Meta-ORB: A Highly Configurable and Adaptable Reflective Middleware Platform

Fábio Moreira Costa
fmc@inf.ufg.br
Instituto de Informática
Universidade Federal de Goiás
Bloco IMF-I, Campus II, UFG
74001-970 – Goiânia – GO – Brasil

Abstract

Middleware platforms have filled up an important gap in distributed application development, as they enable a high level of abstraction, masking the complexities of distributed systems programming. Recently, though, technology developments in areas such as multimedia systems, networking and mobile computing have made feasible new categories of applications that are not properly supported by conventional middleware. In common, such applications pose important requirements regarding platform configurability and adaptability, for which the *black box* nature of conventional middleware is not suitable. In this context, reflection and meta-level architectures represent a concrete solution, offering a principled way to *open up* the design and implementation of a platform, enabling its internal structure and behaviour to be dynamically inspected and adapted. In this way, new application requirements and environmental changes can be accommodated through platform configuration and evolution. This paper presents an approach for reflective middleware which combines meta-information management with meta-object protocols, offering a common basis for platform configurability and adaptability. The paper also describes a concrete implementation of the approach.

Keywords: Middleware, Reflection, Meta-Information Management, Reflective Middleware.

1 Introduction

The advent and popularisation of middleware platforms have greatly contributed to the widespread development and use of distributed systems and applications. Technologies such as CORBA [1], Java RMI [2], DCOM [3] and, more recently, J2EE [4] and .NET [5] have gained considerable space as part of the software development scenario. Such technologies offer a software layer that sits above operating systems and provides standard interfaces and services on top of which applications are developed. This enables a distributed environment where the problems originated from distribution are hidden away from the programmer. Typical non-functional concerns, such as location of services and resources, language and operating system interoperability, and replication, are made *transparent*, leaving the programmer free to deal with the functional concerns of the applications.

Middleware technologies have thus enabled the notion of open systems to be extended to the realm of distributed systems. Applications built using middleware can typically rely on the platform's standard interfaces to provide the common distributed systems services in a way that is both interoperable and portable across different operating systems and languages. Nevertheless, this conventional approach to openness is only partial, as current practice in

middleware standards requires only the interfaces to be open, while the details of platform implementation are left for the vendors to decide. Usually, this leads to rigid, monolithic platforms, which cannot be tailored to the particular needs of different applications. Firstly, not all applications use all services built into a platform, resulting in memory footprints larger than needed. In addition, some applications may need specialised services which are not provided by mainstream middleware (such as in the case of multimedia applications). Secondly, application requirements may evolve during runtime, demanding adaptations on the platform implementation, so that the set of services provided remain appropriate after, for instance, changes in the execution environment. This leads us to conclude that middleware platforms should be built around more flexible architectures, following an *open implementation* approach [6]. In this way, programmers can configure customised versions of a platform, as well as change their internals at runtime as need arises.

Reflection and meta-level architectures represent a principled way to open up the internal implementation of a platform for dynamic inspection and modification. As a complement to this, meta-information management offers a basis for middleware configuration, allowing one to specify and build customised instances of a platform. This paper presents an architecture for reflective middleware, called Meta-ORB, which exploits the combined use of such techniques in order to provide a flexible middleware architecture to support distributed multimedia applications. The paper is structured as follows. Section 2 reviews the basic concepts of reflection and meta-level architectures, while section 3 discusses the principles of meta-information management. Section 4 considers the application of reflection in the area of middleware, with an emphasis on requirements for reflective middleware. Section 5 then presents the specific approach adopted in the research and describes the Meta-ORB reflective middleware architecture, followed by section 6, which discusses the implementation of the platform. Finally, section 7 reviews important related work, while section 8 presents relevant conclusions and further considerations.

2 Reflection and meta-level architectures

The fundamentals of reflective computing systems were introduced by B. C. Smith and can be summarised by his *reflection hypothesis* [7], which argues that a system can be made to manipulate representations of itself in the same way as it manipulates representations of its application domain. Such a system is said to have a *self-representation*, which can encompass the several aspects of its implementation. In addition, if there is a relationship of *causal connection* [8] between the self-representation and the actual system, meaning that changes in one have corresponding effects in the other, the system is said to be *reflective*. The self-representation can thus be used for *inspection* and *adaptation* of the system's internals.

The architecture of a reflective system, also known as a meta-level architecture, is usually structured in levels, where the bottom level, known as *base-level*, deals with computation about the domain of application, whereas the levels above it, known as *meta-levels*, perform computation about the system itself. More precisely, each meta-level is concerned with the representation and manipulation of the level below it (which is its relative base-level), giving rise to the notion of a reflective tower of meta-levels, as illustrated in Figure 1. As shown in the figure, the act of a meta-level exposing the internals of its (relative) base-level is known as *reification*. This corresponds to the establishment of an explicit representation of the base-level system and its internal implementation, in terms of programming entities that can then be manipulated at runtime. Modifications to this self-representation result in corresponding

changes to the reified elements of the base-level, a process known as *reflection* or *absorption*. Given a particular base-level entity, the set of meta-level entities reifying it is know as the entity's *meta-space*.

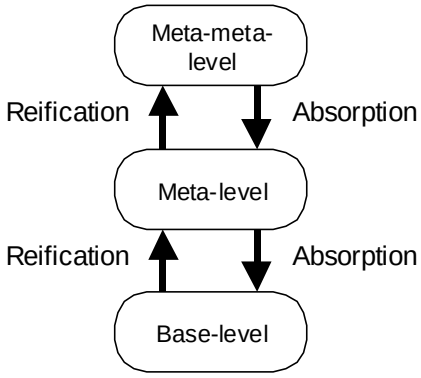

Figure 1 - Reification, reflection in a meta-level architecture.

The *object-oriented* paradigm provides a clean way to structure the meta-level. In general, object-orientation allows the distribution of the reflection mechanisms and interfaces among multiple, distinct meta-level entities [9]. Regarding terminology, in object-oriented reflection, the entities that populate the meta-level are called *meta-objects*, while those entities at the base-level are known as *base-level objects*. While the interfaces of base-level objects provide an object protocol for access to application functionality, the interfaces of meta-objects provide a *meta-object protocol* (MOP) [10], which allows reflective access to the implementation of the system. Importantly, the same object model should be employed at both base- and meta-level, meaning that reflection can be re-applied at the meta-level itself.

Finally, the design of reflective systems usually follows a distinction between *structural* and *behavioural* reflection, initially conceived in the context of programming languages [11]. Structural reflection is defined as the ability of a language (or platform) to provide a complete reification of the program currently executing, together with the abstract data types that are part of the program. On the other hand, behavioural reflection is the ability of a language (or platform) to provide a complete representation of its own semantics, in terms of the internal aspects of its runtime environment. Hence, while structural reflection usually deals with the functional properties of a system, behavioural reflection is typically concerned with the non-functional properties.

3 Meta-information management

3.1 Basic concepts

Reflective techniques inherently deal with meta-information in order to build the self-representation of base-level entities. Meta-information is kept about the reified aspects of a system, in explicit or implicit form, as part of the state of the meta-objects. Reflection, however, does not imply a consistent framework for modelling and maintaining meta-information, especially considering issues of sharing and distribution. The provision of such a framework is precisely the goal of meta-information management, and its presence is an important, often overlooked requirement for reflective middleware.

For the purposes of this paper meta-information can be defined as information about the system itself, instead of about the application domain of the system. The structured use of

meta-information is typically based on the concepts of *model* and *meta-model*. Models represent meta-information about the runtime entities that compose a given system, and may provide enough detail to enable instantiation of the system, as well as introspection on its internals. On the other hand, meta-models comprise higher-level meta-information, targetted at the representation of models. A meta-model thus describes the constructs that are available for modelling the entities of a system or application [12]. This paper is mainly concerned with the management of meta-information at the level of models.

In addition, besides the use of models and meta-models, an effective architecture for the management of meta-information must also provide facilities to assist with [13]:

- *meta-information definition*, such as with a language with well-defined syntax and semantics (conforming to the meta-model), as well as tools, such as compilers to validate and translate textual meta-information into a machine-readable form; alternatively, interactive tools (such as with a GUI) can be used for this purpose;
- *meta-information maintenance*, with a distributed and persistent *repository* with features for creating, deleting, managing and manipulating meta-information;
- *definition, storage and evaluation of relationships*, such as compatibility and substitutability, between different entities of meta-information; and
- *meta-information interchange*, based on mappings and tools to transfer meta-information between different repositories, possibly using different meta-models.

A well-known example of a general-purpose meta-information management architecture is the OMG Meta-Object Facility (MOF) [14], which provides a framework for defining and managing models and meta-models, along with the meta-information they comprise. Another example, although restricted to the CORBA meta-model, is the Interface Repository defined as part of the CORBA specification [1].

3.2 Meta-information management for middleware

The demand for a principled approach to meta-information in middleware comes from two basic needs, namely *type management* and *configuration management*. The former refers to the management of type-related meta-information describing the externally visible features of runtime entities, as well as relationships between them. This is especially useful in the open services environment supported by middleware, where new services can be dynamically introduced or evolved, and where service users dynamically bind to service providers. In this context, the availability of runtime meta-information describing the types of servers and clients is vital for the dynamic discovery of services, as well as for type checking and bridging of service types before binding [15].

Configuration management, in turn, refers to the activities of building a system from smaller parts in a structured way. This involves the creation, allocation and binding of primitive components in order to form more complex, composite components [16]. Explicit meta-information can be used to describe the internal configuration of the components of a system, in terms of *templates* with enough detail to allow their instantiation. Such templates also serve as runtime documentation of the configuration of a system and its components, thus providing a basis for reconfiguration. Using meta-information management techniques, templates can be defined and managed in terms of a meta-model. This enables the association between templates and typing meta-information, which in turn permits the use of type relationships to search and compare configurations, as well as to validate interconnections between the elements of a configuration.

It is therefore important to recognise the role of meta-information management as a principled basis for the definition, instantiation and management of customised middleware platforms. A promising scenario for the future will be the widespread existence of libraries of template and type meta-information describing alternative implemenations for the several aspects of middleware, which can be selected and combined (or even extended) in order to produce platforms that are tailored to particular requirments. It is important, however, that a uniform meta-information management architecture (such as the MOF) is used, so that types and templates can be consistently defined and unambiguously interpreted in the kind of heterogeneous environment typically supported by middleware.

4 The reflective middleware approach

4.1 Motivation

The overall rationale for reflective middleware comes from the need to *open up* the platform implementation, in order to allow the customisation and runtime adaptation required by dynamic applications. Existing middleware technologies have recognised the need to address this problem, although tackling it with a different approach, by adding flexible features on top of their core architectures (an example is the portable interceptor feature recently added to CORBA [17]). Despite the usefulness of such features, the degree of support for customisation and adaptation is only partial, not covering all aspects of the design and the different phases of a platform's lifecycle. This is mostly due to the inherent "black-box" nature of these technologies, which limits the extent to which elements of the design can be opened and exposed to the programmer. Reflection and meta-level architectures, on the other hand, offer a truly generic solution to the above problem, enabling a principled approach to the design of middleware in a way that naturally renders itself to openness [18]. In addition, the use of reflection enables the different aspects of a platform to be manipulated and adapted in ways that were not anticipated during its design.

In general terms, *reflective middleware* refers to the use of a causally connected self-representation to support the inspection and adaptation of the middleware system [19]. The same reflection techniques used in traditional areas (such as programming languages) apply to middleware as well [20]. Besides providing the usual middleware services through standard interfaces (e.g., as defined in the CORBA specification), a reflective middleware platform also provides meta-interfaces that allow the programmer to inspect and manipulate the internals of the platform.

4.2 Overall design issues for reflective middleware

The design of reflective middleware must usually observe the same requirements that are typical of other kinds of reflective systems (such as safety of reflection, the performance impact, and a uniform reflection model). Some requirements, however, are of particular importance in the context of middleware, and form the basis for the approach presented in this paper (see [21] for a more complete list of requirements):

1. *Modular platform infrastructure*. This is essential in order to facilitate a clear identification, at runtime, of the entities that implement specific features of the platform. It also enables such entities to be manipulated (via reflection) independently of each other.
2. *Language and system independence*. In order to enable portability of applications that use reflection functionality, it is important that a MOP for reflective middleware is defined at

a language-neutral level. Thus, the use of reflection capabilities of specific languages or operating systems should be discouraged.

3. *Pervasiveness of reflection*. This issue refers to the range of aspects of a middleware platform (e.g. marshalling, synchronisation, communications protocols, and the several distributed systems services) that can be reified with the reflection mechanisms. Ideally, all aspects of interest in a given context should be amenable to manipulation via reflection.
4. *Unified approach to configurability / re-configurability*. Essentially, facilities for static configuration (which enable the initial selection of the middleware components required in a particular scenario) should be integrated with the mechanisms used for dynamic reconfiguration. In particular, both should be based on a unified terminology, also using a consistent set of meta-level constructs.
5. *Management of meta-level complexity*. Meta-levels for middleware platforms tend to be highly complex, due to the large number of aspects that need to be represented and the interactions among them. In addition, the need to handle such aspects in a distributed environment only adds to the complexity. This requires appropriate meta-level abstractions, in order to make reflective programming a more manageable task.

5 The Meta-ORB architecture

The architecture of the Meta-ORB platform is based on the combination of meta-information management and reflection techniques. While the former is the basis for platform configurability, the latter provides the infrastructure and interfaces for dynamic adaptability. In this section, each of these aspects of the architecture will be examined in separate, describing the specific approaches adopted in the work.

5.1 Platform configurability

5.1.1 The Meta-ORB meta-model

The design of Meta-ORB is centred around an explicit meta-model, described using the MOF. This meta-model defines the kinds of entities used in the construction of instances of the platform, as well as the meta-information structures used to describe such entities. The use of such meta-information to specify and instantiate customised platform configurations qualifies Meta-ORB as a highly configurable framework for middleware.

The Meta-ORB meta-model is an extension of the CORBA meta-model, in a way that allows backward compatibility with the standard. The exensions have been influenced by the computational model for open distributed systems described in [22]. The first-class constructs of the meta-model are described below (and exemplified in section 5.1.2).

- *Interfaces*, which represent access points to the services provided by a component. Three styles of interfaces are provided: operational (for client/server-like interaction), stream (for continuous media interaction) and signal (for more primitive one-way interaction).
- *Component objects* (or, simply, *components*), which represent the units of functionality in the platform. Components can have multiple interfaces and can be defined in a hierarchical way by composing finer-grained components.
- *Binding objects*, which are the equivalent of distributed components, whose internal components can be dispersed across the network. Binding objects are the central concept of the meta-model and provide structured and configurable support to the interaction among remotely located application (and middleware) components.

The corresponding meta-model elements (meta-types) represent both the *type* and *template* aspects of such constructs, meaning that the meta-model provides a basis for the functions of type and configuration management. (In this paper, instances of such meta-types will simply be referred to as *types*).

The meta-model also includes elements to define *auxiliary types*, which do not correspond to first-class entities in the platform, but are essential to their description. Examples include: media types, constructed types and primitive types. In addition, the meta-model includes non-type-related meta-model elements. These elements correspond to the scope-defining constructs of the type system (e.g., module) and to auxiliary constructs, used in the definition of the first-class meta-types (e.g. operation, flow, signal and QoS annotation). A complete description of the Meta-ORB meta-model can be found in [21].

5.1.2 Defining platform configurations

A complete definition of the meta-model, describing all the features available for platform configuration, is out of scope in this paper. However, this section presents a few representative examples that should provide an idea of how the basic meta-model constructs can be used for building customised platform instances. A textual notation is used, based on the Meta-ORB ODL (Object Definition Language) [21]. Typically, the platform designer provides a set of specifications in ODL that define a particular middleware configuration. These definitions are stored as meta-information objects in a repository, from where they can later be retrieved and used to instantiate the whole or parts of the middleware configuration. Additionally, meta-information stored in the repository can be re-used as part of newly defined configurations.

```
module Example {
   primitive component AudioDeviceComp {
      implementatin: AudioDeviceImpl;
      interfaces: AudioDevice audio_interf;
   };
   primitive component VideoDeviceComp {
      implementation: VideoDeviceImpl;
      interfaces: VideoDevice video_interf;
   };
   interface <stream> AVDevice : AudioDevice, VideoDevice {};
   primitive component MixerComp {
      implementation: MixerCompImpl;
      interfaces: AudioDevice audio_interf;
                  VideoDevice video_interf;
                  AVDevice av_interf;
   };
   component AVDeviceComp {
      internal components: AudioDeviceComp audio_comp;
                           VideoDeviceComp video_comp;
                           MixerComp mixer_comp;
      object graph: (audio_comp, audio_interf):(mixer_comp, audio_interf);
                    (video_comp, video_interf):(mixer_comp, video_interf);
      interfaces: AVDevice av is (mixer_comp, av_interf);
   };
};
```

Figure 2 - An example component configuration specification.

In the first example, as shown in Figure 2, ODL definitions for a composite component are presented. Note that auxiliary definitions have been omitted for lack of space (notably those for interfaces, which are based on a multimedia extension to OMG IDL). The last definition specifies a component for audio/ video processing (`AVDeviceComp`), which is composed of three primitive components, also defined in the example. The configuration of the composite

component is specified in terms of its set of internal components, the object graph representing the way such internal components are connected (adjacent components are linked by means of their interfaces), and the interfaces that the overall component presents to its users. This example illustrates how arbitrarily complex units of functionality can be modelled and configured in terms of structured component composition, using primitive components (which encapsulate binary implementations) and composite components.

The next example similarly shows how distributed configurations can be specified using the binding construct. Figure 3 shows the specification of a complex binding object, aimed at connecting the interfaces of audio/video components of the kind defined above. The binding is built out of components and other binding objects (their definitions were omitted for brevity) that implement the different elements of middleware functionality, such as stubs, protocol filters and transport protocols. The binding definition is given in terms of the type of the binding control interface (which exposes functionality to control the operation of the binding, such as to pause and resume its operation), the type of the internal binding objects used in the configuration, and the roles implemented at each of the binding endpoints. In this particular case, a single role is defined as the binding is symetrical (i.e., both its endpoints are meant to connect interfaces of the same type). The definition of the binding role is similar to a composite component definition, except for the *cardinality* part, which specifies the maximum number of endpoints conforming to the role that can be created in a given binding instance (this means that multipoint bindings are supported). In addition, the definition of the configuration of a binding role (i.e., its object graph) must also specify the connection points with other binding roles, in order to connect the whole binding.

```
module Example {
   binding AVBinding {
      control interfaces: CtrlInterf ctrl is (CtrlComp, ctrl_interf);
      internal bindings: AudioBinding audio_binding;
                         VideoBinding video_binding;
      role AVBindingPartic {
         components: AVStubComp stub;
                     AudioFilterComp audio_filter;
                     VideoFilterComp video_filter;
         target interface: AVDevice is (stub, av_interf);
         cardinality: 2;
         configuration:
            (stub, audio_interf):(audio_filter, audio_interf);
            (stub, video_interf):(audio_filter, video_interf);
            (audio_filter, forward_interf):(audio_binding, audio_role);
            (video_filter, forward_interf):(video_binding, video_role);
      };
   };
};
```

Figure 3 - An example binding configuration specification.

The way such definitions are provided, stored and managed is an implementation-specific issue (see section 6). The essential requirement is that such definitions can be made public in some form (typically using a repository), so that they can be accessed by platform configuration tools, by designers looking for pre-defined configurations (for re-use as part of more complex configurations), and by the reflection mechanisms, as seen below.

5.2 Dynamic adaptation

5.2.1 Principles and structure of the meta-level

As seen above, the entities that constitute platform configurations have their structure fully described by meta-information elements. *Dynamic reconfiguration* thus requires some means to manipulate such meta-information at runtime, in a way that is causally connected with the respective instances of platform configuration. This is the role of the *reflective meta-level*, which completes the architecture of the Meta-ORB and contributes towards fulfilling the requirements identified in section 4.

Reflection in the Meta-ORB can be used for dynamic inspection and adaptation in the context of both platform and application elements. To this end, the overall architecture is conceptually divided into *base-level*, where the actual functionality of the platform and applications is defined, and *meta-level*, where the reflection capabilities are defined. The design of the meta-level follows the principles of the Open ORB reflective middleware architecture [23], as discussed below.

A fundamental principle of the meta-level is the use of object-oriented reflection, which means that the entities that populate the meta-level are uniquely identifiable objects. The definition of *object*, in the Meta-ORB, refers to the structural constructs defined in the meta-model, namely components, binding objects and interfaces, although, typically, only component objects (and their interfaces) are used in the constitution of the meta-level. Hence, the meta-object protocol (MOP) is realised in terms of the interfaces of components that play the role of meta-objects. In addition, object-oriented reflection also assumes that the base-level is similarly structured in terms of objects, meaning that meta-objects are used to reify components, binding objects and interfaces. Importantly, in the Meta-ORB approach the state of meta-objects must always have a direct correspondence with the meta-information elements that describe their respective base-level objects. In practice, such meta-information is used, during the reification process, as the basis for initialising the state of meta-objects.

The other important principle on which the design of the meta-level is based is known as *multi-model reflection framework*, first introduced by [24] in the AL-1/D language. This approach consists in applying *separation of concerns* to the design of the meta-level itself, so that the meta-space of an object is partitioned into a number of distinct aspects, each one realised by a different meta-object. This is important to reduce the complexity of the meta-level, especially considering the multitude of features that must be managed in a meta-level for middleware.

Each separate aspect of the meta-level is defined in terms of a *meta-space model*, which represents the structure and functionality for the reification of a base-level object according to that aspect. Figure 4 illustrates the concept of using distinct meta-objects (each one corresponding to a different meta-space model) to reify a given base-level object.

Currently, five meta-space models are specified, with well-defined abstract design and semantics. The meta-space models are categorised according to the usual distinction between *behavioural* and *structural* reflection, discussed in section 2. The behavioural part of the meta-space consists of two meta-space models: *Resources* and *Interception*. The former is responsible for reifying and managing the resources (such as storage and processing) used by base-level objects, while the latter deals with the manipulation of implicit behaviour associated with the interfaces of components (e.g., pre- and post-processing that affect the non-functional properties of the interactions on a given interface). Typical operations

provided by the behavioural meta-object protocols include the addition or removal of interceptors at an interface, and the re-definition of the properties of the resources allocated to an object (such as the amount of memory and scheduling parameters).

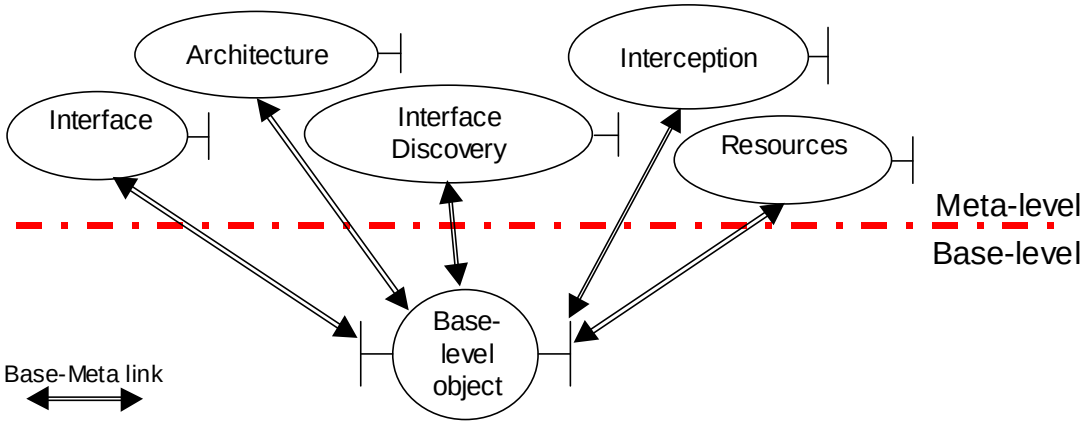

Figure 4 - Reifying a base-level object according to multiple meta-space models.

Structural reflection, on the other hand, is represented by three distinct meta-space models: *Interface Discovery* (which reifies the set of interfaces supported by a component or binding object), *Interface* (which reifies the constitution of a particular interface, in terms of the operations, flows or signals it provides), and *Architecture* (which reifies the internal configuration of a component or binding object, in terms of an object graph representing its internal components and the way they are connected).

5.2.2 Dynamic adaptation of platform structure

The current version of Meta-ORB is focused on structural reflection, based on the *Interface Discovery*, *Interface*, and *Architecture* meta-space models. However, only the latter is meant for adaptation, whereas the former two are meant for inspection only (i.e., to discover the services provided by a component, in terms of interfaces and their operations).

Adaptation according to the *Architecture* meta-space model is achieved through the manipulation of the object graph that represents the configuration of a given platform element. The meta-object protocol associated with this meta-space model offers operations for inspecting the structure of a configuration, as well as for changing it, by adding, removing or replacing components. For instance, in a binding configuration, such as the one described in Figure 3, if the available bandwidth of the underlying network suffers a drop, it may become impossible to sustain the previously agreed quality of service. Under the circunstances of rigid middleware infrastructures, such as with conventional middleware, this would typically cause the binding to be torn down. On the other hand, in the Meta-ORB reflective middleware, the use of the *Architecture* meta-object may help overcome the problem in a more satisfactory way. The solution could involve the selection of an alternative video encoding method with lower bandwidth requirements, as well as a component type (defined in the meta-information repository) that implements it. The *Architecture* meta-object can then be used to replace the current video filter components (at each of the binding endpoints) with components of the selected type, without disrupting the overall service (although the user might experience some downgrading of the video output quality). The code for implementing such reconfiguration, using the Python-based prototype (see section 6), is shown in Figure 5.

```
import MetaORB
# Obtain a reference to the Architecture meta-object.
arch_mobj = MetaORB.get_arch_mobj(bind_ctrl.get_binding_name())

# Obtain the type of the new component from the Type Repository.
new_video_fiter_type = MetaORB.TypeRep.lookup_name('LowBandwidthVFilter',
                                                   dk_Binding)

# Pause the binding, so that reconfiguration can be performed without
# breaking its consistence.
bind_ctrl.pause()

# Invoke the appropriate operation of the Architecture MOP to replace all
# occurrences of the video filter component (in all endpoints conforming
# to the AVBindingPartic role) with components instantiated from the new
# component type.
arch_mobj.role_replace_component(AVBindingPartic, video_filter,
                                 new_video_filter_type)
# Resume normal operation of the binding
bind_ctrl.resume()
```

Figure 5 - Reconfiguration example (in the Python prototype).

Other examples of use of dynamic adaptation can be found in the so called 24x7 applications (i.e., those applications that need continous, non-stop availability). In such cases, the use of the *Architecture* meta-object protocol represents a natural way to adapt the application or the underlying platform (e.g., when user requirements or business rules change) without having to stop, reconfigure and then re-start it.

The bottom line for using reflection in such a way is therefore the convenience of making *runtime* structural changes to an application or to the underlying platform. In addition to smoothing the change process (by preserving continuous availability of the adapted service), this approach also enables a simplification of the process of system evolution, as changes can be made in a localised way, without affecting the whole system. There are, however, problems that need to be solved in order to make the approach completely feasible. One problem is the transfer of state when replacing a stateful component. The current approach is to require stateful components to support a control interface with operations to handle the copy of state from the replaced component into the new one. Another related problem occurrs when the adaptation of a binding object involves the manipulation of components that lie on its data path. In such cases, performing the adaptation will cause the flow of data through the binding to be shortly interrupted, which may cause the loss or the inconsistency of information transferred. In the current design, the approach is twofold. First, a mechanism is provided that allows the meta-level programmer to pause the operation of the binding before starting the adaptations and to resume it once adaptations are completed (as seen in Figure 5). Second, the design encourages efficient implementations of the MOP, so that, for instance, adaptations can be made to fit within the interval between successive packet arrivals. Nevertheless more sophisticated solutions to the problem of smoth adaptation are still required, and are the subject of future work.

5.2.3 The impact on meta-information management

As seen above, an important requirement is that the self-representation maintained by a meta-object is always consistent with the type of its base-level object. However, as a result of

adaptation, the configuration of the base-level object (and thus its self-representation) becomes different from that specified in the type. To solve this apparent contradiction, Meta-ORB adopts an approach based on *type evolution* [25], which means that the type of an object is changed (into a version of the original type) when the object is subject to adaptation. However, the new type is only published in the repository when the base-level object becomes stable (i.e., no further adaptations are envisaged) and the meta-object is explicitly asked to do so (until then, a private copy of the type is kept in the meta-object). As an interesting consequence, the approach enables new component and binding types to be derived as a result of reflective adaptations. Such new types (once published) can be used to create objects that contain, from scratch, the results of previous adaptation efforts.

6 Implementation

A prototype implementation of the Meta-ORB architecture has been developed with the goal of demonstrating its feasibility and applicability [26]. The focus of this work was on the functionality and the qualities of the architecture, rather than performance. This is reflected on the chosen implementation environment, based on the Python programming language [27], which favours rapid prototyping instead. Despite this, experiments have shown that the performance of the prototype is appropriate for simple multimedia applications [21]. In addition, by implementing the prototype purely in Python, portability to a variety of operating systems is guaranteed, which was also a factor when choosing the language. The implementation is structured in three main modules, according to the abstract design discussed in section 5. These modules are briefly described below.

6.1 Platform Core

This module implements the core features that are necessary to support the Meta-ORB programming model. Specifically, it contains the basic distribution infrastructure, with naming and capsule management services, as well as the primitive constructs to support the meta-model, such as interface references and local bindings (which are links between the interfaces of locally connected componentes). In addition, this module defines the runtime representation for the first-class constructs of the programming model: interfaces, components and binding objects. In particular, regarding the latter, the implementation encourages the use of the General Inter-ORB Protocol (GIOP) as the basis for communication between the components of binding objects. This is on the way of providing interoperability with CORBA, though further work is still needed (e.g., to use interface references that are compatible with IOR standard). Finally, higher-level services are also defined in this module, notably component and binding factories, which are the entities responsible for the instantiation of components and binding objects based on specified type meta-information.

6.2 Type Repository

This module implements the meta-information management framework of Meta-ORB, providing support for both the platform core and its meta-level. Its logical structure is an extension of the CORBA Interface Repository, in order to comprise the new meta-types introduced by the Meta-ORB meta-model, in addition to those that are native of CORBA. The implementation is based on replication of the repository, in order to increase performance when accessing type definitions. Persistence of type definitions is achieved through their

simple serialisation and storage in the local file system of each repository replica (use of a database system is considered for future development). Creation of new type definitions, in turn, is performed through a master-slave colaboration between the repository replicas, where the master is the replica that receives and processes a given type creation request, propagating the new type definition to the slave replicas. Type versions are created in a similar way, though there is a centralised manager responsible for generating unique version numbers. The module also introduces tools to facilitate the definition and manipulation of meta-information, such as a GUI-based browser, used to specify, edit, publish and search for type definitions.

6.3 Meta-level

This module corresponds to the mechanisms and facilities for structural reflection provided by the platform. It follows the framework described in section 5.2, with the design defined in terms of the constructs of the programming model. Thus, meta-objects are themselves components, and are created and managed using the services provided by the Platform Core and Type Repository modules. The overall approach is to provide a default design and implementation, with meta-object types that offer a representative meta-object protocol. This design can then be extended with new meta-object types, either through static type definition, or through reflection (i.e., using meta-meta-objects) and type evolution. The precise meta-object protocols currently implemented are described in [21].

The implementation of the *Interface* and *Interface Discovery* meta-objects is straightforward, as they simply provide a convenient way to access type meta-information about the base-level objects. Their use is preferred instead of direct access to the respective types in the repository, as they should provide up-to-date type meta-information (considering any previous adaptations and evolution of the type).

Architecture meta-objects, on the other hand, have a more complex implementation, as they also provide for adaptation. This means that causal connection must be explicitly maintained, which is achieved by allowing meta-objects to directly manipulate the runtime representation of their respective base-level objects, so that they can perform the absorption of reflective computation (see Figure 1).

7 Related work

The work presented in this paper has been developed as part of the Open ORB project at Lancaster University. This project offers a generic architecture, from which the overall approach for reflective middleware used in Meta-ORB was derived. Different realisations of this architecture have been proposed, focusing on aspects such as efficiency, resource management and software architectures [23]. Meta-ORB, in turn, is another realisation of this architecture, focusing on complementing it with a comprehensive and consistent approach for managing the meta-information used by the reflection mechanisms.

Outside the scope of Open ORB, other projects have also adopted reflection as a principled way to build flexible middleware platforms, though following different approaches. OpenCORBA [28], for instance, is a reflective implementation of CORBA based on the meta-class approach and on the idea of modifying the behaviour of a middleware service by replacing the meta-class of the class defining that service. This is mainly used to dynamically adapt the behaviour of remote invocations, by applying the above idea to the classes of stubs and skeletons. The use of meta-classes, however, has the consequence of making such

adaptations reflect on all instances of a class. In contrast, in Meta-ORB reflection is based on per-object meta-objects, enabling to isolate the effects of reflection (so that other objects are not affected when reflection is used to alter a particular object). In reflective middleware, this is a desirable property as the components of a middleware system tend to be fairly independent of each other (even though they might have the same class).

DynamicTAO [29] is another representative reflective middleware architecture. It is based on an extension of the TAO ORB [30] with the concept of architectural awareness, making explicit the architectural structure of a system in a causally connected way. Middleware configurations are defined in terms of prerequisite specifications, which represent the components of the platform and the dependencies among them. These specifications are used by an automatic configuration service to instantiate the platform components and the components on which they depend. At runtime, such prerequisites are managed by component configurators, which are in charge of keeping the consistency of dependencies as new components are added or removed from the system. This approach is similar to the use of architectural reflection in Meta-ORB, with the added value of dependency management. However, dynamicTAO restricts the use of reflection to coarse-grained components, limiting its applicability to control more detailed structures of the platform.

Regarding the management of meta-information, although all reflective middleware architectures (such as the ones discussed above) deal with meta-information in one way or another, the treatment is typically *ad hoc*. On the other hand, the isolated use of meta-information management in middleware, notably for type management purposes has been proposed in the literature (such as in [31]). To our knowledge, however, Meta-ORB is the first middleware architecture to integrate a comprehensive and pervasive framework for meta-information management with a principled reflective meta-level. This has the benefit of unifying the use of meta-information in the system (e.g., preventing that different meta-object implementations use different meta-level representations), as well as providing a basis to closely integrate the configuration and adaptation features of the platform.

8 Conclusion and further considerations

This paper has presented Meta-ORB, a reflective middleware platform based on a combination of a meta-level architecture with a meta-information management framework. The overall aim of the research has been to propose an approach that permits the integration of configuration and reconfiguration facilities in a highly flexible middleware architecture. The foundation concepts used in the research have been surveyed, and motivation for their use in middleware has been considered. The paper also discussed the use and implementation of the major features of the platform.

The design of Meta-ORB was guided by the requirements presented in section 4, which are used here as a framework against which to evaluate the proposed approach. Firstly, the use of components as the basic building blocks of middleware emphasises modular platform architectures, which naturally eases the identification of the entities that need to be adapted. Second, the architecture of Meta-ORB has been defined in an abstract way, not tied to the specific facilities of any given programming language or system. This enables its implementation (and consequent interoperability) in different programming environments. Third, the fact that the reflection mechanisms are built to operate on the basic building blocks of the architecture means that any aspects of a middleware platform can be subject to reflection. Fourth, the integration of meta-information management and reflection provides a

natural basis to unify the facilities used for configuration and dynamic reconfiguration, contributing to preserve the consistency between the mechanisms used during design and run time. Last, but not least, the use of a multi-model reflection framework as the principle to structure the meta-level is an important contribution to lower the complexity of meta-level programming. In this way, reflection can be performed according to a particular meta-space model, largely ignoring the details of aspects managed by other meta-space models. Overall, the approach has been proven feasible for the support of distributed multimedia applications [21]. In particular, the dynamic adaptation features enable to the underlying platform structure to continuously match the requirements of the application as they evolve over time.

Nevertheless, enhancements to the Meta-ORB architecture are still possible, in order to fully explore the possibilities of the proposed approach. In particular, the author is currently investigating the feasibility of the approach in mobile and wireless environments. In this context, the configuration and adaptation features should decisively contribute to the definition of custom middleware platforms that suit, both statically and dynamically, the limited and variable resource availability that is typical of such environments. To make this feasible, however, future work is required in order to define more lightweight implementations of the reflection and meta-information mechanisms of the platform. Another area for future work consists in the implementation of the behavioural reflection part of the multi-model reflection framework. In particular, the *Resources* meta-space model is required in order to take full advantage of adaptability as a means to control quality of service. This, however, requires further investigation to extend the current approach in order to meaningfully represent and manage behavioural meta-information within the repository framework. There is also scope for research to further exploit the meta-information management approach. In particular, the use of (meta-)meta-information at the meta-model level (the current design only deals with meta-information at the level of models) offers a promising new approach to platform extensibility and interoperability. More specifically, the use of meta-meta-information to interpret the structure and semantics of model elements can help bridging platforms with different meta-models. In addition, by enabling the explicit manipulation of meta-models, it is possible to dynamically extend a platform with new kinds of constructs (such as to introduce new styles of communication). Finally, security and safety issues involving the use of reflection should be addressed, such as by enabling access control to the MOP and by preventing reflective adaptations that may result in inconsistent platform configurations. Such enhancements are envisaged for future versions of the platform.

Acknowledgements

This research was developed at Lancaster University, as part of of the author's Ph.D. Thesis, under the sponsorship of CNPq and UFG. The author would like to thank the members of the Open ORB team at Lancaster for their useful contributions, in particular: Gordon Blair, Geoff Coulson, Katia Saikoski, Rui Moreira, Nikos Parlavantzas, Hector Duran and Tom Fitzpatrick.

References

[1] OMG, *The Common Object Request Broker: Architecture and Specification - Revision 2.6,* Object Management Group. Needham, MA USA, 2001.

[2] Sun. Java Remote Method Invocation (RMI), Sun Microsystems, Inc., 2000. http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html.

[3] Microsoft. Microsoft COM Technologies - DCOM, Microsoft Corporation, 2000. http://www.microsoft.com/com/dcom.asp.
[4] Sun. Java 2 Platform Enterprise Edition - Docomentation Page, Sun Microsystems, Inc., 2001. http://java.sun.com/j2ee/docs.html.
[5] Microsoft. .NET Development, Microsoft Corporation, 2000. http://msdn.microsoft.com/net/.
[6] Kiczales, G., *Beyond the Black Box: Open Implementation*. IEEE Software, **13**(1): p. 8-11, 1996.
[7] Smith, B.C. Reflection and Semantics in a Procedural Language. Ph.D. Thesis, MIT Laboratory of Computer Science (MIT Technical Report 272), Massachusetts Institute of Technology, 1982.
[8] Maes, P. *Concepts and Experiments in Computational Reflection*. in *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*. Orlando, FL USA, 1987.
[9] Foote, B. *Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?* in *ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures*. Ottawa, Canada, 1990.
[10] Kiczales, G., J. des Rivieres, and D.G. Bobrow, *The Art of the Metaobject Protocol*. MIT Presss, 1991.
[11] Malenfant, J., M. Jacques, and F. Demers. *A Tutorial on Behavioural Reflection and its Implementation*. in *Proc. of Reflection'96*. San Francisco, CA USA, 1996.
[12] Mili, H., et al. *Metamodelling in OO - OOPSLA'95 Workshop Summary*. in *Addendum to the proceedings of the 10th annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*. Austin, TX USA, 1995.
[13] Crawley, S., et al. *Meta-Information Management*. in *Proc. 2nd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'97)*. Canterbury, United Kingdom, 1997.
[14] OMG, *Meta Object Facility (MOF) Specification, version 1.3 (OMG Document formal/2000-04-03)*, Object Management Group. Needham, MA USA, 2000.
[15] Kutvonen, L. *Management of application federations*. in *Proc. International IFIP Workshop on Distributed Applications and Interoperable Systems (DAIS'97)*. Cottbus , Germany, 1997.
[16] Crane, S., et al. *Configuration Management for Distributed Software Services*. in *Proc. IFIP/IEEE International Symposium on Integrated Network Management (ISINM'95)*. Santa Barbara, USA, 1995.
[17] OMG, *Interceptors Published Draft with CORBA 2.4+ Core Chapters (Portable Interceptors specification)*. OMG Document ptc/2001-03-04 ed, Object Management Group. Needham, MA USA, 2001.
[18] Blair, G.S. and M. Papathomas. *The Case for Reflective Middleware*. in *Proc. 3rd. Cabernet Plenary Workshop*. Rennes, France, 1997.
[19] Coulson, G. What is Reflective Middleware? Distributed Systems Online Journal, IEEE Computer Society, 2000. http://boole.computer.org/dsonline/middleware/RMarticle1.htm.
[20] Cazzola, W., S. Chiba, and T. Ledoux, *Reflection and Meta-Level Architectures: State of the Art, and Future Trends*, in *Object-Oriented Technology (ECOOP'2000 Workshop Reader)*, J. Malenfant, S. Moisan, and A. Moreira, Editors, p. 1-15. Springer, Heidelberg, Germany, 2000.
[21] Costa, F.M. Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware. Ph.D. Thesis, Computing Department, University of Lancaster, 2001. http://www.comp.lancs.ac.uk/computing/users/fmc/pubs/thesis.pdf.
[22] Blair, G.S. and J.-B. Stefani, *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.
[23] Blair, G.S., et al., *The Design and Implementation of Open ORB version 2*. IEEE Distributed Systems Online Journal, **2**(6), 2001.
[24] Okamura, H., Y. Ishikawa, and M. Tokoro. *AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework*. in *Proc. International Workshop on New Models for Software Architecture (IMSA'92)*. Tokyo, Japan, 1992.
[25] Costa, F.M. and G.S. Blair. *Integrating Reflection and Meta-Information Management in Middleware*. in *Proc. International Symposium on Distributed Objects and Applications (DOA'00)*. Antwerp, Belgium, 2000.
[26] Costa, F.M. Meta-ORB: Integrating Reflection and Meta-Information Management in Middleware DMRG, Lancaster University, 2001. http://www.comp.lancs.ac.uk/computing/users/fmc/M-ORB/.
[27] van Rossun, G. Python Documentation, version 2.2 Python Labs., 2001. http://www.python.org/doc/.
[28] Ledoux, T. *OpenCorba: a Reflective Open Broker*. in *Proc. 2nd International Conference on Reflection and Meta-level Architectures (Reflection'99)*. St. Malo, France, 1999.
[29] Kon, F., et al. *Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB*. in *Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*. New York, USA, 2000.
[30] Schmidt, D.C., D.L. Levine, and S. Mungee, *The design of the TAO real-time object request broker*. Computer Communications, **21**(4): p. 294-324, 1998.
[31] Brookes, W., et al., *Types and their management in open distributed systems*. Distributed Systems Engineering, **4**(1): p. 177-190, 1997.