A case study of Middleware to Middleware: MOM and ORB interoperability

Jérôme Hugues, Fabrice Kordon, Laurent Pautet, and Thomas Quinot

{hugues, pautet, quinot}@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. Diversity in distributed applications leads to diversity in distribution models, and hence in middleware. However, large systems may need different types of middleware and interoperability between them, requiring "Middleware to Middleware" architectures. We have introduced the *schizophrenic middleware* concept as a general solution for interoperability between distribution models. PolyORB, our implementation of a schizophrenic middleware, demonstrates full interoperability between CORBA, SOAP, and the Ada 95 Distributed System Annex (DSA). In this paper, we present an assessment of the usability of our platform to implement Message Oriented Middleware (MOM). We then study MOM and ORB interoperability, from both an architectural and a functional point of view, and finally discuss benefits provided by our architecture to implement middleware.

1 Introduction

Middleware is a commonly accepted solution to ease the development of large heterogeneous distributed systems. The choice of a particular middleware is a key design issue: as there is no "one true" distribution solution, this choice is not neutral and may deeply influence the final design and performance of an application.

There is therefore a need to rapidly tailor middleware to fit an application's requirement of a specific *distribution model*. 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 Object, Distributed Shared Memory (DSM), and Message Passing.

A solution is to reuse or adapt existing software components provided by generic middleware and instantiate them according to a distribution model to create a *personality*. However, the cost of this personalization may be important, thus reducing the benefits of this approach.

Diversity among existing middleware also introduces a new layer of heterogeneity which may lead to new incompatibilities, impeding the reuse of legacy applications, or interaction between different applications. Designing middleware to be as independant as possible from the underlying distribution model, and to satisfy interoperability needs of interconnected application has become a major issue: *Middleware to Middleware* (M2M) [Bak01].

We have introduced the *schizophrenic middleware* concept [QPK01] as a solution to both the genericity and the interoperability problems. Schizophrenic middleware extends generic middleware to simultaneously support multiple interacting personalities within the same middleware instance. Interoperability is provided through dynamic gateways between these interacting personalities. We have implemented a free software schizophrenic middleware, PolyORB [PQK+01], to assess the feasibility of this concept. We have also implemented personalities: CORBA [OMG98], SOAP [W3C00] and the Ada 95 Distributed System Annex [ISO95] (DSA). We have practised pervasive code reuse when personalizing our middleware. We also demonstrated that dynamic gateway operate between various distribution models.

In this paper, we show that our schizophrenic middleware is able to support different distribution models. Specifically, we address Message Oriented Middleware (MOM) personalization for PolyORB. We then discuss effective gains provided by schizophrenic middleware architecture both as a rapid prototyping platform to deploy new middleware at low cost and as a solution for interoperability between distribution models.

We first give an overview of the schizophrenic middleware concept, and of our implementation: PolyORB. We then study existing MOM, giving the structure of a canonical MOM, and compare significant MOM architectures with PolyORB's design. We show that MOM can be built using PolyORB. We finally illustrate this claim with the design and implementation of MOMA, Message Oriented Middleware for Ada, derived from JMS specifications. Design and performance issues, as well as interoperability with object-oriented distributed systems, are discussed.

2 Overview and architecture of PolyORB

The choice of a middleware impacts the design of distributed applications. When integrating legacy components in a complete application, it is often necessary to use various types of middleware supporting distinct distribution models. We refer to the capability of a middleware to allow entities that exist within different distribution models to exchange information with, and perform services on behalf of, each other, as *interoperability* or *interoperability between distribution models*.

Most middleware implementations provide similar sets of common abstractions to distributed applications. Generic middleware may therefore be built around canonical elements (with a design patterns or component approach [GHJV94]), which are personalized to conform to a specific distribution model. Existing projects have investigated

this approach: Jonathan [DHTS98] for CORBA or Java/RMI middleware, Quarterware [SSC98] for CORBA, RMI or MPI, and Advanced Communication Toolkit [FM99] (ACT) for CORBA and cBus (a MOM personality). They demonstrate that most of middleware functionality can be described as a personality-agnostic set of services. A personality is then defined as a set of concrete modules providing access to generic middleware services.

However, these projects do not provide distribution model interoperability. A personalized middleware remains monolithic and only one distribution model is allowed per instance. PolyORB [PQK+01] strives to resolve this issue. It extends the notion of generic middleware to allow *simultaneous* support for multiple personalities within a single executing middleware instance. We refer to middleware that exhibit this property as *schizophrenic*.

In the remainder of this paper, we call *entities* the elements of a distributed application that are made remotely accessible through middleware, such as CORBA objects, RPC procedures, JMS queues, etc.

2.1 Overview of PolyORB

Let us now list properties of *schizophrenic middleware*:

- **patterns:** middleware components are identified to well-known design patterns, their implementation demonstrate their effective use.
- **customizability:** developers may customize middleware to fit specific needs, such as resource management policy, transport protocol, etc.
- **genericity:** middleware factors out common behaviors or services that can be shared by different distribution personalities.
- **interoperability:** entities, implemented over distinct personalities are able to *transparently* interact.

To meet these properties, PolyORB decouples personalities. *Application-level* personalities and *protocol-level* personalities are connected to the *neutral core layer* as shown in figure 2. Compared to a generic middleware architecture (figure 1), it supports interaction between personalities by means of the neutral core layer (shared by all personalities).

Application personalities constitute the adaptation layer between application components and middleware. They provide APIs to register application entities with PolyORB's core, and interoperate with the core to allow the exchange of requests with remote entities.

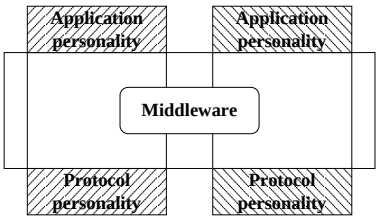

Fig. 1. Generic architecture

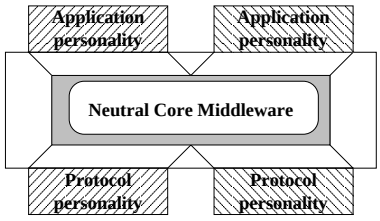

Fig. 2. Schizophrenic architecture

- On the client side, it maps requests made by client entities from their personality-specific representation to a personality-independent one. This neutral representation is then passed to the neutral core for further processing; results are translated back from neutral to personality-specific form.
- On the server side, it receives requests for local objects from the core layer, assigns them to actual objects, and returns results.

Protocol personalities handle the mapping of requests (representing interactions between application entities in a personality-neutral fashion) onto messages exchanged through a communication network, according to a specific protocol. The requests are received either from application entities (through an application personality and middleware core), or from another protocol personality, in which case PolyORB acts as a proxy performing protocol translation between third-party nodes.

The Neutral Core Middleware acts as an adaptation layer between application and protocol personalities. It manages execution resources and provides 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 protocol personalities.

By construction, the core layer acts as dynamic gateway to allow application and protocol personalities to mutually use their services. 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.

2.2 Architecture

Schizophrenic middleware requires a flexible implementation, So PolyORB extensively rely on reusable modules: whenever possible we use design patterns to facilitate code maintenance and readability. We use component-oriented programming, meaning that services are built according to the design pattern 'component', communication between

them is done via synchronous message exchange. The TAO project has demonstrated how design patterns can be powerful tools [SC97]. We have also experienced in our own projects like GLADE [PT00] how they can be used to allow configurability for real-time tasking profile [DB98]. With the help of design patterns, we also achieve the schizophrenia properties.

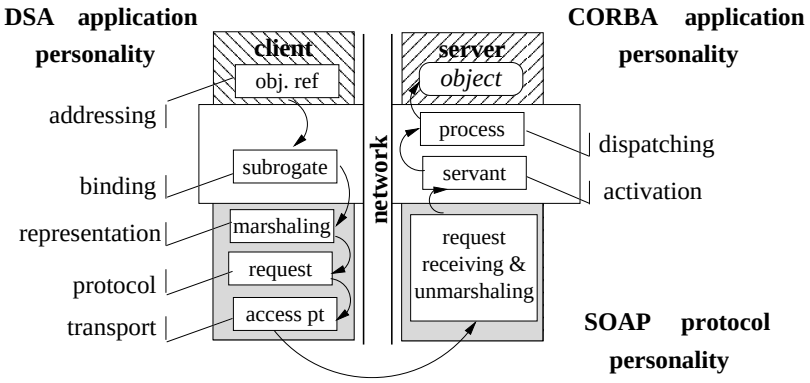

Fig. 3. Invocation request path

Basic services: PolyORB's core and personalities design reflects customization, interoperability, and genericity. We have pointed out in [QPK01] fundamental services in distributed applications. We have extended it and list now seven different services.

Figure 3 presents an example of interaction between these services. This scenario demonstrates the successive steps of a method call where a DSA application invokes a method of a CORBA object using the SOAP protocol.

– **Addressing** Each entity is given a unique identifier within the entire distributed application.

  Before any invocation calls on the CORBA object, the DSA application personality looks up the DSA reference associated to the CORBA object.

– **Binding** Middleware establishes and maintains associations between interacting objects and resources allowing this interaction, e.g. a socket, and an associated protocol stack. This service is inspired in part by the ODP binding [ODP95,BS97] and in part by the binding protocol of the SOR system [Sha94].

  In the example, the middleware core on the client side creates an object that comprises a socket and a SOAP protocol machine, and acts towards the application on behalf of the remote object it is bound to. Local calls to this "placeholder", or

surrogate, object are translated to communication with the middleware node that supports the remote object.

– **Marshaling** Request parameters must be translated into a representation suitable for transmission over network.

  The DSA function issued a request on the CORBA object. The request parameters are marshaled from a PolyORB independent representation to a protocol personality dependent representation, here SOAP.

– **Protocol** Middleware implements a protocol for the transmission of requests amongst nodes.

  Marshaled parameters and invocation meta data (caller and callee information, etc) are serialized.

– **Transport** A communication channel is established between a node and an object to transmit requests.

  The SOAP protocol personality opens a socket to the remote node.
  On the remote node, the request is received: Transport, Protocol and Representation services unpack and unmarshal it. It is then passed to the core middleware.

– **Activation** Middleware ensures that a concrete entity implementing objects is available to execute the request.

  The core middleware calls the CORBA object adapter, it ether finds an available servant, or create one.

– **Dispatching** Middleware assigns execution resource to process every incoming request.

  An executing thread is affected to the incoming request, depending on PolyORB's tasking policy. The request is then executed by the CORBA object.

The core middleware provides an implementation for all these basic services. These can be used as is by personalities, or extended to provide applications with specialized functionality.

Advanced Services: Some distributed model also provides advanced services. They are facilities that solve higher level problems specific to distribution:

– **Naming** services provide association between entities references and symbolic names, e.g. CORBA COS Naming, JNDI.

- **Termination** services determine consensus on whether a distributed application has completed its task or not. Such as service is present in `GLADE`, the 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.
- **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, i.e. the set of interactions that they support, and the types of their parameters. An example of such service is the CORBA Interface Repository:

Personalization: Various distribution models are implemented by creating personality modules that are built on top of PolyORB's core functionalities. Genericity of the middleware core allows reuse of common services. First studies showed that up to 65% of the code used to build CORBA middleware on top of PolyORB's architecture comes from the neutral core middleware.

Moreover, since the neutral layer decouples relationships between personalities, interoperability is obtained in constant cost; there is no combinatorial explosion when adding a personality even if it needs to interoperate with several others.

SOAP and GIOP protocol personalities, and CORBA and DSA application personalities, have been implemented and used to demonstrate full interoperability. To assess the full spectrum of schizophrenic middleware capabilities, a MOM personality was contemplated as a test case. Further sections give an overview of MOM architectures, then detail our work to integrate a MOM personality to PolyORB and discuss interoperability between MOM and ORB.

3 Case studies in MOM architectures

Message Oriented Middleware are distributed systems based on messages exchange. In some respects, they extend the mailbox model to applications. They are used in wide-scale applications like several information systems [Inc00]. Messages represents data meaningful either for the emitter, the receiver, or the MOM. They are used to save, route, deliver or get information amongst application nodes scattered across a network.

Typical MOM characteristics are well-identified. Among them, we find message tranport protocols, routing policies and some MOM-related services (naming services, ...). Unfortunately, each vendor used to come with its own API and a closed architecture. WebsphereMQ (formerly MQSeries[1]) from IBM has become the de facto standard although different vendor-dependent solutions are also available. All of these differ in their message passing mechanisms and their transport policies. Java Message Service [SUN99] (JMS) is the only generally adopted set of public specifications It provides a common API and a general MOM architecture, that covers all of these points.

[1] *http://www-3.ibm.com/software/ts/mqseries*

Message passing mechanisms fall into three categories, depending on the way messages are sent:

- **message exchange:** messages are directly delivered to the receiver. Message deliverance is one to one.
- **message queues:** messages are delivered to queues, then read by or sent to the recipient. Message deliverance is one to one, often called *Point-to-Point* (P-t-P)
- **message topics:** messages are delivered to topics, then read by or sent to one or more registered clients. Message deliverance is one to many, often termed *Publish/Subscribe* (Pub/Sub)

The Queues and Topics models differ only by the number of clients who can access them and read their content. A queue can be read by only one client, whereas topic can be read by all of clients registered as subscribers. In this paper, we use the term *message pool* to denote either a message queue or a message topic.

Transport policies control how a client sends, receives or destroys message. A client can use:

- **synchronous protocols:** the client remains blocked during message emission or reception. Such protocols are seldom used, they require rendezvous mechanisms to synchronize client and message pools, and diverge from message passing paradigms.
- **asynchronous protocols:** transport services do not block client exception and handles concurrently message passing functions.
- **call-back mechanisms:** the client is notified by a call-back mechanism that a new message has arrived in one of its message pools.
- **group communication:** message call-backs are broadcasted to multiple subscribers using a group communication service such as multicast, peer to peer, etc.

Figure 4 presents a typical MOM architecture. *Clients* create, send and receive messages by means of an *interface* based on a software and network architecture to route and store messages. This underlying infrastructure is divided into message pool servers, and a *provider*, which handles the underlying network architecture federating clients and servers, and controls access to message pools as well as message routing and transport (figure 4).

This decoupling between these two functions allow a great variety of designs. We describe some of them, focusing on their similarity to PolyORB design.

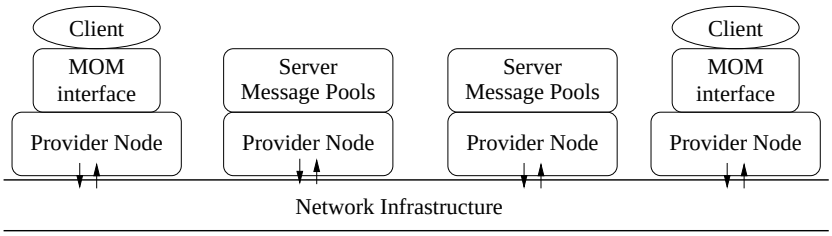

Fig. 4. MOM's canonical architecture

3.1 JMS: one specification, multiple architectures

Sun's Java Message Service is the first publicly available set of specifications provided as a standardized API for both P-t-P and Pub/Sub message passing models. JMS's goal is to provide a simple solution to use MOM in Java. It precisely describes the different steps involved in MOM messages life-cycle: creating, sending, receiving, reading or destroying messages. 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 is implemented the API. Thus, each JMS implementation specifies its own provider, leading to incompatibilities amongst them. Different JMS may not be able to interoperate. Nevertheless the standardized API provides portability of existing applications from a JMS implementation to another.

JMS providers can either be new MOM infrastructures, only supporting JMS API or existing vendor products extended to support this API. This demonstrates flexibility in the design of the JMS provider: most of MOM are now adapted to support the JMS API, other platforms use agents (JORAM [Obj98]), peer to peer technologies (OpenJMS [exo00]).

To some extent, JMS is similar to a PolyORB application personality: it provides the required interfaces to exchange messages. But, the underlying concretization of its interfaces covers the whole distribution logic and thus implies private architecture and implementation choice.

3.2 xmlBlaster: one MOM, multiple protocols

The Open Source project xmlBlaster [xml00] proposes its own MOM API and architecture. Written in Java, this middleware offers both P-t-P and Pub/Sub models, messages are described with XML-encoded meta information.

The main objective of this platform is to offer multiple transport protocols. Thus, communication between a xmlBlaster node and applications can be done using CORBA, RMI, XML-RPC, SOAP, e-mail (SMTP) or raw sockets. The application is independent

from the protocol layer, it is a configuration option set by the application developer or during deployment. It then provides MOM services to a wide range of clients, from CORBA objects to e-mail robots or Web applications. Such an heterogeneity amongst proposed protocols is uncommon.

xmlBlaster is very flexible: protocols for incoming and out-coming message can be different, allowing greater flexibility in application configuration. This is useful for Web applications, a message sent as a SOAP request can be replied by e-mail.

Its message pools can be accessed using CORBA, hence their primitives are available from different programming languages, the only requirement is the availability of a CORBA mapping for this language. Currently C, C++, Java, Perl and Python are available. Thus, this architecture demonstrates that a MOM can be fully implemented over CORBA.

The multiple protocols approach proposed by xmlBlaster is similar to PolyORB's one : pools are implemented in xmlBlaster's engine and communicate with its clients through different plug-ins (i.e. instantiation of a protocol). Hence, this design provides the same decoupling as for PolyORB's core and protocol personalities.

3.3 CORBA Notification: CORBA & MOM

Initial CORBA specifications involved coupling between client and server: communication is synchronous and Point-to-Point [GCSO01]. Such a communication model implies synchronization between nodes: they shall be available at the same time to complete the data transmission. This is not optimal for event propagation within an ORB.

CORBA COS Event and its superset COS Notification have been introduced to solve these specific needs to support event-driven mechanisms, such as alarm signals, providing MOM Pub/Sub mechanisms.

The CORBA Event Service defines three distinct roles: the `Supplier` produces event data; the `Consumer` receives and process event data and the `Event Channel` is the abstract medium through which consumer and supplier asynchronously communicate.

This service decouples events suppliers and event consumers: events delivery from one point to another does not require them to know about each other. The Event Channel enables asynchronous communication between consumers and suppliers.

The CORBA Notification Service extends the COS Event Service and provides support for quality of service (QoS) as well as event filtering.

These services provide MOM Pub/Sub functionalities to CORBA objects. They both propose an approach common to PolyORB's one: to build a new distribution model, they use existing middleware services. In this case, CORBA provides its services to build a MOM and act similarly to PolyORB core middleware. However, they still use CORBA constructs and semantics, which may impair performance. Our design is similar, but rely on a reduced set of services as foundation to build middleware, and could potentially demonstrate better performance.

4 Building a MOM on top of PolyORB

The previous section shows that MOM can be implemented on top of very different architectures. We now detail how to build a MOM over PolyORB's architecture.

4.1 MOMA: MOM for Ada

There was no Message Oriented Middleware for Ada when we decided to implement MOMA. We decided to rely on existing concepts to design our own MOM for Ada. To do so, we adopted JMS specification and concepts as a foundation. It covers all typical MOM uses and relies on well known patterns. MOMA specification is a thick translation of JMS API to Ada 95.

Our goal was not only to implement a basic but functional MOM for Ada. We wanted to show through MOMA that MOM can be built on top of PolyORB. We also wanted to demonstrate that MOMA was able to interoperate with other distribution models. We took advantage of the schizophrenic architecture to enrich the classical MOM concepts in order to provide interoperability with other ORB.

In this section, we describe the design of Message Oriented Middleware for Ada (MOMA). A MOMA application is built around the following elements:

- **MOMA messages** carry information between clients. As a convenience, they are encoded in XML in order to ease messages readability.
- **MOMA message servers** handle the previously defined message queues and topics.
- **MOMA clients** use MOMA API to format, send or receive MOMA messages. They interact with MOMA message servers.
- **other clients** can use several protocol layers to exchange messages with MOMA servers. This requires PolyORB to be configured with the appropriate protocol layer or protocol personality.
- **Administration objects** are used to setup the MOM and to activate functionalities such as authentification or naming services.

MOMA messages are divided into three parts, a *header* field containing information to route the message, a *QoS* field that details the quality of service applied to this message and a *payload* that is the actual data carried by the message.

Payload can be of the following types:

- **Data Message** contains data marshaled from various Ada 95 types: predefined types, records, arrays or strings. We also provide support for raw data which may be used to marshal binary data such as an image.

– **Request Message** holds an invocation request on an entity visible in the middleware instance. In some respects, this may represent a method invocation on an object present on the destination node.

A Data Message carries data to its destination. In this case, the receiver is in charge of parsing and processing the message. We say that this message is implicitly deliver to the receiver.

A Request Message is an invocation request sent to particular objects and requires a specific handling. Two processing policies are available and they reproduce the call-back mechanisms presented in section 3 :

1. The arrival of such a message may result in an implicit invocation call. In this case, the receiver polls on the message pool and executes the invocation call itself.
2. The object receiver registers a call-back in the MOM. At the request message arrival, the MOM activates this call-back in order to execute the invocation method.

4.2 Mapping MOMA functionalities to PolyORB

As for JMS, a provider supports MOMA distribution functions like all the message passing mechanisms. Moreover, MOMA's provider handles distribution issues raised by MOMA servers and clients communication, as well as message pools.

Let us give an overview of MOMA communication core mechanisms and their articulation. In a canonical MOM (see figure 4), a MOM interface provides primitives to clients and allows them to interact with message pools through requests (e.g. to post and receive messages). In between, an associated protocol transports them. We can then identify PolyORB's personalities to handle these two aspects of MOM: message pools are part of an application personality, whereas a protocol personality addresses transport mechanisms.

The MOMA provider is then the combination of these two personalities and the core middleware: each one provides services to mask distribution. We detail this point in section 5.1.

Application personality: Interactions between a client and a message pool are similar to method calls on objects in a distributed application. We can then define a MOM application personality implementing message queues and topics mechanisms. MOM clients will interact with these entities to send and receive their messages.

Thus, this personality propose two entities `Queue` and `Topic`, that respectively provide P-t-P and Pub/Sub message passing models. Their primitives are:

Publish	to post a new message to a pool
Get	to a/synchronously get messages from a pool
Delete	to delete messages from a pool
Subscribe	to subscribe to a topic
Unsubscribe	to unsubscribe to a topic

Protocol personality: Another key point is the definition of a protocol personality. It can support several transport mechanisms: synchronous or asynchronous requests, callback mechanisms and group communication. None of them are mandatory to build a MOM, even if asynchronism and group communication provide better performance in most cases. Protocol personality actual capabilities depend on the performance expected for MOM. Its definition is then minimal: the protocol personality provides mechanisms to transport requests between a MOM client and a message pool.

Hence, existing protocol personalities are sufficient for our purpose.

MOMA related services: MOMA personalities only detail how clients and pools communicate, and operation on pools.

Yet, MOMA does not only provide facilities for raw message passing: it also provides numerous annex services, which are not directly linked to distribution but still required for normal operations:

- **Message formatting:** message pools do not need to know how payload messages are built, hence they are only data stored but not used by both application and protocol personalities. Only MOM clients requires a message formatting API, it is then defined outside of MOMA personalities.
- **Naming service:** MOM clients should use a naming service to get references to message pools from a symbolic name.
- **Authentification service:** MOM clients access to message pools is restricted to authorized clients only, who provide correct credentials.
- **Administration service:** MOM message pools require to be set up either on startup time, or during application execution. The Administration service provides control to define message pools characteristics such as pool size, time out delay before message deletion, number of concurrent clients, etc.

Other services linked to MOM life-cycle can be defined, such as logging, redundancy services. Nevertheless, they are not MOM core services. These services could be distributed or not. Yet, this is a configuration issue out of the scope of this paper: it would require to precisely define Quality of Services requirements for each services.

Thus, we have proposed a mapping of MOMA functionalities to PolyORB layered structure. It shows that application personality is the only PolyORB specific part required to implement MOMA, we reuse existing protocol personalities. We defined MOM annex services, but did not assess their role in PolyORB's architecture.

5 Implementation and discussion of the architecture

Message pools are a key mechanism in the implementation of MOMA. We have several options to implement them: over an existing application personality, or in a new dedicated one.

Building MOM over an ORB has already been addressed by xmlBlaster or CORBA COS Event: MOM is built on top of an existing ORB by registering MOM entities with the ORB as object implementations; clients and message pools interact using the ORB's method calls. Hence, one might consider using the CORBA or DSA application personality. However, we want our MOM to act in a MOM-only distributed application or in a heterogeneous middleware instance, mixing ORBs and MOM.

We want our implementation to be light enough to provide good performance. Using CORBA or DSA application personalities to implement message pools may not be optimal : they introduce mechanisms to handle a large set of functions such as dynamic invocation, exception handling. MOM primitives are clearly defined and do not require these mechanisms in a MOM-only architecture. We chose not to use existing application personalities and design a specific one, that provides the minimal set of required functionnalities to support message pools primitives. We then show how message pools implementation provides interoperability with ORB implementation objects.

Then, we demonstrate how these interacting objects lead to interoperability between MOM and ORB distribution models and provide a few test cases for which this feature is interesting.

5.1 Implementation of MOMA

In the previous section, we restricted the implementation of MOMA to the creation of an application personality for the implementation of a message pool. We now detail how we actually build one.

We want MOMA personality to be as light as possible. Yet, a MOMA distributed application also requires the basic services we defined.

PolyORB's layered design concentrates in its core and protocol personalities some of the services a middleware should implement: when a PolyORB node receives a request for a local object, a protocol personality handles the request, unmarshals the data and creates a PolyORB request, execution request on a local object, independent from any distribution models. Thus, we reuse addressing, binding, marshaling, protocol and transport services from the core or existing protocol personalities.

The activation and dispatching services have to relay this request to the servant concretizing a MOMA message pool for execution. This is the only part specific to this distribution model.

We chose a simple implementation of these services: activation is done on entities created at startup time, dispatching service associates one thread per message pool. Thus, we provide a way to create simple servants, sufficient to holds message pool implementation entities and to execute primitives on them.

Hence, the MOMA provider is the combination of both core middleware and application, and protocol personalities: they all provide services required to mask distribution issues.

Schizophrenia as a conception scheme lets us concentrate only on the object implementation: genericity alleviates distribution problems: they are already solved by existing components. Thus, PolyORB let us rapidly prototype our MOM, and concentrate only on MOMA functionnalities instead of distribution issues.

MOMA message pools and administration objects are the fundamental elements of our distributed model. As part of a PolyORB's middleware architecture, they will be visible from other application personalities: an ORB client can send an invocation request on a MOMA message pool, and interacts with it to send or receive messages; a MOMA message pool can send a call-back invocation request to an ORB object.

Interaction between these two elements is similar to the invocation scheme described section 2.2. We thus provide interoperability between MOMA message pools and ORB objects. Yet, we need to assess the extent of this mechanisms, and how we can use MOM & ORB distribution models simultaneously in a distributed application.

Benefits from schizophrenia: Our implementation demonstrates basic MOM functionalities as well as effective interoperability between various distribution models through dynamic gateways. Thus, schizophrenia clearly alleviated most of the required work.

Moreover, the development process shows how genericity allows some feature to be reused by others personalities: initially, PolyORB's core lacked support for asynchronous requests. Thus, neither CORBA nor DSA could make asynchronous calls. PolyORB's design allowed us to add it as part of the core middleware, thus all personalities gained this feature. We did not only implement MOMA features into PolyORB's architecture, we also make CORBA and DSA benefit from it. Hence, CORBA can now use oneway operations, DII, DSA can now use the pragma `Asynchronous`.

So, our platform does not only demonstrate interoperability between distribution models, but also how a large part of the mechanisms can be factored out of personality specific module and then reused: MOMA implementation was reduced to the development of pure application logic, generic layer provides most of required services.

Note to reviewers: final paper will detail this point, and provide figures on actual code reuse.

5.2 Assessment of Interoperability

In the previous section, we have detailed the implementation of MOMA message pools and showed how they can interoperate with ORB implementation objects. We now extend this result to MOM & ORB interoperability.

We have defined interoperability between distribution models as the capability of a middleware to allow entities defined within a given distribution model to interoperate with remote entities from a different distribution model. Such a definition is natural when the platform only involves ORB-like distribution model. We want to extend this notion and consider middleware architectures mixing MOM and ORB distribution models. This raises several problems, and leads to the question of interoperability between MOM, and between MOM and ORB.

In the first case, several solutions are already available and functional. For example, gateways between MOM such as SonicMQ bridge for MQSeries [Sof00] bring the required functions to route message transparently from one MOM infrastructure to another. They are tied to two MOM vendors, and thus provide limited interoperability. They lead to combinatorial explosion: a specific bridge is required for each combination of MOM, and may also impair scalability: a bridge is a bottleneck between two MOM infrastructures.

PolyORB, thanks to multiple protocol personalities, could also propose interconnection with other existing MOM, such as MPI or JMS implementations. This would require the instantiation of a protocol personality. We did not assess its feasibility: xmlBlaster already demonstrates such an architecture, introducing multiple protocol plugins as a way to interoperability.

Instead, we chose to study MOM and ORB interoperability. We first have to extend the definition of interoperability we gave to handle this case, and check its consistency.

PolyORB design makes MOMA objects available in the same middleware instance than CORBA or DSA objects, hence one entity can call entities of another object, whatever its application personality. This covers interoperability as previously defined above: a CORBA object can call MOMA pools primitives and interact with a message pool, and the Request Message allows a MOMA client to send make a method call on a CORBA object.

We can then extend this result and build distributed applications in which MOM and ORB entites interact. We detail some test case where a MOM-ORB interoperability may be interesting.

– A client, whatever its application personality, may send a processing request to a MOMA message topic entity to which several CORBA or DSA objects are registered. They will then process the request and return the result if any. Hence, a MOMA object may serve as a proxy object between a client and a server object, providing new services. Such a configuration may easily implement distributed computation, log or redundancy services. It may also allow a nomadic client to have its requests processed when it is off-line: it sends its request to a message pool, a server receives this request, processes it and then post the reply to the message pool. Later, the client accesses the results.

– As MOMA message queue objects are built on top of a minimal application personality, their methods are the low level primitives defined to directly access queues. We can then contemplate using MOMA message facilities to implement services for the CORBA personalities, for instance CORBA COS Notification or COS Events which provide MOM functionalities.

These scenarios are a part of the applications that may use both MOM and ORB architectures. Both paradigms can be combined to alleviate some of the problems set by large scale or nomadic applications and propose new features to a distribution model: ORB may benefit from MOM's solutions for off-line request processing, group communication; MOM may have their request processed by ORB implementation objects.

Hence, coupling MOM and ORB facilities in the same middleware architecture brings great flexibility to distributed application design, and provides interesting combination.

5.3 Discussion

We have detailled the implementation of our MOM, and gave some test cases describing how MOMA can interact with ORB personalities. Yet, our architecture does not cover all issues raised by MOM-only and MOM & ORB applications.

MOM-only applications MOMA still relies on the existing protocol personalities: CORBA and SOAP. The communication models they use are not designed to efficiently support one-to-many communications used by large Pub/Sub infrastructures. Without a convenient protocol personality supporting group communication, MOMA-only applications may not scale, limiting application performance. But this only an implementation issue: group communication semantic is fully integrated to MOMA API. Thus, our architecture allows us to extend MOMA to support full MOM functionnalities by providing the corresponding protocol personality.

Construction of requests We proposed test cases of MOM & ORB applications. Yet, interaction between MOM and ORB entities is limited by the knowledge of entities reference and their interfaces. We have defined advanced services: naming service and interface repository services to provide visibility on existing entities and interfaces. Naming service will simplify entities localization, interface repository will provide all the necessary information to construct a request, they will ease application deployment.

Payload message formatting In some cases, an ORB has to send a message to a MOM client, it needs to format the payload message. We defined the MOMA message formatting service, to provide the abstraction to build the payload, and implented it as a separate service, independent from PolyORB. This service must be visible by all object implementations, either from PolyORB, but also from other native middleware.

Hence, MOM & ORB interoperability lacks support for some advanced services to ease interactions between MOM clients, message pools and ORB objects. But the first steps to allow interoperability between these two distribution models are now completed, we have experienced it through simple test cases.

6 Conclusion

In this paper, we have defined *schizophrenic middleware*. Schizophrenic middleware extends the concept of generic middleware by allowing several personalities to exist simultaneously on the same middleware instance. Moreover, these personalities can interact in order to produce dynamic gateways. We defined an original architecture to support this concept. This architecture is based on the decoupling of application and protocol personalities by means of a neutral core layer. PolyORB, our free schizophrenic middleware, ensures an excellent code factorization ratio, and provides an interesting solution to the issue of interoperability between distribution models (M2M).

We first validated our work on interoperability between RPC-like distribution model such as CORBA or DSA. We have extended PolyORB to support personalization into a Message Oriented Middleware. An initial case study has demonstrated that the canonical architecture of classical MOM fits perfectly in the schrizophrenic architecture. Therefore, the design of the corresponding application and protocol personalities was quite natural. As an initial requirement, the application personality provides an Ada message passing interface modeled after JMS. PolyORB thus becomes the first free middleware providing a MOM solution for the Ada community.

As a consequence of schizophrenic properties, PolyORB enables MOM and ORB applications to interoperate. We have successfully demonstrated this interoperability between Ada CORBA and MOM applications and we are about to proceed to more general experiments. Moreover, this middleware is not dedicated to a specific protocol and can be configured for any classical protocols like GIOP, SOAP or raw sockets. We are currently extending PolyORB to provide some of the most frequent services available in classical MOM.

In the future, we also want to address *semantic interoperability* to let a component *A* based on distribution model DM_A interact with a component *B* based on distribution model DM_B the way it could do if *B* was based on DM_A. So far, this can only be achieved using a dedicated gateway translating DM_A protocol and behavior to DM_B ones. However, such a gateway is typically an ad-hoc component that cannot be reused. The goal of semantic interoperability is to extend the capabilities of the dynamic gateways provided by a schizophrenic middleware. This should increase the reuse of components in distributed applications.

References

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

[BS97] G. Blair and J. Stefani. *Open Distributed Processing and Multimedia*. Addison Wesley, 1997.

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

[DHTS98] B. Dumant, F. Horn, F. Dang Tran, and J-B. Stefani. Jonathan: an open distributed processing environment in java. In *Proceedings of the IFIP International Conference on Distribut ed Systems Platforms and Open Distributed Processing*, pages 175–190, Londres, 1998. Springer Verlag.

[exo00] exolab.org. Openjms, 2000. `http://openjms.exolab.org`.

[FM99] C. Francu and I. Marsic. An Advanced Communication Toolkit for Implementing the Broker Pattern. In *Proceedings of ICDCS'99*. IEEE, June 1999.

[GCSO01] Pradeep Gore, Ron Cytron, Douglas C. Schmidt, and Carlos O'Ryan. Designing and optimizing a scalable CORBA notification service. In *LCTES/OM*, pages 196–204, 2001.

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

[Inc00] MINT Communication Systems Inc. Financial middleware - from theory to reality, 2000. `http://www.simc-inc.org/archive9798/Apr20-1998/shefi.pdf`.

[ISO95] ISO. *Information Technology – Programming Languages – Ada*. ISO, February 1995. ISO/IEC/ANSI 8652:1995.

[Obj98] ObjectWeb. Joram – java open reliable asynchronous messaging - datasheet, 1998. `http://www.objectweb.org`.

[ODP95] ODP. ODP Reference Model: overview, 1995. ITU-T -- ISO/IEC Recommendation X.901 -- International Standard 10746-1.

[OMG98] OMG. *The Common Object Request Broker: Architecture and Specification, revision 2.2*. OMG, February 1998. OMG Technical Document formal/98-07-01.

[PQK+01] Laurent Pautet, Thomas Quinot, Fabrice Kordon, Samuel Tardieu, Fabien Azavant, Vincent Niebel, Sébastien Ponce, and Tristan Gingold. Polyorb, 2001. `http://libre.act-europe.fr`.

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

[QPK01] Thomas Quinot, Laurent Pautet, and Fabrice Kordon. Architecture for a reuseable object-oriented polymorphic middleware. In *Proceedings of PDPTA'2001*, Las Vegas, Nevada, Etats-Unis, June 2001.

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

[Sha94] M. Shapiro. A binding protocol for distributed shared objects. In *Proc. of the 14th Int'l Conf. on Distributed Computing Systems (ICDCS-14)*, pages 134–141, Poznan (Pologne), June 1994.

[Sof00] Sonic Software. Sonicmq bridge for mqseries user's guide, 2000. `http://www.sonicsoftware.com`.

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

[SUN99] SUN. Java message service, 1999.

[W3C00] W3C. *Simple Object Access Protocol (SOAP) 1.1* , May 2000. W3C note.
[xml00] xmlBlaster.org. xmlblaster, 2000. http://www.xmlblaster.org.