between portability and cost of data conversion from machine representation to network representation (and back).

Example The chosen representation must balance the cost of conversion from platform representations and back, against the overhead of marshalled data. For example, a common data representation may be specified for all data types, with an added endianness flag in each message indicating whether items are stored low-order byte first or high-order byte first [12]. This provides a simple representation, with limited overhead, while suppressing the cost of "byte-swapping" operations. As these operations involve a significant portion of the critical path in request marshaling, this optimization is greatly beneficial to performance.

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.

Example CORBA defines the Generic Inter-ORB Protocol (GIOP) for this purpose.

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.)

Example The Portable Object Adaptor of CORBA provides several policies on various issues (automatic activation, system- or user-controlled object identifier assignment...)

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.

Rationale Several policies for deciding when to create a thread, and on which thread to dispatch a given request, can be defined. The choice of a particular threading policy is discussed in [18].

Example `GLADE` implements a pool of dynamically created threads. Most incoming requests can thus be served immediately, while keeping the memory and context switching footprint of the runtime within limits.

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.

3.3 Example

Let us suppose that we have a very simple object with a single method *echo* taking a string argument, and sending that string back to the caller. Invoking this method involves the following steps:

Addressing The caller determines a reference to the object.

Transport The caller-side ORB extracts a network address from the reference and creates a connection to the object's ORB.

Marshaling It converts the string passed to the method into a representation suitable for network transport, for example a four-byte, little-endian integer containing the length of the string, followed by the string itself.

Protocol It creates a *Request* message containing the name of the method, the object reference, and the marshaled argument, along with a unique request ID. It sends the message across the transport link to the callee ORB.
The callee ORB receives the message.

Activation The callee ORB looks up the object reference in its internal tables, and associates it with an implementation object.

Dispatching The callee ORB finds an idle thread within its available thread pool, and assigns the execution of the request onto that thread.

Marshaling The callee ORB marshals the echoed string as above.

Protocol It creates a *Reply* message containing the request ID and the marshaled result string. The reply is sent back to the caller.
The caller ORB unmarshals the result and returns it to the calling routine.

3.4 Architecture summary and rationale

We define a layered architecture decoupling application-level personalities from middleware protocol-level personalities through a generic middleware core layer, as show on figure 2. The idea behind this architecture is close to modern compiler architectures, organised as a parsing front-end, an abstract intermediate representation, and a code generation back-end, with well-defined interfaces between each layer. In both cases, the topmost and bottom layers are completely

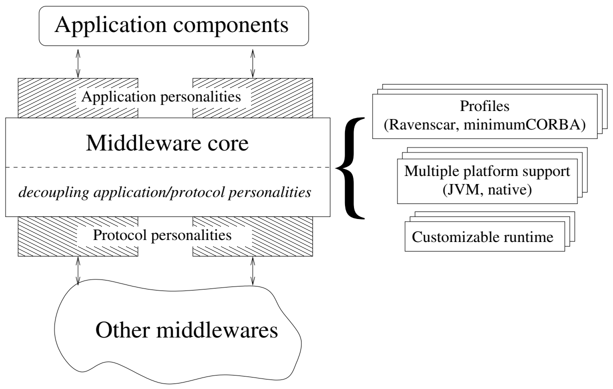

Figure 2: Overview of the POLYORB architecture

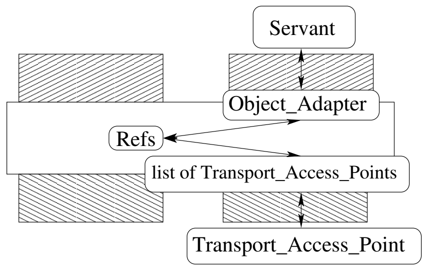

Figure 3: *Addressing* function

independant, and one may be replaced without changing the other. This allows new variants of each layer to be introduced with linear, instead of quadratic, cost.

The application-layer personalities are the interfaces presented by the middleware to the application components using it: client and server objects. These personalities may depend on code generated automatically from a user-provided service descriptions. Such code is used to map personality-specific requests sent by clients objects to server objects from a personality-dependent representation to a personality-independent one.

This representation is then passed to a 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 objects defined by the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface. It is also the key point in avoiding a combinatorial explosion when adding new personalities to the generic middleware. 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.

The points we identified in 2.2 are addressed as follows:

genericity This design captures the common behaviors that are represented in various distribution platforms. These behaviors 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 POLYORB, 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 behavior. This will also allow support of multiple platforms (native compiler, Java virtual machine). For example, we have implemented an abstract representation of Berkeley sockets which can be used either on top of UNIX or Windows native sockets, or Java socket classes, and provide a unified interface to TCP/IP communication.

interoperability This architecture decouples the personalities presented to the local application and to other middlewares (i.e. other computing nodes): objects created within one application personality are available to any client that uses a middleware for which a suitable protocol personality exists.

4 Integration in the architecture

In this section, we explain how the proposed architecture implements the fundamental services described in section 3.2. The diagrams in this section give a coarse-grained indication of each component's position and relationships within the architecture outlined in section 3.4.

Addressing (figure 3): as an object is created and registered with the ORB, references designating that object can be created. A reference associates a local identifier of the object (provided by the application layer) and a transport access point address (provided by the protocol layer). The *Refs* module, which creates object references, therefore needs to be have interface with both layers, and must be situated in the generic part of the ORB software.

Transport (figure 4): the transport function consists in transferring information from one node to another: it con-

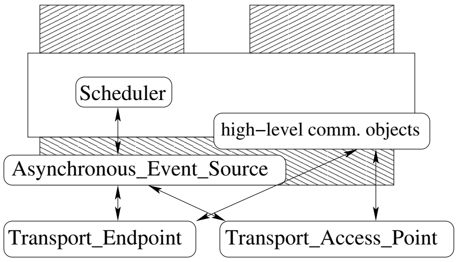

Figure 4: *Transport* function

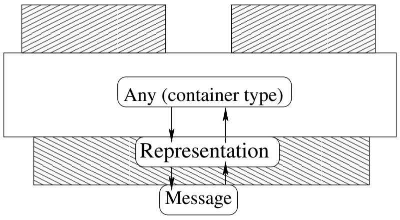

Figure 5: *Representation* function

stitutes the lowest level of the communication subsystem.

This service is implemented by overriding several abstract classes:

Transport_Access_Point A listening point with an address, to which remote middlewares can connect.

Transport_Endpoint An open (connected) communication point which a middleware can use to send data to another.

Asynchronous_Event_Source Any entity that can receive external events which are to be processed by middleware activities, and that can be polled by the middleware when such events are expected.

A typical set of concrete implementations of these classes is: a listening socket, a connected socket, and a set of file descriptors that can participate in a *poll(2)* system call.

Representation (figure 5): the representation function consists in translating a given data D having type T into a message that can be sent to another ORB, following a representation imposed by an inter-ORB communication protocol (for example the CDR representation in the case of CORBA's GIOP protocol):

$$< D >_T \xrightarrow{Representation} msg$$

Information may be lost along this transformation: the resulting message needs to contain enough information to reconstitute the value D knowing T and msg, but not necessarily enough to reconstitute $< D >_T$ knowing only msg (for example, in CDR/GIOP, an arbitrary sequence of four bytes can correspond either to a long integer or to a structure with two short integer members). Indeed, the reverse transformation would require a "magical" retrieval of the typing information that was lost during the first one.

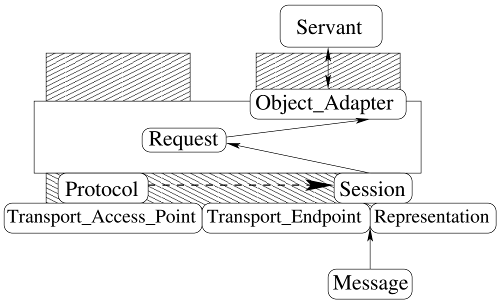

Figure 6: *Invocation protocol* function

The reciprocal transformation that must be implemented by the representation function must therefore take msg **and** T as input, and produce the typed data as output:

$$(msg, < \cdot >_T) \xrightarrow{Representation} < D >_T$$

where $< \cdot >_T$ denotes an empty container carrying T type information.

The client of the representation service thus has to have an *a priori* knowledge of the data type to be received. The service is implemented by providing a concrete implementation of the two abstract transformations mentioned above.

Protocol (figure 6): the "method invocation protocol" function bears some similarity with the representation function: it transforms high-level abstractions in a local form into messages that can be sent out to other nodes, and vice-versa.

Method invocation also organizes the sequence of operations involved in a method call: from a local invocation request, prepare and emit a message, then wait for a reply, then translation of the reply into the form expected by the caller; from a remote invocation message received from another ORB, preparation of an internal request object, unmarshaling of parameters and target object identity, then execution of the request.

Requests handled by the protocol function are represented as instances of a personality-independent class *Request*, that contain:

- target object identification;
- method identification (implies a common convention between the application and protocol layers);
- parameters and result of the invocation.

This class is similar to CORBA's *Request* and *ServerRequest* pseudo-interfaces.

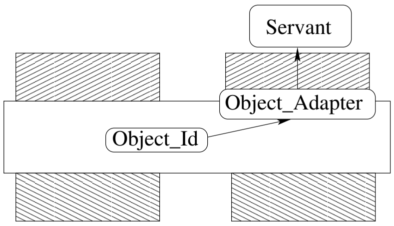

Figure 7: *Activation of implementation objects* function

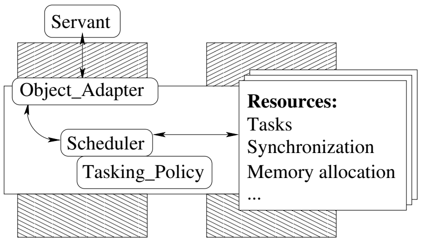

Figure 8: *Dispatching* function

Each protocol personality needs to implement at least the following functionalities:

- transformation of a request into a message;
- transformation of a message into a request, querying the object adapter to obtain the description of the method's profile. Method profile information is used to call the representation function in order to unmarshal method arguments.

Activation (figure 7): as a request received and decoded by the protocol layer is ready for execution, the ORB has to associate the target logical object with a concrete structure implementing it. This task, consisting in "incarnating" the object as a concrete servant is done by the object adapter.

Dispatching (figure 8): when a request is going to be executed by a servant, it needs to be assigned to a thread of execution. This thread can be either an application thread, or taken from an ORB thread pool, or be created specially for the current session, or for the current request.

The assignment of a request to a thread is implemented by the *Scheduler* object. The selection of a thread assignment policy is done by delegating some of the scheduler's behavior to *Tasking_Policy* objects.

5 Implementation

We have selected an event-driven design: the middleware should not dictate, but rather adapt to, tasking and real-time requirements of the application. Consequently, the middleware may not introduce new control threads in the application, and should be activated only when necessary (i.e. when data is received from a remote ORB, or when an application object attempts to invoke a remote method).

Core Layer (UML diagram on figure 9): the main task of the generic core layer is the management of execution threads. It is achieved by the following objects:

Scheduler organizes the execution of ORB activities triggered by these events. Different policies may be used to assign events to threads of control.

Tasking_Policy act as delegates of *Scheduler*. Concrete derivations of these classes allow the configuration of scheduling behavior.

Application layer (UML diagram on figure 10): the *Application layer* manages application objects, and contains the following classes:

Object_Adapter an "object directory" which associates logical references (*Object_References*) to actual object implementations (*Servants*);

Object_Reference a piece of data, created by an object adapter, and comprising one or more *object profiles*. Each profile designates a communication endpoint where requests can be received, the identification of the object adapter that created the reference, and the identification of the designated logical object.

Servant the root class for all object implementations, i.e. anything capable of executing a request.

The *Servant* and *Object_Reference* classes are derived in a personality-specific fashion to be integrated with application components.

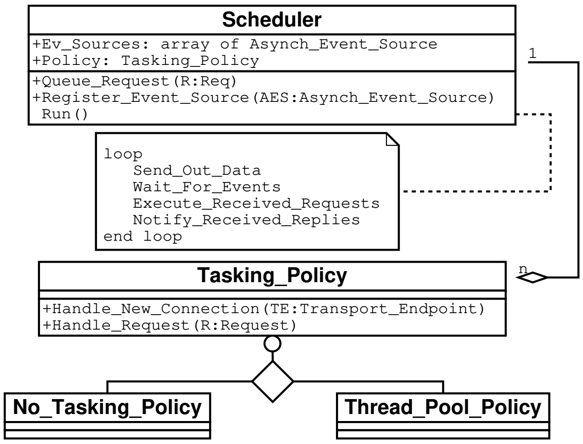

Figure 9: Core layer

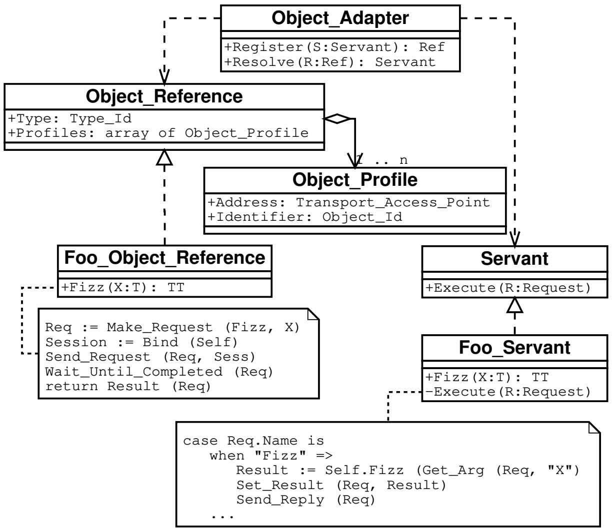

Figure 10: Application layer

Protocol layer (UML diagram on figure 11): the *Protocol layer*, in charge of inter-middleware communication, is composed of several abstract classes:

Asynch_Event_Source is any entity able to report external (asynchronous) events. Possible concrete derivations of this class include sockets, System V IPC objects, etc.

Filter defines an interface for dynamically-typed message passing within POLYORB. Any entity that needs to handle communication with other ORBs implements this interface. This allows actual message types exchanged between middleware components to be separately specified, in order to resolve cross-visibility issues.
Filters are created using *Filter_Factory* objects, which can be chained together. A chain of filter factories constitutes a template for a stack of active filters.

Transport_Access_Point allows incoming connections from remote middlewares to be received. A transport access point has an address which can be communicated to other nodes, and that designates it unambiguously within a communication domain. A transport access point acts as a factory of transport endpoints. It can also be viewed as an asynchronous event source (*AES_View*).
The transport access point view is presented to upper communication layers; the asynchronous event source view is presented to the scheduler.

Transport_Endpoint allows message exchange with remote middlewares. Like a transport access point, a transport endpoint can also be viewed as an asynchronous event source (*AES_View*).
An example of concrete transport endpoint is *Transport_Endpoint_Socket*, which is implemented using Berkeley sockets.

Protocol is the interface for one specific method invocation protocol (for example GIOP or SOAP, the Simple Object Access Protocol [22]). A *Protocol* acts as a factory of *Session* objects.

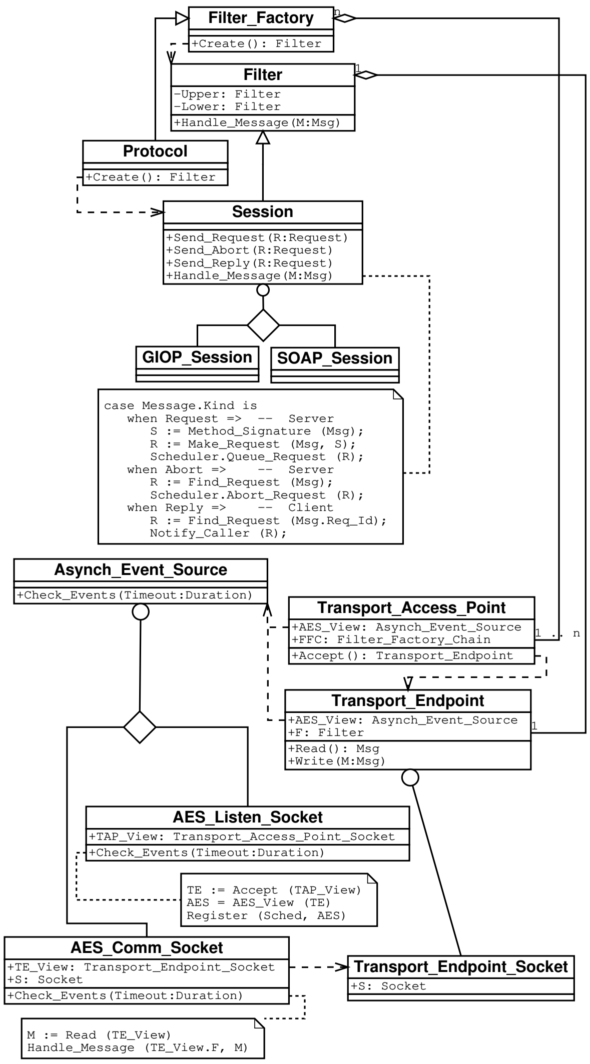

Figure 11: Protocol layer

Session embodies the state of an open protocol connection and carries out the exchange of messages necessary to pass method invocation requests from application objects to remote middlewares and back.

6 Supporting distribution models with advanced services

In section 3.2, we have identified functionalities that are necessary and essential to the definition of a distributed objects-based middleware.

However, some distributed object offer sophisticated abstractions whose implementation pose additional requirements on middleware functionalities. Example of such services are:

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.

Rationale This service lets applications determine the references of objects they need at run-time. It is essential when an application has to adapt to environment changes: it allows developers to avoid using hard-coded object references, making the distributed application reconfigurable.
Moreover, the naming function is essential to the initialisation of a distributed application: it lets distributed objects get "acquainted". It allows initial references to remote objects to be distributed to all nodes that require them.

Example DSA has an implicit naming service: all nodes can invoke operations defined by units categorized as *Remote Call Interfaces*. The runtime takes care of locating on which node the named unit resides. This service is also used to prevent multiple declarations of unique entities like RCI units or to check remote unit versions. For these reasons, the naming service has to be an internal entity of the `GLADE` communication system.

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.

Example `GLADE` provides a global termination algorithm for DSA. If all nodes have terminated their own work, and no message is in transit in the communication network, it can be proven that no new computing work can happen [9]. In that case, global termination can be decided. Global or local termination policies can be selected per partition [15].

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 this 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. To this effect, it reifies the process of invoking a remote method. After a 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.

Example CORBA defines an Interface Repository service and an associated Dynamic Invocation Interface.

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 just as 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.

Rationale This service provides developers with a very straightforward way to define a set of globally available data.

Example DSA offers *Shared Passive* data: library units containing only variables, which can be accessed consistently from any node in a distributed application. `GLADE` provides several shared data supports like file or distributed virtual shared memory [8].

Like the fundamental services mentioned in previous sections, these advanced services can be implemented in a personality-independent fashion. These possible extensions to the generic middleware components allow the support of sophisticated abstractions defined by personalities with a high-level distribution model.

7 Conclusions and future works

In this paper, we have addressed the problem of "interoperability" between middlewares. By "interoperability", we mean the capability of a middleware to let components based on various distribution models cooperate.

Projects such as Jonathan [4] and Quarterware [20, 19] propose to implement generic middlewares offering various personalities. These middlewares can be instantiated according to a variety of distribution models (such as MPI, CORBA, RMI). However, they do not address "interoperability", since different personalities cannot run simultaneously on a same middleware instance.

This announces a new emerging class of middlewares: simultaneously running several personalities in the same middleware instance corresponds to a *schizophrenic* behaviour as mentioned in [21]. As a consequence, several components implemented using different distribution models could interoperate without any change. This should increase the potential for reusability in modern distributed systems.

Providing interoperability between various distribution models is difficult, because of the significant discrepancies in the level of abstraction provided: the Ada 95 DSA distribution model is much more sophisticated than CORBA core functionality, or the Message Passing.

This paper presents the design of POLYORB (Distributed Reusable Object-Oriented Polymorphic Infrastructure). POLYORB achieves genericity and addresses the issue of "interoperability" by a strict decoupling of the two interfaces of a distribution middleware: the interface to *application objects*, and the interface to *ther middlewares*.

Our design separates *fundamental* services, required to implement a basic set of distributed objects functionalities, from *advanced* ones, required to provide higher abstractions. Five fundamental requirements are taken into account:

1. genericity;
2. patterns and code reuse;
3. customizability towards application requirements;
4. distribution model interoperability.

Since POLYORB's design and implementation are completed, we are currently working on several personalities (in the order of potential achievement): SOAP, MOM (Message-Oriented Middleware) close to Java Message Service, CORBA, and DSA.

References

[1] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

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

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

[4] B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stéfani. Jonathan: an Open Distributed Processing Environment in Java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer Verlag, Sept. 1998.

[5] D. H. H. Ingalls. The Smalltalk-76 Programming System Design and Implementation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona*, pages 9–16, Jan. 1978.

[6] ISO. *Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*. ISO, Feb. 1995. ISO/IEC 7498-1:1994.

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

[8] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[9] F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing*, 2(3):161–175, 1987.

[10] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, June 1995. Version 1.1.

[11] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *ACM 1999 Java Grande Conference*, pages 153–159. ACM, June 1999.

[12] Object Management Group. *The Common Object Request Broker: Architecture and Specification, revision 2.2*. Object Management Group, Feb. 1998. OMG Technical Document formal/98-07-01.

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

[14] 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.

[15] L. Pautet and S. Tardieu. *GLADE User Guide*. Ada Core Technologies, 2000.

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

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

[18] D. C. Schmidt and S. Vinoski. Comparing Alternative Server Distributed Programming Techniques. *SIGS C++ Report*, 7(8), Oct. 1995.

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

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

[21] M. Swanson, L. Stoller, T. Critchlow, and R. Kessler. The design of the schizophrenic workstation system. pages 291–306, 1993.

[22] World Wide Web Consortium. *Simple Object Access Protocol (SOAP) 1.1* , May 2000. W3C note.