The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications

Fábio M. Costa, Hector A. Duran, Nikos Parlavantzas, Katia B. Saikoski, Gordon Blair, and Geoff Coulson

Distributed Multimedia Research Group,
Department of Computing, Lancaster University,
Lancaster, LA1 4YR, U.K.
{fmc, duranlim, parlavan, saikoski, gordon, geoff}@comp.lancs.ac.uk
http://www.comp.lancs.ac.uk

Abstract. The increasing complexity of building distributed applications has positioned middleware as a critical part of complex systems. However, current middleware standards do not address properly the highly dynamic and diverse set of requirements posed by important classes of applications, such as those involving multimedia and mobility. It is clear that middleware platforms need to be more flexible and adaptable and we believe that an open engineering approach is an essential requirement. More specifically, we propose the use of object oriented reflection based on a multi-model reflection framework as a principled way to achieve such openness. This leads to middleware that is flexible, adaptable and extensible, and, in consequence, capable of supporting applications with dynamic requirements.

1 Introduction

Engineering distributed applications is inherently more complex than non-distributed ones due to problems of heterogeneity and distribution. Middleware platforms aim to isolate developers from this extra complexity and have recently emerged as a critical part of distributed software systems. CORBA, DCOM and Enterprise Java Beans represent a few of the many competing technologies. The basic role of middleware is to present a unified programming model to developers that masks out distribution concerns and allows them to concentrate mainly on the application semantics.

The rapidly expanding visibility and role of middleware in recent years has emphasised the following problem. Traditional middleware is monolithic and inflexible and, thus, it cannot cope with the wide range of requirements imposed by applications and underlying environments. This is especially evident in the case of new application areas, such as multimedia, real-time, CSCW (Computer Supported Cooperative Work) and mobile applications, which have specialised and dynamically changing requirements. The problem has already been recognized and all current middleware architectures offer some form of configurability. However, this is typically piecemeal, ad-hoc and only involves selection between

a fixed number of options. Clearly, a more systematic and principled solution to the problem is needed.

We believe that an open implementation approach is essential for developing flexible and adaptable middleware platforms. More specifically, the solution we propose adopts object-oriented reflection as a principled way to inspect and adapt the underlying open implementation. While reflection has principally been applied to languages and operating systems, we are convinced that middleware is the most natural level for reflective facilities. Our object-oriented meta-level architecture is based on the idea of structuring the meta-space in terms of orthogonal meta-models. Each of these meta-models provides dynamic adaptability of an independent set of middleware implementation aspects. In addition, reflective computation is performed under strictly controlled scope, since each meta-level entity acts upon a limited set of base-level entities.

This paper is structured as follows. Section 2 explores current distributed object platforms. Section 3 presents our arguments towards the need for the open engineering of middleware. Section 4 briefly describes computational reflection, which is the basis for our proposal. In section 5 we explain our approach to providing a reflective architecture for middleware, while in section 6 we present some early implementation results, notably a prototype meta-object protocol. In section 7 we present some examples of how reconfiguration takes place in our approach. Finally, section 8 discusses related work, and section 9 presents some concluding remarks.

2 Distributed Object Technologies

Object oriented technology has been combined with distributed computing to create a new generation of client/server technology: *distributed object computing*. The central idea of distributed objects is to have servers that expose objects, and clients that invoke methods on these objects locally or *remotely*. Such an environment is normally realised by providing an infrastructure that allows objects to interact transparently, independent of being in the same or in different address spaces.

Three main technologies are widely used as distributed object infrastructures: CORBA[9], Java RMI[20] and DCOM[4]. CORBA (Common Object Request Broker Architecture) is the distributed object platform proposed by the OMG (Object Management Group) which contains an open bus – the ORB (Object Request Broker) – on which objects can interoperate. Sun Microsystems developed its own Java ORB, called Java RMI (Remote Method Invocation). RMI provides a way for clients and servers running the Java Virtual Machine to interoperate. Microsoft also proposed its own solution for distributed object computing: DCOM (Distributed Component Object Model). DCOM architecture is an extension to Microsoft's component model (COM). While COM defines components and the way they are handled, DCOM extends this technology to support components distributed across many computers.

EJB (Enterprise Java Beans)[19], COM+[5] and CORBA Components[11] represent the latest trend in middleware platforms, which simplify even more the development of distributed component-based applications. The significance of these architectures lies in that they achieve a separation of concerns between the functional aspects of the application and the non-functional aspects such as distribution, resource management, concurrency and transactions. The latter aspects are managed by a container and the developers are freed from writing system code that deals with them.

As regards to flexibility and adaptability, we can see a clear trend towards opening up the implementation of middleware. For example, the CORBA specification defines interceptors as a way to open up internal aspects of the invocation path, allowing the injection of additional behaviour. Similarly, DCOM offers custom marshalling, which enables developers to bypass the standard remoting architecture to create customised connections. Moreover, EJB, COM+ and CORBA Components support declarative configurability of non-functional system aspects (e.g. transaction policies) using attributes. In all these cases, we can observe that the configurability is piecemeal and done using ad hoc and static (compile or deployment time) mechanisms. The implementation of the platform is mostly hidden and out of the control of the application developer.

3 The Case for Open Engineering of Middleware

While some business applications are based on the interaction of objects to exchange text (database query/update, etc.), others have more general requirements such as the exchange of multimedia data or need for run-time adaptation. For example, a company may be selling video on-demand services to clients whose connections to the network may vary significantly. Some clients may have high speed network connection, while others have more limited ones. Moreover, during the provision of services, some clients may face network congestion problems that may lead to the need for some kind of adaptation. This means that, in this example, the network conditions play important role in the provision of the service. The price a client pays for the service and the service itself are different for different clients.

These kind of applications require some flexibility from the underlying system. This is not provided by current middleware platforms in a principled way. This happens because the approach taken emphasises *transparency*, meaning that application developers do not need to know *any* details about the object from which the service is being required. But it also means that the developers do not get to know the details about the platform that supports the interaction as well. Therefore, it is not possible to provide the flexibility required.

This *black-box* approach is not suitable for developing flexible applications. Thus, it is necessary to design a distributed object platform that exposes its internal details and allows modifications to the way the infrastructure handles the interactions, for example, the way middleware deals with resource reservation.

Based on these observations we claim that middleware should provide several extra features in terms of adaptability, that is, *configuration* as a means of selecting specific operation at initialization time and *reconfiguration* as a means of changing the behaviour at run time. This can be done by inspecting the internal details of a system and changing them. Another important feature is *extension*. This allows new components to be introduced in the middleware in order to handle situations not already encountered.

For example, consider the transmission of audio over a channel whose throughput may change over time. Firstly, a *configuration* can be applied to establish the communication according to the channel (quality of audio is proportional to the throughput). Secondly, once the transmission is already running, *reconfiguration* will allow a filter to be introduced in order to reduce the audio quality due to a network congestion. Reconfiguration can happen again in case the network recovers its normal operation.

One can claim that adaptation could be performed at the operating system or application levels. However, adaptation at the operating system level is platform-dependent and requires a deep knowledge of the internals of the operating system. In addition, unwanted changes at this level could be catastrophic as every single application running in a node could be affected. Furthermore, some research in operating systems [7] and networking [8] has advocated leaving as much flexibility and functionality to applications as possible in order to satisfy the large variety of application requirements. On the other side, adaptation at the application level imposes an extra-burden to the application developer. Besides, the adaptation mechanisms developed at this level cannot be reused since they are application specific.

This leads us to emphasise that adaptations should be handled at the middleware level which ensures at the same time platform independence and isolation from the implementation details.

Based on these issues, we can say that traditional middleware platforms such as presented in section 2 are not suitable for flexible applications. An *open engineering* of middleware platforms would be a proper way to overcome this problem. In the next section we briefly review the concept of computational reflection, which is the basis for our approach.

4 Computational Reflection

A reflective system, as defined by Maes [17], is one that is capable of reasoning about itself (besides reasoning about the application domain). This definition implies that the system has some representation of itself in terms of programming structures available at run time. The process of creating this self-representation is known as reification. In addition, this self-representation needs to be causally connected with the aspects of the system it represents, in such a way that, if the self-representation changes, the system changes accordingly, and vice-versa.

In a reflective system, we call the base-level the part of the system that performs processing about the application domain as in conventional systems,

whereas the meta-level is the term used to refer to the part of the system whose subject of computation is the system's self-representation. In an object-oriented environment the entities that populate the meta-level are called meta-objects, which are the units for encapsulation of the self-representation and the associated reflective computations. Thus the process of reification of some aspect of the system normally results in a meta-object being created. A further consequence of object-orientation is that reification can be applied recursively, such that meta-objects, as normal objects, can have associated meta-meta-objects and so on, opening the possibility for a tower of meta-objects, where each meta-object allows reflection on the meta-object below it. In practice, however, only a few levels are necessary and the topmost level can be hard-coded in the execution environment.

5 Our approach

5.1 Applying Reflection to Middleware

As middleware platforms are just another kind of software platforms (such as programming languages and operating systems), applying reflection to them may follow the same principles. One has to identify the concepts and structures that compose the base-level, as well as the way they are going to be represented at the meta-level.

The base-level of a middleware platform can be considered as the services it provides, either implicitly or explicitly through its interfaces. Examples of such services are the communication of interactions among objects (with all the implicit activities it involves, such as marshalling, dispatching and resource management), the supporting services (e.g. name resolution and location, management of interface references, distributed object life cycle, relocation, migration, etc.), and other common services (e.g. security, transactions, fault tolerance and trading). Such services are usually modelled and implemented as cooperating objects, and the platform takes the form of a configuration of objects (although such a configuration is not normally visible at the base-level).

The meta-level consists of programming structures that reify such configurations of services and allow reflective computation to take place. It must cover the different aspects that are involved in middleware, such as the topology of configurations, the interfaces of the objects providing services, and the behaviour of such objects. The meta-level makes it possible to inspect the internals of the middleware and to alter its implementation, by means of well known meta-interfaces. For example, a new service can be added to the platform or an existing one can be reconfigured by appropriately using operations available at the meta-interfaces. The sources or invokers of such reflective computations may be the application which is running on top of the platform, or the platform itself, which may have mechanisms for quality of service monitoring that can trigger control actions involving reflection.

5.2 The Underlying Object Model

Both base- and meta-level of our reflective architecture for middleware are modelled and programmed according to a uniform object model. In what follows, we present the main concepts of this object model.

The basic constructs of the model are objects, which are also the units of distribution. Objects may have one or more interfaces, through which they interact with other objects, and each individual interface is identified by a unique interface reference, which contains enough information to allow the location of the implementation object and to get a description of the services it provides through the interface.

The programming model supports three kinds of interfaces in order to model the different styles of interactions an object may provide:

- *Operational interfaces* are meant to enable client-server style interactions, based on method invocation semantics. They consist of the signatures of the methods provided by the interface and (optionally) the methods required from other bound interfaces.
- *Stream interfaces* allow interactions by means of continuous flows of data, usually with temporal properties. Such a style of interactions is usually found in multimedia applications that involve the communication of audio and video data.
- *Signal interfaces* are meant to support one-way interactions that may have temporal (i.e. real-time) properties. The signal interaction style is more primitive than the other two, and may be used to model or implement them.

Objects may only interact through their interfaces, and such communication is achieved, primarily, by means of *explicit bindings*, which are first class objects in the model and encapsulate the services related to communications. Explicit bindings may be local (between objects in the same address space) or distributed (spanning different address spaces or machines) and their establishment may be requested either by a third party or by one of the objects to be bound.

The use of explicit binding objects is essential in order to provide a level of control over continuous media communications. This is due to the fact that a binding object is normally identified with an established connection among two or more parties. Explicit bindings allow, therefore, the exposure and manipulation of the mechanisms and properties involved in multimedia interactions, setting the context for their monitoring and adaptation.

However, for interactions with a transient nature, where the overheads or the programming effort involved in explicit binding establishment are often inconvenient, the model also supports *implicit bindings*. At the level of the programming model, an implicit binding is transparently created when an object gets an active interface reference of another object and uses it as a pointer to access its functionality (similarly to the way CORBA objects interact). As a consequence, implicit bindings provide no control over their internal implementation and are therefore unsuitable for applications requiring guaranteed levels of service [2].

Finally, quality of service (QoS) properties can be associated with the interactions among objects. This is supported by QoS annotations on the interacting interfaces and by the negotiation of contracts as part of the establishment of bindings. A complementary QoS management mechanism may then be used to carry out monitoring and control tasks based on such contracts.

5.3 Structure of the Meta-level

Basic Principles. As a consequence of the programming model introduced above, the basic constructs for reflective computation are *meta-objects*. The collection of the interfaces provided by the meta-objects are then referred to as the *meta-object protocol*, or simply MOP. In addition, as meta-objects themselves are objects, they can be reified as well (in terms of meta-meta-objects).

Procedural reflection is used, meaning that the reified aspects of the platform are actually implemented by the meta-objects. This is a somewhat primitive but powerful approach since it allows the introduction of completely new behaviour or functionality to the platform. Additionally, more elaborate declarative meta-interfaces can be built on top of the procedural meta-interfaces.

Another principle of our approach is the association of meta-objects with individual base-level objects (or set of objects) of the platform, with the purpose of limiting the scope of reflective computation. In this way, the effects of reflective computation are restricted to the reified objects, avoiding side-effects on other parts of the platform. Access to the meta-objects is dynamically obtained using a basic part of the meta-object protocol provided by the infrastructure, and making reference to the interface of the object being reified. If a requested meta-object does not exist yet, it is instantly created, which means that meta-objects are created on demand. Therefore, not necessarily all base-level objects have to have the overheads associated with reflection.

Finally, a crucial design principle is the adoption of a *multi-model reflection framework*, by applying separation of concerns to the meta-level itself. Several orthogonal aspects of the meta-level are identified and modelled by means of distinct meta-objects. This was motivated by the great complexity involved in a meta-level for middleware. By modelling (and implementing) each aspect as an independent meta-model, both the meta-level design and the use of reflective computation are made simpler. The concept of multi-model reflection was first introduced in AL-1/D [21], which defines a set of meta-models to deal with the different aspects normally found in distributed systems. AL-1/D, however, does not stress the importance of orthogonality between the meta-models, a feature which greatly improves their manageability. Currently, in our architecture, we have defined four meta-models, which are the subject of the next section: encapsulation, compositional, environment, and resource management. However, the framework is open, in the sense that other meta-models can be defined in order to model other aspects of middleware that may be identified in the future.

Meta-models. As discussed above, the meta-level is divided into independent and orthogonal meta-models, each one representing a different aspect of the plat-

form. We categorise these meta-models according to the two well-know forms of reflection: structural and behavioural (also know as computational) [24]. In our architecture, structural reflection deals with the configuration of the platform, in terms of which service components it has (compositional meta-model), as well as the interfaces through which the services are provided (encapsulation meta-model). Behavioural reflection, on the other hand, deals with how services are provided, in terms of the implicit mechanisms that are used to execute invocations (environment meta-model) and the association of resources with the different parts of the platform (resources meta-model). The structure of the meta-space is illustrated by figure 1. In what follows, we analyse each of the meta-models in detail.

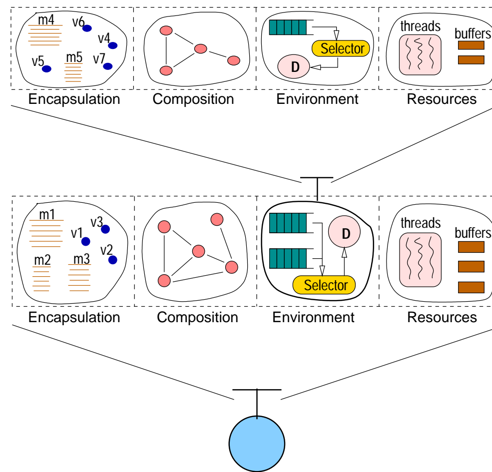

Fig. 1. Overall structure of meta-space.

Encapsulation. The *encapsulation meta-model* provides access to the representation of a particular interface in terms of its set of methods, flows or signals, as well as the interface attributes. It also represents key properties of the interface including its inheritance structure.

The level of access provided by a particular implementation of the encapsulation meta-model is dependent on the dynamic facilities that are present in the programming environment. For example, with compiled languages such as C access may be limited to inspection of the associated IDL interface. With more open languages, such as Java or Python, more complete access is possible, such as being able to add or delete elements (methods, flows, signals, and attributes) to or from an interface. This level of heterogeneity is supported by having a type hierarchy of meta-interfaces ranging from minimal access to full reflective access to interfaces. Note, however, that it is important that this type hierarchy is open and extensible to accommodate unanticipated levels of access.

Compositional. The *compositional meta-model* provides access to a compound object in terms of its configuration of components. The composition of an object is represented as an *object graph* [14], in which the constituent objects are connected together by edges representing local bindings. Importantly, some objects in this graph can be (distributed) binding objects, allowing distributed configurations to be created.

The compositional meta-model provides facilities for inspecting and manipulating the structure of the object graph, allowing access to individual components, as well as to insert or remove components. This meta-model is of crucial importance since it allows one to reify the configuration of the middleware platform and adapt it dynamically (such as to add new services). In particular, it allows one to reify the configuration of distributed binding objects, making it possible to inspect and change the internal binding components that implement the interaction mechanisms.

Environment. The *environment meta-model* represents the execution environment for the interfaces of objects in the platform. In such a distributed environment, this corresponds to functions such as message arrival, message queueing, selection, dispatching, and priority mechanisms (plus the equivalent on the sending side). In a similar design, the ABCL/R reflective language [24], collectively refers to these functions as the computational aspect of the base-level object.

Different levels of access are supported by environment meta-objects, depending on the dynamic capabilities of the execution environment. For example, a simple meta-model would only deal with the configuration of parameters of the existing mechanisms in the environment, such as the sizes and priority levels of the queues of arriving messages. More complex meta-models, however, would allow the insertion of additional levels of transparency or control over the actual implementation of such mechanisms. As with the encapsulation meta-model, this level of heterogeneity is accommodated within an open and extensible type hierarchy.

A further and crucial characteristic of the environment meta-model is that it is represented as a composite object. Hence, the environment aspect of the meta-space may be inspected and adapted at the meta-meta-level using the compositional meta-model. An example of such kind of adaptation would be to insert a QoS monitor at the required point in the corresponding object graph.

Resource Management. Finally, the *resource meta-model* is concerned with both the resource awareness and resource management of objects in the platform. Resources provide an operating system independent view of threads, buffers, etc. Resources are managed by resource managers, which map higher level abstractions of resources onto lower level ones.

Crucially, we introduce *tasks* as a logical activity that a system performs, e.g. transmitting audio over the network or compressing a video image. Importantly, tasks can span both object and address space boundaries. Each task in the system has a representation in the resource meta-space in terms of a *virtual task machine* (VTM). There is a one-to-one mapping between tasks and VTMs. Thus,

a VTM represents all the abstract resources that a task uses for execution, such as threads and memory buffers. In addition, VTMs represent a unit of resource management, and may be seen as virtual machines in charge of executing their associated tasks.

The resources meta-space can therefore be viewed as three complementary hierarchies representing respectively resources, resource factories and resource managers at different levels of abstraction. The first one is the abstract resource hierarchy. VTMs are at the top of this hierarchy with raw resources at the bottom. Resource factories provide a second type of hierarchy. VTM factories are represented at the top of the hierarchy, whereas factories for lower level abstract resources lie at the levels below. Thirdly, we have a manager hierarchy. Managers defined at the top of the hierarchy are in charge of managing higher-level resources, e.g. VTMs, whereas managers next to the bottom manage lower-level resources, e.g. processors.

As an example, consider a particular instantiation of the resource framework shown in figure 2 below.

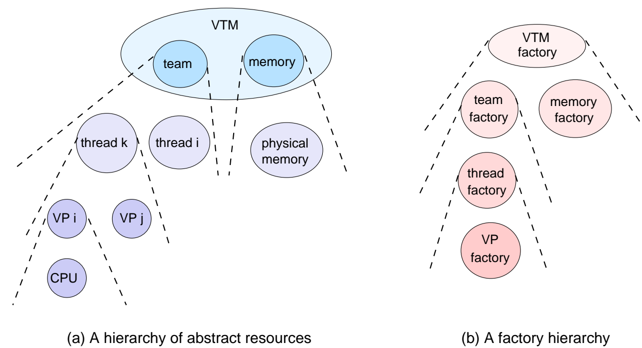

Fig. 2. Structure of the resources meta-space.

Firstly, the VTM has a hierarchy of processing resources, as shown in figure 2(a), where *user-level threads* run on top of *virtual processors* (kernel-level threads). At a higher level, a *team* abstraction represents two or more user-level threads within the VTM. Finally, at the bottom of this hierarchy we have a representation of a physical processor. Secondly, the VTM also has an abstract resource representing a memory buffer pool (*mem*), which in turn is a higher abstraction of physical memory. In a similar way, there is a hierarchy of abstract resource factories, as shown in figure 2(b). The VTM factory is composed by both the team factory and the memory factory. The team factory uses a thread factory, which in turn is supported by a virtual processor factory.

A complete description of the resource management entities that populate this meta-model can be found in [1] and [6].

6 Prototype Meta-object Protocol

We have implemented a significant subset of the architecture in order to allow the use and demonstration of its reflective features. The prototype consists of a base platform, which offers services for creating distributed bindings objects in order to support the communication between user entities. In addition, it provides meta-objects that implement each of the four meta-models described above, allowing dynamic adaptation of the platform. The implementation environment was based on Python, an interpreted object-oriented language that offers a powerful and flexible programming model which facilitates the implementation of the reflective features.

The prototype implements a simple meta-object protocol with enough expressiveness to illustrate the use of the meta-objects in meaningful scenarios. The prototype MOP consists of a basic part, which provides access to the meta-objects, and four other parts, which are specific to each of the meta-models respectively. We see this as an initial MOP, from which more elaborated ones can be devised following the idea of the hierarchy of meta-object protocols mentioned above. In what follows we present each of these parts of the MOP.[1]

Basic MOP. The basic part of the meta-object protocol, presented in table 1, consists of methods which are directly provided by the infrastructure.

Table 1. Meta-object protocol – basic part

Method	**Description**
`encapsulation(interface)`	return a reference to the encapsulation meta-object that reifies the given interface
`composition(interface)`	return a reference to the compositional meta-object that reifies the given interface
`environment(interface)`	return a reference to the environment meta-object that reifies the given interface
`resources()`	provide access to the resource meta-space corresponding to the current address space; the operation returns a reference to the top-level abstract resource factory, i.e. the VTM factory.

As the table shows, the basic MOP has one method for each of the four meta-models, which returns an interface reference to the appropriate meta-object corresponding to a specific base-level entity. Note that in the case of the first three meta-models, the base-level entity corresponds to an interface, whereas for the resource meta-model, an address space is the base-level concept that is reified. Also note that if the meta-object does not exist, it is created.

[1] Note that all excerpts of code that appear hereafter, including the method signatures in the MOPs, are presented in Python syntax.

Encapsulation MOP. The encapsulation part of the meta-object protocol (described in table 2) consists of methods to access and manipulate the reified features of an interface's encapsulation. These methods are all provided at the interface of the encapsulation meta-object.

Table 2. Meta-object protocol - encapsulation part

Method	**Description**
`inspect()`	return a description of the methods and attributes of the base interface
`addAttribute(attr_name, type, mode)`	add a new attribute to the base interface, with type and mode as given
`delAttribute(attr_name)`	remove the attribute from the base interface
`getAttribute(attr_name)`	return the value of an attribute
`setAttribute(attr_name, value)`	set the value of an attribute
`addProvMethod(method_sig, function)`	add a new method (of signature method_sig) to the row of methods provided by the interface
`delProvMethod(method_name)`	remove a provided method from the interface
`addReqMethod(method_sig)`	add a new method to the list of methods required (i.e. imported) by the interface
`delReqMethod(method_name)`	remove a required method from the interface
`addPreMethod(method_name, function)`	register the function as a pre-method for the named method
`delPreMethod(method_name)`	remove the pre-method of the named method
`addPostMethod(method_name, function)`	register the function as a post-method for the named method
`delPostMethod(method_name)`	remove the post-method of the named method

Composition MOP. The compositional meta-object offers a meta-object protocol composed by the methods described in table 3. These methods are used for inspecting and manipulating the object graphs that represent the configuration of base-level compound objects.

Environment MOP. The meta-object protocol provided by the environment meta-object is as described in table 4. This is a fairly primitive MOP, in the sense that it does not offer a high level of structuring of the reified mechanisms in an interface's environment. Further extensions will, for example, allow the reification of particular mechanisms of the environment, such as queues of messages and dispatching mechanisms.

The current environment MOP is based on the concept of *before* and *after* computations, which consist of arbitrary processing that wraps the execution of every message received by a given interface. (Note that this is different from pre- and post-methods, which are assigned to particular methods, not to the

Table 3. Meta-object protocol – compositional part

Method	Description
`getComponents()`	return a list with the description of the components in the object graph; this description contains the component name, its type, and its kind (stub, filter, binding, etc.)
`getComponentsByKind(kind)`	return the components that conform to the given kind
`getBoundInterface(interface)`	return the interface that is locally bound to the given interface, if any.
`getLocalBind(obj1, obj2)`	return the names of the interfaces by which the two objects are locally bound (if there is a local binding between them)
`addComponent(comp_descr, position)`	add a new component to the object graph; the description of the new component consists of its name, type, kind, and any initialisation arguments; the position for its insertion is given by the names of the two adjacent interfaces between which the component is to be inserted (note that a new component can only be inserted between two existing interfaces; if there are any other interfaces to which it needs to be bound, the current MOP requires the corresponding local bindings to be explicitly established afterwards)
`delComponent(comp_name)`	remove the named component from the configuration (note that any broken local bindings need to be subsequently repaired)
`replaceComponent(comp_name, comp_descr)`	replace the named component with a new one (described in comp_descr as above); this equals to deleting the old component, adding the new one, as well handling any extra local bindings
`localBind(interf1, interf2)`	establish a local binding between the two named interfaces, if they are compatible (this is simply another way of accessing the local binding service provided by the infrastructure)

Table 4. Meta-object protocol – environment part

Method	Description
`addBefore(name,function)`	add the function as a *before* computation in the environment of the base interface
`delBefore(name)`	remove the named *before* computation
`addAfter(name, function)`	add the function as an *after* computation in the environment of the base interface
`delAfter(name)`	remove the named *after* computation

whole interface.) The before computation can apply any sort of treatment to the message prior to delivering it to the interface's implementation object. Such treatment could be, for example, the delay of the message for a certain amount of time in order to smooth delay variations. Similar considerations apply to after computations. In addition, there can be a chain of before and after computations, where each individual one is identified by its own name. This functionality is similar to the idea of interceptors in CORBA.

Resource Management MOP. The meta-object protocol for the resources meta-model is described in table 5.

Table 5. Meta-object protocol – resources part

Meta -object	**Method**	**Description**
Factory	`newResource(size, mgntPolicy, schedParam)`	create an abstract resource of a given size and associate a management policy with it; scheduling parameters are passed in case of the creation of processing resources
Scheduler	`suspend(abstResource_id)` `resume(abstResource_id)`	suspend an abstr. processing resource resume an abstr. processing resource

Abstract resource	`getLLRs() /setLLRs(llrs)`
	`getHLR() / setHLR(hlr)`
	`getManager() / setManager(newMgr,param)`
	`getFactory() / setFactory(f)`
		get/set lower level resources
		get/set the higher level resource
		get/set the manager of this resource
		get/set the factory of this resource

As the table shows, the resources meta-space is composed by three kinds of meta-objects, according to the three complementary hierarchies of resource management discussed in section 5.3. Together, the meta-interfaces of these three kinds of meta-objects constitute the resources part of the MOP.

At each level of the resource management hierarchy, meta-objects implement these MOP meta-interfaces in a different way, depending on the abstraction level and on the type of the resources being managed. For instance, a team factory provides a different implementation for the operation `newResource()` than that provided by a thread factory or by a memory buffer factory. However, the overall meaning of this operation is consistent in all three cases.

7 Examples

In this section some example scenarios are presented to illustrate the use of the different meta-models to dynamically adapt the services provided by the platform.

Compositional Meta-model. Figure 3 shows a two-level binding object which provides the simple service of connecting the interfaces of two application objects. The purpose of such a binding may be, for example, to transfer a continuous stream of media from one object to the other. At run-time, some external monitoring mechanism notices a drop in the network throughput, demanding a reconfiguration of the binding object in order to support the negotiated quality of service. This reconfiguration may be in terms of inserting compression and decompression filters at both sides of the binding, hence reducing the actual amount of data to be transfered.

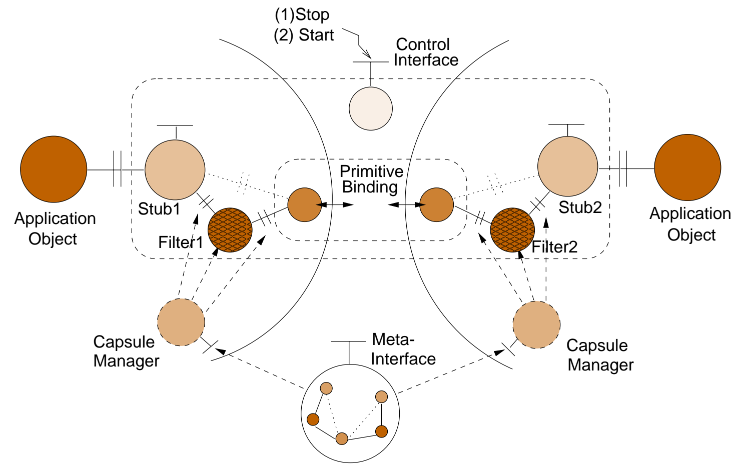

Fig. 3. Adaptation using the compositional meta-model

As the picture shows, the compositional meta-object (`meta_obj`) maintains a representation of the binding configuration (the object graph). The compositional MOP provides operations to manipulate this representation, and any results are reflected in the actual configuration of components in the binding object. In this particular case, the following two calls are made to the meta-object:

```
meta_obj.addComponent(filter1, (stub1.interf2, prim_binding.interf1))
meta_obj.addComponent(filter2, (stub2.interf2, prim_binding.interf2))
```

By having a QoS control mechanism to call the above methods, the following effect is produced (for each call):

1. the previously existing local bindings between the stub and the primitive binding is broken;
2. the new filter object is created;
3. new local bindings are established to connect the interfaces of the filter to the interfaces of the stub and the primitive binding.

Note that the meta-object does the remote operations by calling special supporting services on the interface of the capsule manager at each side of the binding (i.e. operations `create_component` and `local_bind`).

Environment Meta-model. As another example of adaptation, consider a binding object used for the transfer of audio between two stream interfaces. In order to provide a better control of the jitter in the flow of audio data, the interface of the binding connected to the sink object can be reified according to the environment meta-model. The environment meta-object can then be used to introduce a *before* computation that implements a queue and a dispatching mechanism in order to buffer audio frames and deliver them at the appropriate time, respecting the jitter requirement.

Resources Meta-model. As a further example, we show how resource management adaptations can be achieved in our architecture by using the resources MOP (with the assistance of the encapsulation MOP). For this example, consider an application that transmits audio over the network, from an audio source to an audio sink object. For this purpose, we set up a stream binding as depicted in figure 4. Task 1 regards the activity of transmitting audio over the network. This (distributed) task is subdivided into two lower level tasks, task 2 and task3, for sending and receiving audio respectively.

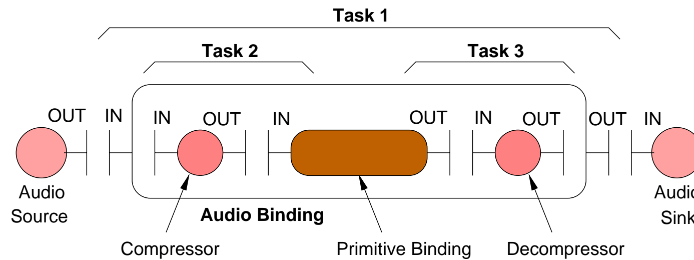

Fig. 4. A stream binding and its associated tasks

As an example of fine-grained resource reconfiguration consider that the audio binding is initially using a rate monotonic scheduling strategy. However, over time some changes are introduced to the system, e.g. another similar audio binding is introduced. In this case, there might be a need to change the scheduling policy for a dynamic scheduling policy such as EDF. Consider that the new binding has been established from another site than the initial local site. As soon as the QoS mechanism detects a lack of processing resources in the remote site, the QoS manager performs the following operations, considering that the task dictionary `vtmDict` was obtained from the meta-space:

```
VTM3 = vtmDict.getVtm('receive')
```

```
team = VTM3.getLLRs()['TEAM']
thread = team.getLLRs()['THREADS'][0]
thread.setManager(EDFscheduler, schedParam)
```

The first line reifies the encapsulation meta-object concerning the interface OUT of the audioBinding object and then a VTM (Virtual Task Machine) dictionary of such interface is obtained. This dictionary maps the interface methods onto VTMs. The VTM that supports the execution of the operation *receive* is then retrieved. The lower-level resources of the VTM are then inspected through the `getLLRs()` operation of the resources MOP (this is an example of navigating the resource management hierarchy using the *get* operations described in table 5). As a result, the team of threads supporting the receive operation is obtained. Similarly, the team resource object is inspected and a particular thread is obtained. Finally, the scheduling policy of this processing resource is changed to an EDF policy by calling the `setManager()` operation of the resources MOP. Note that this is equivalent to expelling the thread from the current scheduler and subsequently admitting it to the EDFscheduler.

As an example of coarse-grained adaptation consider the same audio application. In case of a drastic shortage of resources due to dynamic changes to the environment of the application, one of the existing stream bindings may be suspended on behalf of the other. The binding associated with the VTM with lowest priority would be suspended. The QoS control mechanism would proceed as follows, after finding out which is the lowest priority distributed VTM, which happens to be VTM_x:

```
vtmSched = VTM_x.getManager()
vtmSched.suspend(VTM_x)
```

As a result, the VTM scheduler will suspend the corresponding VTM. The operation of suspending this VTM encompasses the suspension of the local and remote VTMs concerning the transmission and reception of audio streams respectively. This in turn involves the suspension of their corresponding underlying processing resources, e.g. threads.

8 Related Work

There has been a growing interest in using reflection to achieve adaptation at the middleware level. A pioneering piece of work in this area was the CodA meta-level architecture[18]. CodA decomposes the meta-level in to a set of fine-grained components that reify aspects such as message sending, acceptance and execution. However, it is difficult to support unanticipated changes in the meta-level due to the tight coupling of components.

In the FlexiNet [13] project, researchers at APM have experimented with a reflective middleware architecture that provides a means to dynamically modify the underlying protocol stack of the communication system. However, their solution is language specific as applications must be written in Java.

Researchers at Illinois [23] have developed a reflective ORB that provides facilities to dynamically modify the invocation, marshalling and dispatching of the ORB. However, the level of reflection is coarse-grained since only these elements of the ORB are reified. Further research at this institution has lead to the development of dynamicTAO [22], which is a CORBA-compliant reflective ORB supporting run-time distributed reconfiguration. One difference from our work is that this system do not make use of explicit meta-objects for reifying the ORB. Instead they use a collection of entities known as component configurators. These components are in charge of both maintaining dependencies among system components and providing hooks for specific strategy implementations of the ORB (e.g. marshalling, scheduling, concurrency, etc). However, the focus is still very much on large-grained platform-wide configuration.

The work done in OpenCorba [15] follows a meta-class approach. In this system meta-classes allow reflective computation in order to control the behaviour and structure of all instances of a base-level class. In contrast, our work follows a meta-object principle, which allows a finer-grained (per-object) means to achieve adaptation. That is, changes to a meta-object are only spread to its corresponding base-level object.

Finally, note that OMG has standardised MOF (Meta Object Facility)[12] as a way of representing and managing meta information and XMI (XML Metadata Interchange) [10] as way for interchanging it. This effort is essentially different from our approach in the sense that it involves managing static meta-level descriptions of type systems rather than dynamic adaptation. However, the MOF can be used to represent the types of the meta-information that is used by meta-objects to perform reflection and dynamic adaptation. We are currently investigating this use of the MOF by implementing a second version of the reflective platform, which integrates the MOF concepts into our multi-model reflection framework.

9 Final Considerations

In this paper we have presented an approach for the open engineering of middleware. This leads to middleware that is flexible, adaptable and extensible and, in consequence, suitable to support applications with dynamic requirements.

Our approach applies reflection as a principled way of opening up the middleware implementation. The main points of our approach are: (1) *multiple meta-models*, which give us a means to separate different aspects of middleware implementation; (2) the use of *object graphs* as a uniform way to reify the platform configuration; and, (3) the use of *independent meta-spaces* for individual components of the platform which limits the scope of adaptation and provides a means for fine grained adaptation.

We have implemented a prototype in Python to demonstrate the main ideas. Notably, this implementation provides an initial meta-object protocol, which illustrates how reflective computation can be performed using the four meta-models. Although the prototype itself was limited to a single language envi-

ronment, it has shown the feasibility of the general approach. Ongoing work is looking at lightweight component architectures as an efficient and language independent way to structure the implementation.

Our experience so far has shown that the approach is generic and powerful enough to cover a wide range of aspects of middleware. Moreover, the use of a uniform interface for adaptation contributes to reducing the complexity of developing dynamic and complex applications. On the other hand, the intrinsic openness of the approach might compromise the integrity of the platform (by unexpected or undesired adaptations). This was expected since we focused on the provision of flexibility. Further work will look at ways of alleviating this problem. For instance, we can benefit from experiences such as those presented in [3] and [16], which use software architecture principles to represent the static and dynamic properties of configurations. This would allow us to extend our compositional meta-model in order to use the rules of the architectural style for constraining and managing configuration adaptations.

Finally, several case studies are under development in order to evaluate the architecture in more realistic application scenarios.

Acknowledgments

Fábio Costa would like to thank his sponsors CNPq, the Brazilian National Council for Scientific and Technological Development and the Federal University of Goiás (UFG), Brazil. Katia Barbosa Saikoski would like to thank her sponsors, Federal Agency for Post-Graduate Education (CAPES), Brazil and Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil. Hector A. Duran would like to thank his sponsor, the Autonomous National University of Mexico (UNAM).

References

1. G. Blair, F. Costa, G. Coulson, F. Delpiano, H. Duran, B. Dumant, F. Horn, N. Parlavantzas, and J-B. Stefani. The Design of a Resource-Aware Reflective Middleware Architecture. In *Second International Conference on Reflection and Meta-level architectures (Reflection'99)*, St. Malo, France, July 1999.
2. G. Blair and J-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.
3. G. S. Blair, L. Blair, V. Issarny, and A. Zarras P. Tuma. The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware2000)*, IBM Palisades Executive Conference Center, Hudson River Valley, New York, USA, 4th - 8th April 2000.
4. Microsoft Corporation. Microsoft COM Technologies - DCOM. Internet Publication - http://www.microsoft.com/com/tech/dcom.asp, March 1998.
5. Microsoft Corporation. Microsoft COM Technologies - COM+. Internet Publication - http://www.microsoft.com/com/tech/complus.asp, May 1999.
6. H. Duran and G. Blair. A Resource Management Framework for Adaptive Middleware. In *3th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC'2K)*, Newport Beach, California, USA, March 2000.

7. D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *15th ACM Symposium on Operating System Principles*, pages 251–266, December 1995.
8. S. Floyd, V. Jacobson, C-G. Liu, S. McCanne, and L. Zhang. A Reliable Multicast Framework for Light-weight Session and Application Level Framing. *IEEE/ACM Transactions on Networking*, December 1997.
9. Object Management Group. CORBA Object Request Broker Architecture and Specification - revision 2.2, February 1998.
10. Object Management Group. XML Metadata Information (XMI), October 1998.
11. Object Management Group. CORBA Components Final Submission. OMG Document orbos/99-02-05, February 1999.
12. Object Management Group. Meta Object Facility (MOF) Specification, Version 1.3 RTF, September 1999.
13. R. Hayton. FlexiNet Open ORB Framework. Technical Report 2047.01.00, APM Ltd., Poseidon House, Castle Park, Cambridge, CB3 ORD, UK, October 1997.
14. A. Hokimoto, K. Kurihara, and T. Nakajima. An Approach for Constructing Mobile Applications Using Service Proxies. In *ICDCS'96; Proceedings of the 16th International Conference on Distributed Computing Systems; May 27-30, 1996, Hong Kong*, pages 726–733, Washington - Brussels - Tokyo, May 1996. IEEE.
15. T. Ledoux. OpenCorba: a Reflective Open Broker. In *2nd International Conference on Reflection and Meta-level Architectures*, pages 197–214, St. Malo, France, July 1999.
16. O. Loques, A. Sztajnberg, J. Leite, and M. Lobosco. On the Integration of Configuration and Meta-Level Programming Approaches. In R. J. Stroud W. Cazzola and F. Tisato, editors, *Object-Oriented Reflection and Software Engineering*, volume 1826 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
17. P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of OOPSLA'87*, pages 147–155. ACM, October 1987.
18. J. McAffer. Meta-level Programming with CodA. In *Proceedings of 9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 190–214. Springer-Verlag, 1995.
19. Sun Microsystems. Enterprise JavaBeans Specification Version 1.1. Internet Publication - http://java.sun.com/products/ejb/index.html.
20. SUN Microsystems. Java Remote Method Invocation - Distributed Computing for Java. Internet Publication - http://www.sun.com, 1998. White Paper.
21. H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In *Proceedings of International Workshop on New Models for Software Architecture (IMSA'92)*, Tokyo, Japan, November 1992.
22. M. Román, F. Kon, and R. Campbell. Design and Implementation of Runtime Reflection in Communication Middleware: the dynamicTAO Case. In *ICDCS'99 Workshop on Middleware*, Austin, Texas, May 31 - June 5 1999.
23. A. Singhai, A. Sane, and R. Campbell. Reflective ORBs: Supporting Robust, Time-critical Distribution. In *Proceedings of ECOOP'97 - Workshop on reflective Real-Time Object-Oriented Programming and System*, Finland, June 1997.
24. T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OPSLA'88*, pages 306–315. ACM, September 1988.