# Structuring Reflective Middleware using Meta-Information Management: The Meta-ORB Approach and Prototypes

Fábio M. Costa and Bruno da Silva Santos
Instituto de Informática
Universidade Federal de Goiás
Cx. Postal 132, CEP 74001-970 - Goiânia GO - BRAZIL
{fmc | brunosilva }@inf.ufg.br

## Abstract

*Reflection is now an established technique for achieving dynamic adaptability of middleware platforms. It provides a clean and comprehensive way to access the internals of a platform implementation, allowing its customisation in order to achieve the best performance and adequacy under given operation environments and user requirements. In addition, the use of a runtime component model for the design of the internal platform structure facilitates the identification of the elements to be adapted, as all platform aspects are built in terms of components. The major limitation of this approach, however, is related to the multitude of aspects that make up a middleware platform, together with the requirement of keeping platform consistency after adaptations take place. This paper presents the results of ongoing research contributing to reduce this limitation. The approach is based on the use of a common meta-model, together with meta-information techniques to provide a uniform way to specify and manipulate platform configurations. Both platform configuration and runtime adaptation are always specified using a small number of building blocks defined in the meta-model. The paper also describes the overall architecture of the Meta-ORB platform, which demonstrates this approach, and presents its two implementations: a proof-of-concept prototype written in Python, and a Java-based implementation aimed at supporting mobile devices. The results are also evaluated from a quantitative perspective, according to the requirements of multimedia applications, one of the major areas of application of reflective middleware.*

**Keywords:** Reflective middleware, Meta-information management, Dynamic reconfiguration.

*This work was partially supported by FUNAPE/UFG and CNPq.

## 1 Introduction

The Meta-ORB platform is one of the instantiations of the Open-ORB reflective middleware architecture [1]. It is based on an novel approach that seamlessly integrates, through a common meta-modelling architecture, the reflective capabilities of the platform (used for dynamic adaptation) and its flexible configuration features [4, 3]. Static configuration is achieved through the use of a well defined component model, which identifies the main constructs available for building the platform. Using this model, customised instances of the platform can be achieved as configurations of interconnected components, each one fulfilling a particular functionality of the middleware. Such configurations are specified using the concepts and constructs defined by the meta-model, such as components, interfaces and bindings (explicit connections between components). These same concepts are equally used to build the end-user applications that run on top of the platform, thus achieving a uniform programming model that spans both the infrastructure and the application levels.

Dynamic reconfiguration, on its turn, is achieved through a reflective meta-level architecture, which provides a meta-object protocol (MOP) for inspection and adaptation of the structure and behaviour of the platform. In particular, the MOP allows the programmer to discover the interfaces of the components and make dynamic calls to the operations defined on those interfaces. Most importantly, however, is the MOP's ability to provide a causally-connected representation of the platform's configuration. This allows the meta-programmer to inspect the configuration (in terms of a graph of components) and make changes on it, such as by the addition, removal or replacement of components.

The meta-model of the platform underlies all the above functionality. It is present in the form of meta-information which is distilled from the definitions (using a component configuration language) of components, interfaces and bindings. Such meta-information is made available through a repository, which provides component and binding factories with the necessary information to create specific platform configurations. In addition, the repository also feeds the reflective meta-objects with the meta-information required to reify the internal configuration of the platform, in the form of a graph of components interconnected through their interfaces and bindings. In this way, both configuration and dynamic reconfiguration activities are based upon the same conceptual framework, freeing the developers from the burden of having to learn different terminology and concepts for each case.

This paper presents a detailed description of the Meta-ORB approach, focusing on the foundational concepts and their application in the context of a reflective middleware architecture. The paper also describes the two existing implementations of this architecture: a Python-based proof-of-concept prototype, and a newer implementation based on Java and aiming at mobile devices. The remaining of the paper is structured as follows. Section 2 introduces the basic concepts of reflection and meta-level architectures, as well as the concept of meta-information management and its application in middleware. Section 3 describes the base-level architecture of Meta-ORB, together with the major elements of its meta-model, while section 4 presents the meta-level architecture and the reflective facilities of the platform. Both sections present useful examples of platform configuration and dynamic reconfiguration in order to illustrate the use of such facilities. Section 5 then describes the architecture and implementation of the two existing prototypes, followed by section 6, which presents a performance evaluation of the Python-based prototype. Finally, section 7 discusses relevant related work, and section 8 presents some conclusions of this research.

## 2 Foundation

### 2.1 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 [31], 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 both its state and behaviour. In addition, if there is a relationship of causal connection [22] between the self-representation and the actual state and behaviour of the 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 is usually structured in levels, thus the term *meta-level architecture*. 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. In principle, as with recursive procedures, this tower can have an indefinite number of levels. In practice, however, the use of techniques such as the lazy creation of meta-levels (instantiating them on demand, upon a reification operation) means that typically only a few levels are actually present.

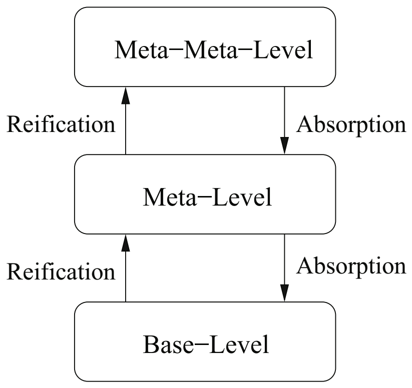

Figure 1: Overall architecture of a reflective meta-level system.

As shown in Figure 1, 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*.

#### 2.1.1 Behavioural and structural reflection

The design of reflective systems usually follows a distinction between structural reflection and behavioural reflection, initially conceived in the context of programming languages [14, 24]. Structural reflection is defined as the ability of a language 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 (also referred to as computational reflection [22]) is the ability of a language to provide a complete representation of its own semantics, in terms of the internal mechanisms of its runtime environment (such as method scheduling and dispatching). Note that these two styles of reflection are complementary to each other, with many reflective architectures providing both.

### 2.1.2 Object-oriented reflection

A well-defined meta-level structure is an important ingredient to facilitate the use of a reflective architecture, due to the multitude of aspects that may need to be handled. What is needed is a meta-level that allows each of the concepts of the system to be easily identified, in terms of discrete elements that can be handled separately from each other. The object-oriented paradigm provides a clean way to structure the meta-level. In general, object-orientation allows the partitioning of the reflection mechanisms and interfaces among multiple, distinct, meta-level entities [22]. 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. Thus, while the interfaces of base-level objects provide an object protocol for access to the system's externally visible functionality, the interfaces of meta-objects provide a *meta-object protocol* (MOP) [18], which allows reflective access to the internal 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.

## 2.2 Meta-information management

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 either 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, targeted at the representation of models. A meta-model thus describes the constructs that are available for modelling the entities of a system or application [9]. 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 [6]:

- 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) [27], 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 [28].

### 2.2.1 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 [20].

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 [10]. 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 would be the widespread existence of libraries of template and type meta-information describing alternative implementations for the several functional elements of middleware, which can then be selected and combined (or even extended) in order to produce platforms that are tailored to particular requirements. 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

## 3 Core meta-model

Configurations of the Meta-ORB platform are built in terms of a set of building blocks defined according to a well defined meta-model [3]. The major building blocks are components and binding objects. While the former are used for encapsulating local functionality of the platform (or applications), the latter are aimed at realising remote access between components in an explicit way. In addition, all interactions among components and bindings are made via well-defined interfaces. The meta-model definition was inspired by the ISO RM-ODP (Reference Model for Open Distributed Processing) standard [17], with the concept of object replaced with that of a component [35].

Meta-ORB components can be of two kinds: primitive or composite. Primitive components can be seen as encapsulation of implementation artefacts (such as language-level classes), giving them a higher level status, as an entity that can be manipulated and interacted with using the platform programming model. Composite components, on the other hand, are more elaborate entities, which are made up with other components, interconnected by their interfaces and forming a component graph, as shown in the examples below.

Following the same idea, binding objects are also classified as primitive and composite. A primitive binding represents an encapsulation of a transport protocol, in order to allow its use according to high-level interfaces that are more tailored to the types of the components connected through the binding. A composite binding, on its turn, is an encapsulation of more elementary binding objects, providing a higher level of abstraction and services on top of them. For instance, as shown in Figure 2, a composite binding for a video streaming application can be composed of a pair of video codecs (one at each side of the binding), which are connected through a primitive binding based on UDP. As the figure shows, another kind of component in a binding are stubs, which are responsible for the adaptation of the services provided by the binding object with respect to the external interfaces it must support. In addition, notice that a binding object also has a third interface, which is provided for the purpose of controlling its operation.

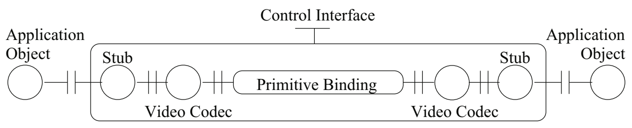

Figure 2: Example composite binding object.

Specific types of component and binding objects, as well as their interfaces, are defined using a special object definition language, called Meta-ORB ODL (Object Definition Language) [3]. This is an extension of the standard CORBA 2.2 IDL [26], with constructs for the definition of components and bindings as first-class entities. Component, interface and binding type definitions are then stored in the *type repository* [4], which provides runtime access to this meta-information for other parts of the platform. Notably, the instantiation of such objects is performed by component and binding factories, which obtain the appropriate definitions from the repository in order to create the proper configurations.

Finally, 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 is out of scope in this paper and can be found in [3].

## 3.1 Examples of platform configurations

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. 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.

In the first example, shown in Figure 3, ODL definitions for a composite component are presented. Note that auxiliary definitions have been omitted for brevity (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 4 shows the specification of a complex binding object, aimed at connecting the interfaces of audio/video components of the kind defined above (the structure of the resulting binding is shown in Figure 5). 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 symmetrical (i.e., both its endpoints are meant to connect interfaces of the same type and with the same semantics). 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 multi-point bindings are supported). In addition, the definition of a binding role configuration (i.e., its object graph) must also specify the connection points between the binding's components and the appropriate roles of its internal bindings.

```
module Example{
  primitive component AudioDevComp{
    implementation: AudioDevImpl;
    interfaces: AudioDev audio_interf;
  };
  primitive component VideoDevComp{
    implementation: VideoDevImpl;
    interfaces: VideoDev video_interf;
  };
  interface <stream> AVDev: AudioDev, VideoDev{};
  primitive component MixerComp{
    implementation: MixerCompImpl;
    interfaces: AudioDev audio_interf;
                VideoDev video_interf;
                AVDev av_interf;
  };
  component AVDevComp{
    internal components: AudioDevComp audio_comp;
                         VideoDevComp 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 3: An example specification of a composite component.

## 4 Reflective meta-level

As seen above, the entities that constitute platform configurations have their structure fully described by meta-

```
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):
                    (video_filter, video_interf);
        (audio_filter, forward_interf):
              (audio_binding, audio_role);
        (video_filter, forward_interf):
              (video_binding, video_role);
    };
  };
};
```

Figure 4: An example specification of a composite binding.

information elements. Reflection 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.

Reflection in Meta-ORB can be used for dynamic inspection and adaptation in the context of both platform and application elements. To this end, the design of the meta-level follows the principles of the Open ORB reflective middleware architecture [1], as discussed below.

The meta-object protocol (MOP) is realised in terms of the interfaces of components that play the role of meta-

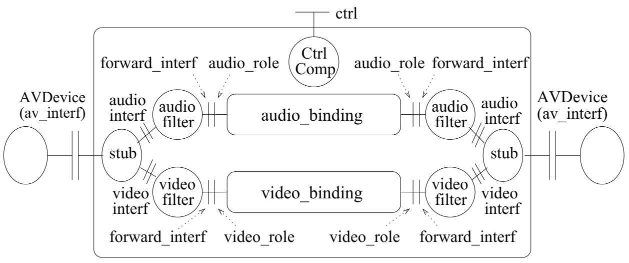

Figure 5: Composite binding for audio-video interaction (as described in Figure 4).

objects. In addition, 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.

In addition, considering the multitude of aspects that must be reified in reflective middleware, the meta-space is partitioned into a number of independent meta-space models. The approach is similar to the multi-model reflection framework introduced by [25]. Each separate concern 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 6 illustrates the concept of using distinct meta-objects (each one corresponding to a different meta-space model) to reify a given base-level object.

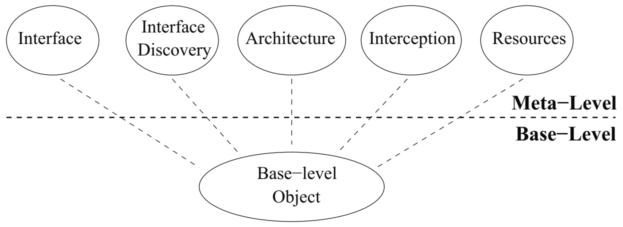

Figure 6: The meta-space reifying a base-level object.

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 [38]. The behavioural part of the meta-space consists of two meta-space models: *Resources* and *Interception*. These are however out of scope in this paper, so we will not further refer to them. Structural reflection, on its hand, is the focus of the prototype and 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)[1], 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

[1]In other implementations of the general OpenORB framework, these two meta-space models are merged into a single one, simply called Interface. The reason for partitioning them in Meta-ORB is to separate the functionality related to finding the interfaces of a component, at one hand, from that related to the dynamic discovery of the operations, flows or signals provided by a single interface.

way they are connected). These three meta-space models are designed so as to be independent of each other with regard to adaptation. This basically means that changes in the configuration of an object effected through the Architecture meta-space model need to respect the types of the interfaces involved. For instance, if a component is replaced by another, the substitute must provide the same interfaces of the replaced component (or interfaces derived from subtypes of the original interface types).

## 4.1 Examples of adaptation

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). The reason for this is to avoid possible incompatibilities (at the level of local bindings) that may arise from the addition or removal of interactions and interfaces. In future versions, this restriction may be removed with the adoption of rules (such as subtype-based evolution) to constrain the adaptations made via these two meta-space models.

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 specified in Figure 4, 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 circumstances of rigid middleware infrastructures, such as with conventional middleware, this would typically mean that the binding should be torn down. On the other hand, in the Meta-ORB reflective middleware, the Architecture meta-object may help overcome the problem in a more satisfactory way.

The solution could involve selecting 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 codec 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, due to the change of encoding). The code for implementing such reconfiguration, in Python, is shown in Figure 7.

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.

```
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
type_of_new_comp = MetaORB.TypeRep.lookup_name(
            'LowBandwidthVideoFilter', dk_Binding)

# Pause the binding, so that reconfiguration can
# take place without affecting its consistency
bind_ctrl.pause()

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

Figure 7: Example script for dynamic binding reconfiguration.

## 4.2 Combining reflection and meta-information management

In Meta-ORB the meta-information management facilities are organised around the concept of a repository. This repository provides for the storage, retrieval and consistency management of meta-information describing the building blocks of the platform. Such facility is described in terms of a set of meta-types, which make up the meta-model of the platform. Among the major meta-types are binding, component and interface, along with other more primitive elements (such as operations, flows and primitive data types). In addition, the meta-model is a direct extension of the CORBA 2.2 object model, meaning that all the constructs prescribed in that version of CORBA are also supported.

According to the usual functionality of a meta-information facility ([6]), the repository provides functions for registering new types, for checking type compatibility, and for the lookup and browsing of existing types. All these functions are meant (though not necessarily) to be automatically generated, based on the description of the meta-model, using MOF-related tools.

All configuration and reconfiguration facilities depend upon these meta-information management features. For instance, in order to create a platform configuration (e.g., a set of components and binding objects), the object factories need to obtain the right type definitions from the repository. In addition, the structural reflection features need to obtain meta-information describing the (type of the) base-level object, so that it can be properly reified.

However, a more subtle relationship between the meta-information management and reflection facilities may arise due to the fact that reification is strongly based on meta-information from the repository. An important requirement of every reflective system 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 successive adaptations, 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* [4], which means that the type of an object is changed (into a new version of the original type) once 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. Another consequence of using type evolution is the possibility to constrain adaptations based on type relationship rules, so that a dynamically created new type does not contradict the properties of the type used to derive it. This is important to keep compatibility with existing clients of an adapted object. Currently, we support subtyping as a type evolution rule, so that the new type must be a subtype (i.e., present all the interfaces) of the original type. The investigation of this approach as a way to check more global properties of the system remains an issue for future work.

# 5 Implementation

## 5.1 Python-based prototype

A prototype implementation of the Meta-ORB architecture has been developed with the goal of demonstrating its feasibility and applicability. 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 [36], which favours rapid prototyping instead. Despite this, experiments have shown that the performance of the prototype is appropriate for simple multimedia applications [3]. 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.

### 5.1.1 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 components). 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 the 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.

### 5.1.2 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 replica-tion 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 collaboration 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. Note that because type definitions (once stored in the repository) are immutable, the problem of keeping consistency among the replicas can be solved quite simply. The solution is based on the reliable distribution of newly created types to all replicas and on the uniqueness of type names and version numbers (which is guaranteed by the central manager). Finally, the Type Repository 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.

#### 5.1.3 Meta-level

This module corresponds to the mechanisms and facilities for structural reflection provided by the platform. It follows the framework described in section 4, 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 [3].

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, such as by creating and deleting componets, and disconnecting and reconnecting local bindings between their interfaces. In this way, meta-objects can perform the absorption of reflective computation (see Figure 1).

### 5.2 Java-based prototype

The first prototype, described above, was mainly aimed at demonstrating the concepts introduced in the Meta-ORB architecture. Current activity in the project is now targeting the development of a fully functional Java version of the platform, aimed at portability across a range of different platforms, as well as better performance. The focus is on exploring the dynamic adaptation facilities of Meta-ORB in mobile computing environments, especially involving handheld devices such as palmtops and mobile phones. This has led to the adoption of J2ME [34], according to the CLDC configuration [32] and the MIDP profile [33], as the main runtime environment. As a result, seamless portability and a smaller footprint of the runtime were naturally achieved, enabling the new version to run on a variety of devices.

An effort was made to retain total compatibility with the previous version, including the programming model and runtime data representation, thus enabling interoperability. However, due to the limitations of J2ME, a number of adaptations were needed in the core architecture. In particular, the use of version 1.0 of MIDP restricted us to HTTP as the sole communication protocol. This means that all primitive and implicit bindings are based on this protocol, having to provide all their features through conventional HTTP request/reply text-based messages, which may affect interaction performance. We expect to remove this constraint as implementations of the latest version of MIDP (2.0) become available, enabling the use of more capable datagram and socket-based connections. Another limitation of J2ME that has influenced the implementation was the lack of native reflection support (as available in the conventional Java Class Library). In particular, it is not possible to make dynamic method calls, which has compromised the flexibility of local bindings (connections between local interfaces) in the platform. As a result, the runtime representation of a component's interface (in terms of a Java class) has to be generated specifically for the particular interface type, as opposed to the generic, interface-independent, counterpart in the Python prototype. Note however, that this limitation is not related to the implementation of the reflection mechanisms of the platform, which are completely independent of the reflective features of a particular programming language.

Finally, limitations of the targeted execution environment, especially in terms of memory, processing power and battery, have led to the need to save as much resources as possible in this implementation. As a result, only the platform core was ported, consisting of the runtime infrastructure (capsule, local name server, component and binding factories, and implicit binding support). Some features, notably the Type Repository, were kept from the previous version (with some adaptations, discussed below, to enable the Python-Java interoperability), whereas others, such as the naming service, where implemented in the more capable J2SE platform. In what follows, a high level description of the architecture of the prototype is presented.

### 5.2.1 Architecture

As stated above, under CLDC/MIDP1.0, all networking has to be done through HTTP. Furthermore, only the client part of this protocol is implemented, meaning that a J2ME device cannot be the target of interactions (e.g., to receive requests). These two limitations have posed the need for a proxy-based architecture, where each J2ME device participating in the distributed environment of Meta-ORB must have a representative object residing in a more capable device located elsewhere (e.g., in the fixed network), which is able to receive interactions from clients (or media producers) and redirect them to the target object in the device. The overall architecture is illustrated in Figure 8, which shows the major elements involved in a platform infrastructure, along with the several possibilities of runtime environment (mainly J2ME and J2SE in this case).

As the figure shows, access to the Type Repository (in Python) is achieved through a servlet that redirects HTTP requests as appropriately formatted sockets-based messages to the closest Type Repository server. The requested type definitions are formatted as text-based messages before returning them to the caller. A similar approach is used for access to the global name server.

Figure 8 also illustrates access to component interfaces through implicit binding. This is achieved using a proxy-based approach, in a way similar to the Middleman architecture proposed in [23]. For each J2ME-based capsule, there is a proxy in charge of receiving and processing requests directed to interfaces located in that specific capsule, as well as handling the replies back. This proxy is created by the capsule manager at the same time as the capsule itself, by using a well-known configuration service (ConfigServer).[2] Communication between clients and the proxy can be either through HTTP (in the case of J2ME clients) or object requests (in the case of J2SE clients). Communication between the proxy and the target interface in the J2ME capsule is always based on HTTP, by superimposing an object request protocol on top of it. As J2ME-based objects (midlets) can only act as clients, requests directed to them have to be conveyed within normal HTTP reply messages. Such communication is based on a polling scheme, where the communications server object (CommsServer) residing in the J2ME capsule sends HTTP requests to the port associated with the proxy and waits for replies containing proper object requests. In other words, there is an inversion of the client and server roles.

As an example of this kind of interaction, consider the access to the (server) interfaces of component and binding factories in order to request the creation of components or the establishment of (explicit) bindings. In this example, the CommsServer object would poll (using an HTTP request message) the Proxy for newly arrived object requests. Such requests would be sent to the CommsServer in an HTTP reply message, which would then be parsed by the CommsServer in order to generate the appropriate local calls to the interfaces of the target components (in the example, the component factory or the binding factory components). The reply for an object request would be sent by the CommsServer to the Proxy via an HTTP request message. The Proxy would then handle the reply to the actual client in an appropriate way.

Finally, the figure also shows an explicit binding object connecting the interfaces of two remote components. Although not shown, the internal implementation of the binding (more precisely, the primitive binding inside of it), is based on a similar proxy mechanism as described above. In particular, the stub component at the J2ME side of the binding is a dedicated version of the communications server, while the stub at the other side takes the role of the (dedicated) proxy. In case both endpoints are based on J2ME capsules, we need an intermediate proxy (located in a J2SE capsule) to handle the HTTP request/reply messages properly.

### 5.2.2 Ongoing work

The above description corresponds to a lightweight version of the Meta-ORB base-level architecture. It fulfils the major elements of the core meta-model and enables the development of applications targeting mobile devices. In particular, all the facilities for flexible platform configuration are present under the headings of the component and binding factories, which take object definitions from the Type Repository and perform the instantiation of customised platform configurations.

The reflective meta-level, on the other hand, is the subject of ongoing work. The focus will again be on the structural part of the meta-space, notably on the architecture meta-space model. This will enable us to evaluate the approach for dynamic reconfiguration in a mobile, resource-

[2] Although not shown in the figure, the ConfigServer need not be in the same capsule as the proxy.

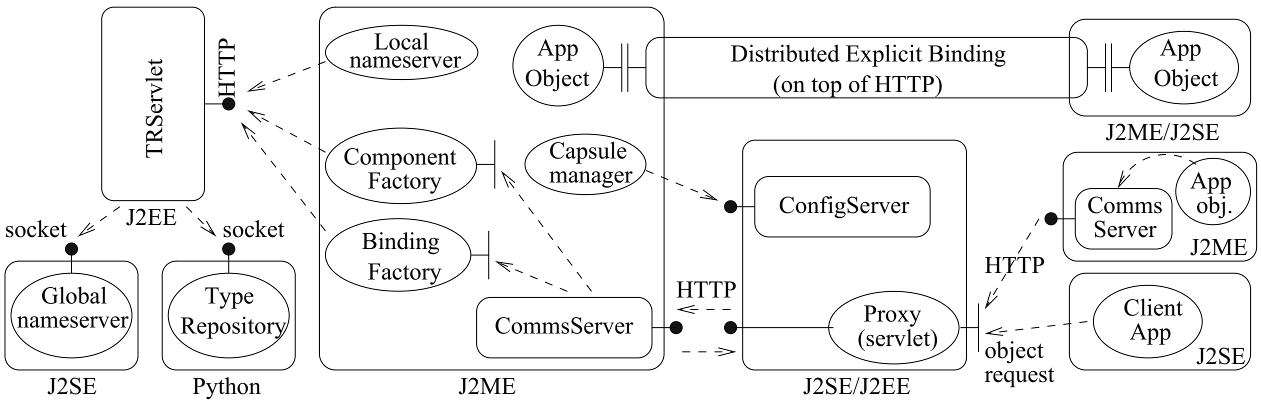

Figure 8: Overall architecture of the Java-based Meta-ORB prototype.

constrained environment, which provides the most interesting requirements for this kind of capability.

## 6 Performance evaluation

In this section we present a detailed performance evaluation of the Meta-ORB Python-based prototype according to three aspects: static configuration, interaction, and reflective reconfiguration. This is provided with a note of caution, due to the interpreted nature of Python.[3] However, the aim is to highlight the relative overhead compared with non-reflective platforms and with the demands of distributed multimedia. Indeed, the results are quite encouraging, giving a rough indication of the level of performance that can be achieved in more efficient language environments.

All experiments were conducted on a 10Mbps Ethernet LAN, using identical Pentium III 800MHz PCs with 256MB RAM, running Windows 2000 and Python 2.1. All measurements were taken using the clock function of the standard Python library, with several runs and averaging to smooth the effects of non-determinism introduced by the OS scheduler and network.

### 6.1 Static configuration performance

As particular instances of the platform are made up with component and binding objects, an evaluation of their instantiation performance is crucial to understand the cost of establishing complete platform infrastructures.

The cost of component instantiation is shown in Figure 9 for three representative cases: primitive components with a varying number of interfaces, composite components made up with flat compositions of primitive components, and composite components with a recursive composition pattern (i.e., hierarchically nested components). As can be seen, the cost scales up linearly with the component's complexity. The graph also gives a rough idea about the cost incurred by other component configurations (e.g., the instantiation of a flat composite component with five internal primitive components and three interfaces will cost approximately 48ms – 38ms for the composition plus 10ms for its three interfaces).

The next experiment shows the performance and scalability of binding instantiation. It considers multi-point bindings with up to six endpoints, each one located in a different machine and consisting of a stub and the endpoint of a primitive binding. The use of such minimal bindings is so that the inherent cost of instantiating distributed bindings is made more evident (the cost of more complex bindings would be the sum of the cost of a simple binding plus that of their internal components).

Given the distributed nature of the binding protocol, where binding endpoints are created in parallel by different local binding factories (see [3]), the impact of the number of endpoints is made less significant. The increases shown in Figure 10 are mainly due to the additional processing performed by the primary binding factory (which coordinates the whole process) to stich the several endpoints together. Nevertheless, the results seem to suggest that such increases tend to attenuate as the number of binding endpoints grow.

### 6.2 Interaction performance

This section discusses the performance of interaction between remote components. Whenever applicable, results are contrasted with two other Python-based platforms:

[3]A corresponding evaluation of the Java-based implementation is the subject of ongoing work.

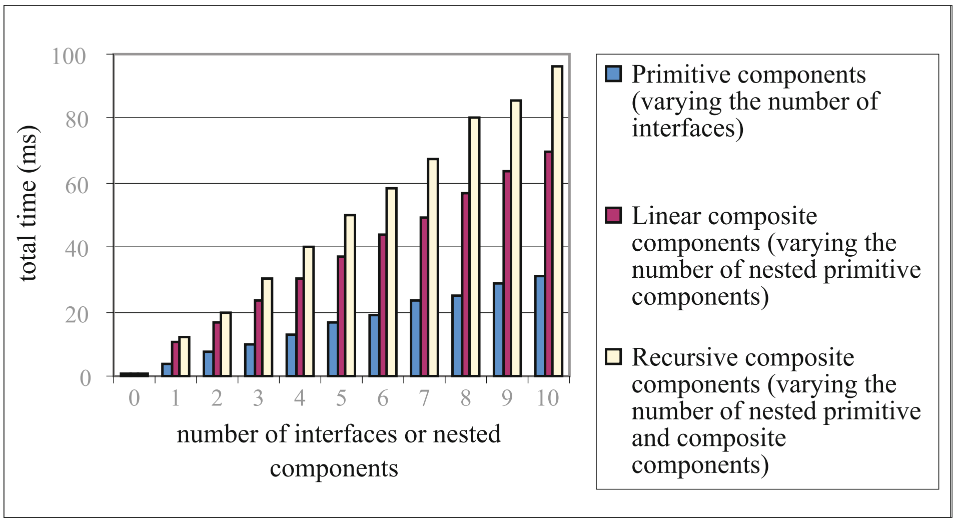

Figure 9: Cost of component instantiation.

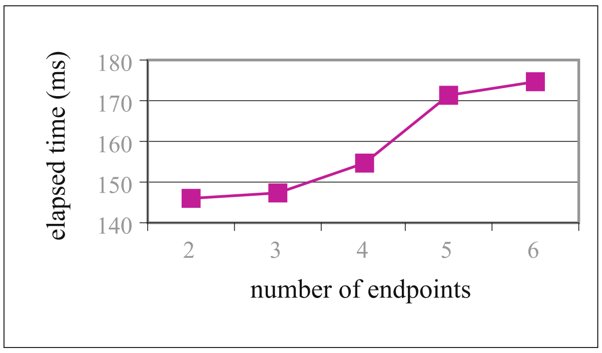

Figure 10: Cost of binding instantiation.

Fnorb [13], a CORBA-compliant middleware platform; and minimal implementations based on TCP/UDP sockets (in order to highlight the overhead introduced by the Meta-ORB programming model). All experiments use the same binding configuration as above, though with only two endpoints.

### 6.2.1 End-to-end delay

The first experiment, shown in Figure 11, illustrates, in logarithmic scale, the round trip delay for request-reply interaction. The comparison with TCP-based sockets shows an overhead of about 30% for small interactions (up to 128 bytes), although the overhead decreases for larger interactions (above 4KB, it does not exceed 10%). This extra cost can be explained by the higher level of abstraction of the Meta-ORB programming model. The complementary analysis shows that the Meta-ORB framework can be used to achieve superior performance in comparison with Fnorb (which is at least about 5ms slower in all cases). In part, this is due to the extra processing performed on an invocation by Fnorb, such as marshaling/unmarshaling, which is significantly more generic than in our prototype. This demonstrates that reflective middleware can achieve better performance by removing unnecessary overhead.

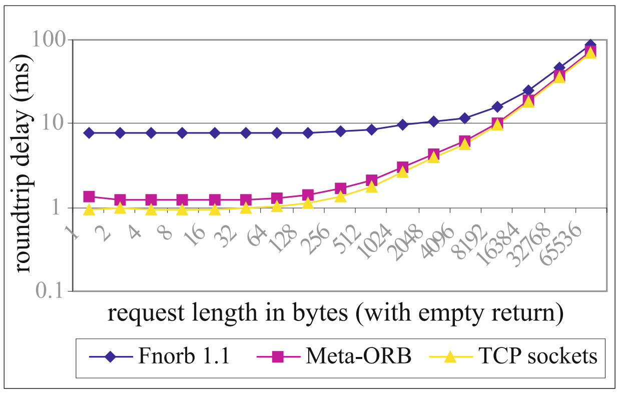

Figure 11: Round trip delay for request-reply interaction in a distributed binding.

An analysis of the end-to-end delay of stream interactions, comparing a Meta-ORB binding with UDP sockets, is shown in Figure 12 (a comparison with Fnorb does not apply, as it does not have support for streams). The delay is estimated by halving the round-trip time.

Compared with the typical delay requirements of multimedia applications (maximum of 250ms for both audio and video, according to [16]), these figures seem appropriate, especially considering frames of moderate sizes (up to 8KB). However, the extra cost of processing complex media in the binding should also be taken into account, suggesting that such functionality should be implemented in C/C++ and integrated into the platform using Python's extension facilities [37].

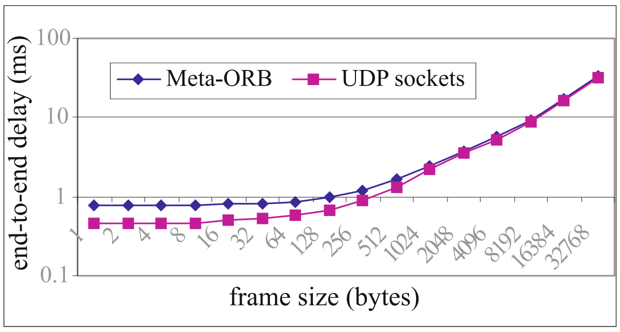

Figure 12: End-to-end delay of stream interaction.

### 6.2.2 Throughput

Figure 13 shows the result of an experiment measuring the user-level throughput of a stream binding on a lightly loaded network, along with a comparison with UDP sockets. As the graph shows, for frame sizes over 32 bytes, the absolute difference in achievable throughput is nearly constant (300-700 Kbits/s). In relative terms, the throughput of a Meta-ORB binding is only 10 percent lower for frame sizes over 512 bytes. This lower throughput is a result of the high-level programming model of Meta-ORB, showing a tradeoff between programmability and raw performance.

Finally, to put the figures in perspective, they can also be compared with the typical requirements of continuous media. In particular, for audio streams, the throughput is clearly suitable. For video streams, however, it can be less than adequate. Nevertheless, considering the requirements of compressed TV-quality video (2-10Mbit/s), as well as the use of larger (over 512 bytes) frame sizes, reasonable results may be achieved.

## 6.3 Reflection performance

### 6.3.1 Meta-object instantiation

The performance of reification is examined here using several experiments that illustrate the creation of meta-objects according to the three structural meta-space models described earlier.

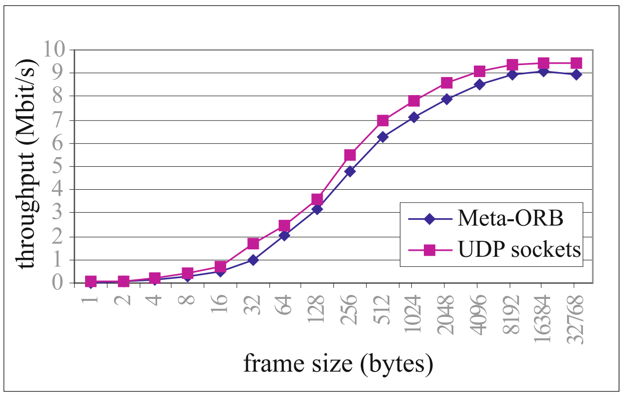

Figure 13: Composite binding throughput.

The time required to create Interface Discovery meta-objects is independent of the particular base-level object to be reified, being around 8.8ms for components and 9.8ms for bindings. Similarly, for Interface meta-objects, reification time is independent of the particular interface, and was about 9.5ms.

On the other hand, for Architecture meta-objects, performance depends on the complexity of the particular base-level object. Two experiments were chosen to show the scalability of architecture reification: firstly, for composite components, using linear composition and a varying number of nested components; and secondly, for binding objects with an increasing number of endpoints (with the simple binding configuration described in 6.1).

The results have shown that reification time is linearly proportional to the number of features present in the base-level object. For components with one internal component, reification takes 10.6ms, with each extra internal component adding up about 0.4ms (e.g., reifying a component with 5 internal components would take 12.6ms). For binding objects, both the number of internal components and binding endpoints influence reification time. We observed that for simple bindings with two endpoints, architectural reification takes about 19.9ms, with each extra endpoint adding up about 0.2ms.

These figures indicate that meta-objects should ideally be created in advance of the need for reflection, especially in the case of Architecture meta-objects and in time-critical applications. Once meta-objects are created, though, their access time is less significant.

### 6.3.2 Adaptation performance

This is a critical issue, as adaptation mechanisms are meant for dynamic use, while the platform is running. In order to demonstrate the level of performance in the current implementation, three experiments were run, using the most common adaptation operations. In order to isolate the inherent cost of reflective adaptation, all components involved in the adaptations are primitive. (Adaptations involving complex, composite, components would have the added costs incurred in the instantiation of the component's internal configuration.) The results, for binding adaptation, are as follows: inserting a new component takes 37.5ms, while component removal takes 44ms and component replacement takes about 90ms. Adaptation of components, in turn, require slightly lower times, as both base- and meta-objects would always be local to each other.

Considering the typical requirements of continuous media, in particular the short inter-frame intervals for audio and video streams, the above results seem to suggest a need for more efficient meta-object implementations, e.g., in C or C++, in order to reduce the possibility of frame loss or even to schedule a given adaptation in between frame arrivals.

## 7 Related work

In the context of the Open ORB architecture, several prototypes have been implemented, each exploring a different aspect of the architecture. In particular, the OpenCOM runtime component model, together with the ReMMoC middleware built on top of it [15, 2], strive for efficiency of implementation and memory footprint. This platform was written in C++, which, besides being an efficient compiled language, allows access to low-level features (such as virtual pointer tables), which greatly optimise performance. The performance of this platform has been shown to be on a par with non-reflective middleware, with the added benefit of further optimisations that can be achieved with the very use of reflective adaptation [5]. Our approach has several similarities with this work, especially the use of reflection for platform optimisation. However, the aim here was to highlight the benefits of the combined use of meta-information management, also showing that it does not impose considerable overhead.

Another relevant outcome of the Open ORB project was the FORMAware framework, aimed at the management of adaptation in component architectures [7]. The approach is based on software architectures as a way to fully describe configurations of components, together with constraints that specify the criteria for validating reconfigurations. For dynamic reconfiguration, a comprehensive meta-object protocol is provided, which enables the handling of all aspects of an architecture. We note that such an approach can conceptually complement our architecture, fitting into the scope of the Architecture meta-space model. In addition, our approach towards the integration of reflection and meta-information management can further leverage the idea of architecture adaptation, especially regarding the seamless integration of configuration and reconfiguration, as well as the notion of type evolution discussed in 4.2, which could use architectural constraints as a basis to validate adaptations.

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 [21], 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 [8] 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.

In parallel with our effort to build a flexible Java-based middleware platform for mobile computing, it is worth mentioning the work carried out in the Arcademis project, which is building a framework for the implementation of customised middleware [29]. Similar to our work, Arcademis also targets the problem of middleware customisability, especially in the context of mobile computing. Their approach is based on a set of abstract classes and interfaces describing general middleware functionality, which can be specialised to produce particular kinds of middleware. One such example is the use of the framework to build an object-oriented middleware for J2ME-enabled devices. Their approach thus differs from ours in the way middleware configurations are specified, which in our case is based on an object definition language. However, in the same way as FORMAware, we can conceive an integration of the two approaches, using frameworks as a way to constrain configuration definitions. Another important difference, however, is the absence of support for runtime reflection and dynamic reconfiguration in Arcademis, although such support could possibly be developed following our overall approach and having a runtime representation of the underlying component framework.

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 [11]). 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.

Finally, we also mention the efforts in the area of aspect-oriented programming (AOP) [19], which is similar to reflection in the sense that it is also a technique to enable separation of concerns in systems such as middleware. The original proposals of AOP were targeted at static aspects, which are combined by the weaving process andthus made unavailable at runtime. There is however a number of research efforts in the direction of dynamic aspects [12], which preserve the distinction among the aspects of a system at runtime, enabling new aspects to be added and old ones to be removed or replaced. We consider reflection and AOP complementary techniques, as reflection (and meta-object protocols) can be used as the mechanism enabling the dynamic manipulation of aspects. In this sense, aspects can be seen as another approach to structure the middleware system, in a way that is orthogonal to the way component composition is used in our approach. Further investigation about this combined use of aspects, components and reflection in Meta-ORB remains an issue for future work.

## 8 Concluding remarks

This paper has presented Meta-ORB, a reflective middleware platform based on a combination of a meta-level architecture with meta-information management concepts. The overall aim of the research is to develop 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, together with their application in the context of middleware. The paper discussed the architecture of the platform, together with its two implementations, a Python-based proof-of-concept prototype, and a Java-based implementation, which is currently under development and targets wireless mobile devices. The paper also presents an evaluation of the approach based on the first prototype.

The work has enabled us to draw some important conclusions about the design and implementation of adaptive middleware platforms. The most important of such conclusions is related to the benefits of an integrated approach combining runtime reflection an explicit runtime meta-model representation of the platform. This enables the use of the same set of abstractions for configuring a platform from scratch and for adapting it at runtime, relieving the user from the need to learn a different set of concepts and tools. In addition, the concept of type evolution has shown to be an important step towards enhancing the process of developing customised middleware, as new versions of a platform configuration can be produced at runtime, by successive adaptations in order to match real operation scenarios. The most promising versions can then be turned into proper configuration definitions and stored in the Type Repository for later use in order to reproduce the successful evolved configurations in other contexts. We believe this is a promising approach to software development in general, and to adaptive middleware in particular.

Another important conclusion is related to the impact of both techniques, reflection and meta-information management, on the overall performance of the platform. The experiments presented in the paper demonstrate that such impact, though not negligible, is within acceptable limits for some important categories of application and is comparable with the performance of non-reflective platforms. Furthermore, we have identified several points for improvement, mainly related to the implementation environment. For example, implementing the more computing intensive components of the platform in an efficient language, such as C++ (while still taking advantage of productivity benefits of Python for placing the components together), would improve several of the performance figures presented in the paper. Thus, we can argue that the main performance bottleneck of the prototype is not the reflective programming model itself. Future work will investigate the above argument with the development of primitive components and component factories implemented in C++, in order to verify if the related performance impact is significant. Ongoing work is also investigating the performance of the platform in environments with more limited resources available, notably comprising the J2ME-based prototype in handheld computers connected by wireless LANs.

## References

[1] Gordon S. Blair, Fábio M. Costa, Katia Saikoski, and Nikos Parlavantzas Hector Duran Mike Clarke. The design and implementation of Open ORB version 2. *IEEE Distributed Systems Online Journal*, 2(6), 2001.

[2] Mike Clarke, Gordon S. Blair, and Geoff Coulson. An efficient component model for the construction of adaptive middleware. In *Proceedings of the IFIP/ACM International Middleware Conference (Middleware'2001)*, Heidelberg, Germany, 2001. Springer-Verlag.

[3] Fábio M. Costa. *Combining Meta-Information Management and Reflection in an Architecture for Configurable and Reconfigurable Middleware*. Ph.D. thesis, University of Lancaster, Lancaster, UK, September 2001. http://www.comp.lancs.ac.uk/computing/users/fmc/pubs/thesis.pdf.

[4] Fábio M. Costa and Gordon S. Blair. Integrating reflection and meta-information management in middleware. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'00)*, Antwerp, Belgium, 2000. IEEE, IEEE.

[5] Geoff Coulson, Gordon S. Blair, and Paul Grace. On the performance of reflective systems software. In *Proceedings of the International Workshop on Middleware Performance (IWMP'04)*, Phoenix, AZ, April 2004. IEEE Computer Society.

[6] Steve Crawley, S. Davies, Jaga Indulska, Simon McBride, and Kery Raymond. Meta-information management. In *Proceedings of the 2nd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'97)*, Canterbury, UK, 1997. IFIP.

[7] Rui Jorge da Silva Moreira. *FORMAware: Framework Of Reflective components for Managing architecture Adaptation*. Ph.d. thesis, Computing Department, Lancaster University, Lancaster, UK, March 2004.

[8] Fabio Kon et al. Monitoring, security and dynamic configuration with the DynamicTAO reflective ORB. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, 2000.

[9] H. Mili et al. Metamodelling in OO - workshop summary. In *Addendun to the Proceedings of OOPSLA'95*, Austin, TX, 1995.

[10] Stephen Crane et al. Configuration management for distributed software services. In *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (ISINM'95)*, Santa Barbara, CA, 1995. IFIP/IEEE.

[11] Waine Brookes et al. Types and their management in open distributed systems. *Distributed Systems Engineering*, 4(1):177–190, 1997.

[12] Robert Filman, Michael Haupt, Katharina Mehner, and Mira Mezini, editors. *Proceedings of the 2004 Dynamic Aspects Workshop (DAW04)*, volume 1, Lancaster, UK, March 2004. Research Institute for Advanced Computer Science. RIACS Technical Report 04.01.

[13] Fnorb. *Fnorb - release 1.1*. CRC for Distributed Systems Technology, University of Queensland, Queensland, Australia, 2000. http://www.fnorb.org.

[14] Brian Foote. Object-oriented reflective metalevel architectures: Pyrite or panacea? In *Proceedings of ECOOP/OOPSLA'90 Workshop on Reflection and Metalevel Architectures*, Ottawa, 1990. ACM.

[15] Paul Grace, Gordon S. Blai, and Sam Samuel. ReMMoC: A reflective middleware to support mobile client interoperability. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'03)*, Sicily, Italy, 2003. IEEE, IEEE Computer Society.

[16] D. B. Hehnmann, M. G. Salmony, and H. J. Stuttgen. Transport services for multimedia applications on broadband networks. *Computer Communications*, 13(4):197–203, 1990.

[17] ITU-T/ISO. *ITU-T X.901 | ISO/IEC 10746-1 Open Distributed Processing Reference Model - Part 1: Overview*, 1995.

[18] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[19] Gregor Kiczales, J. Lamping, C. Maeda A. Mendhekar, C.V. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 220–241, Jyvaskila, Finland, June 1997. Springer-Verlag.

[20] Lea Kutvonen. Management of application federations. In *Proceedings of the International IFIP Workshop on Distributed Applications and Interoperable Systems (DAIS'97)*, Cottbus, Germany, 1997. IFIP.

[21] Thomas Ledoux. OpenCORBA: A reflective open broker. In *Proceedings of the 2nd International Conference on Reflection and Meta-level Architectures (Reflection'99)*, St. Malo, France, 1999. Springer-Verlag.

[22] Patie Maes. Concepts and experiments in computational reflection. In *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, Orlando, FL USA, 1987. American Computer Machinery, ACM Press.

[23] Qusay Mahmoud. Advanced MIDP networking, acessing using sockets and RMI for MIDP-enabled devices. Technical report, Sun Microsystems, Inc., January 2002. http://developers.sun.com/techtopics/mobility/midp/articles/socketRMI/.

[24] Jacques Malenfant, M. Jacques, and F. Demers. A tutorial on behavioural reflection and its implementation. In *Proceedings of Reflection'96*, San Francisco, 1996.

[25] Hideaki Okamura, Yasuhiru Ishikawa, and Mario Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on New Models for Software Architecture (IMSA'92)*, 1992.

[26] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Needham, MA, rev. 2.2 edition, 1998.

[27] OMG. *Meta Object Facility (MOF)*. Object Management Group, Needham, MA, 2000. OMG Document formal/2000-04-03.

[28] OMG. *The Common Object Request Broker: Architecture and Specification*. Object Management Group, Needham, MA USA, rev. 3.0 edition, 2003.

[29] Fernando Magno Pereira, Danielle Gordiano Valente, Geraldo Robson Mateus, and Antonio Alfredo Ferreira Loureiro. Arcademis: A java-based framework for middleware development. In *Proceedings of the 22nd Brazilian Symposium on Computer Networks (SBRC'2004)*, Gramado, RS, 2004. SBC.

[30] Douglas Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, 2000.

[31] Brian C. Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT Laboratory of Computer Science, 1982. MIT Technical Report 272.

[32] Sun. *Connected Limited Device Configuration (CLDC)*. Sun Microsystems, Inc., 2004. http://java.sun.com/products/cldc/index.jsp.

[33] Sun. *J2ME Mobile Information Device Profile (MIDP)*. Sun Microsystems, Inc., 2004. http://java.sun.com/products/midp/index.jsp.

[34] Sun. *Java 2 Platform Micro Edition*. Sun Microsystems, Inc., 2004. http://java.sun.com/j2me/.

[35] Clemens Szyperski. *Component Software: Beyond object-orientation*. Addison-Wesley, 1997.

[36] Guido van Rossun. *Python Documentation, version 2.3*. Python Labs., 2002. http://www.python.org/doc/.

[37] Guido van Rossun and F. L. Drake. Extending and embedding the Python interpreter. Technical report, Python Labs., 2001. http://www.python.org/doc/current/ext/ext.html.

[38] Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'88)*, San Diego, CA, 1988. ACM, ACM Press.