The Design of a Resource-Aware Reflective Middleware Architecture

Gordon S. Blair[1], Fábio Costa[1], Geoff Coulson[1], Fabien Delpiano[2], Hector Duran[1], Bruno Dumant[2], François Horn[2], Nikos Parlavantzas[1], Jean-Bernard Stefani[2]

[1]Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK.
{gordon, geoff, fmc, duranlim, parlavan}@comp.lancs.ac.uk

[2]CNET, France Télécom, 38-40 rue Général Leclerc, 92794 Issy les Moulineaux, CEDEX 9, Paris, France.
{fabien.delpiano, bruno.dumant, francois.horn, jeanbernard.stefani}@cnet.francetelecom.fr

Abstract. Middleware has emerged as an important architectural component in supporting distributed applications. With the expanding role of middleware, however, a number of problems are emerging. Most significantly, it is becoming difficult for a single solution to meet the requirements of a range of application domains. Hence, the paper argues that the next generation of middleware platforms should be both configurable and re-configurable. Furthermore, it is claimed that reflection offers a principled means of achieving these goals. The paper then presents an architecture for reflective middleware based on a multi-model approach. The main emphasis of the paper is on resource management within this architecture (accessible through one of the meta-models). Through a number of worked examples, we demonstrate that the approach can support introspection, and fine- and coarse- grained adaptation of the resource management framework. We also illustrate how we can achieve multi-faceted adaptation, spanning multiple meta-models.

1. Introduction

Middleware has emerged as a key architectural component in supporting distributed applications. The role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution. The importance of the topic is reflected in the increasing visibility of industrial standardisation activities such as OMG's CORBA and Microsoft's DCOM.

The expanding role of middleware is however leading to a number of problems. Most importantly, middleware is now being applied to a wider range of application domains, each imposing their own particular requirements on the underlying middleware platform. Examples of this phenomenon include real-time multimedia systems (requiring an element of quality of service management), and embedded systems (requiring small footprint technologies). In our opinion, this implies that future middleware platforms should be *configurable* in terms of the components used in the construction of a given instance of a platform. With the advent of mobile computing, changes may also occur in the underlying network environment. Given this, it is also becoming increasingly important that middleware platforms are *re-configurable*, i.e. underlying components can be changed at run-time. Finally, and of central importance to this paper, we also believe that future middleware platforms should provide sophisticated facilities for *resource management*, in terms of allocating (potentially) scarce resources, being aware of patterns of resource usage, and adapting to changes in resource patterns.

The authors believe the general solution to these problems is to introduce more openness and flexibility into middleware technologies. In particular, we advocate the use of reflection [Smith82, Kiczales91] as a means of achieving these goals. Reflection was first introduced in the field of programming languages and has recently been applied to a range of other areas including operating systems and windowing systems (see section 5). We believe that similar techniques can usefully be applied to middleware technology. This solution contrasts favourably with existing approaches to the construction of middleware platforms, which typically adopt a black box philosophy whereby design decisions are made by the platform developers in advance of installation and then hidden from the programmer at run-time.

Note that some flexibility has been introduced into middleware products. For example, CORBA now supports a mechanism of interception whereby wrappers can be placed round operation requests. However, this can be viewed as a rather ad hoc technique for achieving openness. The Portable Object Adaptor can also be criticised in this way. Some work has been carried out on developing more open ORBs. For example, the TAO platform from the University of Washington offers a more open and extensible structure, and also features the documentation of software patterns as used in the construction of TAO [Schmidt97]. TAO also offers access to key resource management functionality. Similarly, BBN's QuO platform employs techniques of open implementation to support quality of service management, in general, and adaptation, in particular [Vanegas98]. Finally, a number of researchers have recently experimented with reflective middleware architectures (see section 5). These platforms address many of the shortcomings of middleware platforms. However, none of these initiatives offer a complete and principled reflective architecture in terms of configurability, re-configurability and access to resource management.

Based on these observations, researchers at Lancaster University and CNET are developing a reflective middleware architecture as part of the Open ORB Project. This architecture supports introspection and adaptation of the underlying components of the middleware platform in terms of both structural and behavioural reflection [Watanabe88]. Crucially, the architecture also supports reification of the underlying resource management framework. This paper discusses the main features of this architecture with particular emphasis on the resource management aspect of our work.

The paper is structured as follows. Section 2 presents an overview of the reflective middleware architecture highlighting the multi-model approach that is central to our design. Section 3 then focuses on resource management, discussing the resource management framework in detail and considering the integration of this framework into the reflective architecture. Following this, section 4 presents three contrasting examples illustrating how our architecture can support introspection, fine- and coarse-grained adaptation, and resource-aware adaptation. Section 5 then discusses some related work and section 6 offers some concluding remarks.

2. Open ORB Architecture

2.1. Overall Approach

In common with most of the research on reflective languages and systems, we adopt an *object-oriented model of computation*. As pointed out by Kiczales et al [Kiczales91], there is an important synergy between reflection and object-oriented computing:

> *"Reflective techniques make it possible to open up a language's implementation without revealing unnecessary implementation details or compromising portability; and object-oriented techniques allow the resulting model of the language's implementation and behaviour to be locally and incrementally adjusted".*

The choice of object-orientation is important given the predominance of such models in open distributed processing [Blair97]. Crucially, we propose the use of the RM-ODP Computational Model, using CORBA IDL to describe computational interfaces. The main features of this object model are: i) objects interact with each other through *interfaces*, ii) objects can have *multiple interfaces*, iii) interfaces for *continuous media interaction* are supported, iv) interfaces are uniquely identified by *interface references*, and v) *explicit bindings* can be created between compatible interfaces (the result being the creation of a *binding object*). The object model also has a sophisticated model of quality of service including QoS annotation on interfaces. Further details of this object model can be found in [Blair97]. In contrast, with RM-ODP, however, we adopt a consistent object model throughout the design [Blair98].

As our second principle, we adopt a *procedural* (as opposed to a declarative) approach to reflection, i.e. the meta-level (selectively) exposes the actual program that implements the system. In our context, this approach has a number of advantages over the declarative approach. For example, the procedural approach is more primitive (in particular, it is possible to support declarative interfaces on top of procedural reflection but not vice versa[1]). Procedural reflection also opens the possibility of an infinite tower of reflection (i.e. the base level has a meta-level, the meta-level is implemented using objects and hence has a meta-meta-level, and so on). This is realised in our approach by allowing such an infinite structure to exist in theory but only to instantiate a given level on demand, i.e. when it is reified. This provides a finite representation of an infinite structure (a similar approach is taken in ABCL/R [Watanabe88]). Access to different meta-levels is important in our design although in practice most access will be restricted to the meta- and meta-meta-levels.

The third principle behind our design is to support *per object* (or *per interface*) meta-spaces. This is necessary in a heterogeneous environment where objects will have varying capacities for reflection. Such a solution also provides a fine level of control over the support provided by the middleware platform; a corollary of this is that problems of maintaining integrity are minimised due to the limited scope of

[1] Indeed, we have already experimented with constructing more declarative interfaces on top of our architecture in the context of the resources framework.

change. We do recognise however that there are situations where it is useful to be able to access the meta-spaces of sets of objects in one single action; to support this, we also allow the use of (meta-object) *groups* (see discussion in 2.2 below).

The final principle is to structure meta-space as a number of closely related but distinct *meta-space models.* This approach was first advocated by the designers of AL-1/D, a reflective programming language for distributed applications [Okamura92]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects. Further details of each of the various models can be found below.

2.2. The Design of Meta-space

2.2.1. Supporting Structural Reflection

In reflective systems, structural reflection is concerned with the content of a given object [Watanabe88]. In our architecture, this aspect of meta-space is represented by two distinct meta-models, namely the *encapsulation* and *composition* meta-models. We introduce each model in turn below.

The *encapsulation meta-model* provides access to the representation of a particular interface in terms of its set of methods and associated attributes, together with key properties of the interface including its inheritance structure. This is equivalent to the introspection facilities available, for example, in the Java language, although we go further by also supporting adaptation. Clearly, however, the level of access provided by the encapsulation model will be language dependent. For example, with compiled languages such as C access may be limited to introspection of the associated IDL interface. With more open (interpreted) languages, such as Java or Python, more complete access is possible such as being able to add or delete methods and attributes. This level of heterogeneity is supported by having a type hierarchy of encapsulation meta-interfaces ranging from minimal access to full reflective access to interfaces. Note however that it is important that this type hierarchy is open and extensible to accommodate unanticipated levels of access.

In reality, many objects will in fact be composite, using a number of other objects in their construction. In recognition of this fact, we also provide a *compositional meta-model* offering access to such constituent objects. Note that this meta-model is associated with each object and not each interface. More specifically, the same compositional meta-model will be reached from each interface defined on the object (reflecting the fact that it is the object that is composite). In the meta-model, the composition of an object is represented as an *object graph* [Hokimoto96], in which the constituent objects are connected together by *local bindings*[2]. The interface offered by the meta-model then supports operations to inspect and adapt the graph structure, i.e. to view the structure of the graph, to access individual objects in the graph, and to adapt the graph structure and content.

This meta-model is particularly useful when dealing with binding objects [Blair98]. In this context, the composition meta-model reifies the internal structure of the

[2] This RM-ODP inspired concept of local binding is crucial in our design, providing a language-independent means of implementing the interaction point between interfaces.

binding in terms of the components used to realise the end-to-end communication path. For example the object graph could feature an MPEG compressor and decompressor and an RTP binding object. The structure can also be exposed recursively; for example, the composition meta-model of the RTP binding might expose the peer protocol entities for RTP and also the underlying UDP/IP protocol. It is argued in [Fitzpatrick98] that open bindings alone provide strong support for mobile computing.

2.2.2. Supporting Behavioural Reflection

Behavioural reflection is concerned with activity in the underlying system [Watanabe88]. This is represented by a single meta-model associated with each interface, the *environmental meta-model*. In terms of middleware, the underlying activity equates to functions such as message arrival, enqueueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [Watanabe88, McAffer96].

Again, different levels of access are supported. For example, a simple meta-model may enable the insertion of pre- and post- methods. Such a mechanism can be used to introduce, for example, some additional levels of distribution transparency (such as concurrency control). In addition, additional functions can be inserted such as security functions. A more complex meta-model may allow the inspection or adaptation of each element of the processing of messages as described above. With such complexity, it is likely that such a meta-model would itself be a composite object with individual components accessed via the compositional meta-space of the environmental meta-space. As with the other meta-models, heterogeneity is accommodated within an open and extensible type hierarchy.

2.2.3. The Associated Component Library

To realise our open architecture, we provide an open and extensible library of components that can be assembled to build middleware platforms for a range of applications. By the term component, we mean a robust, packaged object that can be independently developed and delivered. Our components adhere to a component framework that offers a built in event notification service and also access to each of the meta-models.

The component library consists of both primitive and composite components, the distinction being that primitive components are not open to introspection or adaptation in terms of the compositional meta-model. Examples of primitive components include communication protocols such as UDP and TCP, and end system functions such as continuous media filters. Composite components then represent off-the-shelf configurations of components such as end-to-end stream bindings. Importantly, we also provide *group bindings* (or simply groups) as composite objects. The role of groups is to provide a uniform mechanism for invoking a set of interfaces whether they are at the base-level, meta-level, meta-meta-level, etc. Furthermore, through access to their underlying object graph of the group component, it is possible to tailor and adapt the semantics of group message distribution, e.g. by introducing a new ordering protocol.

2.3. Management of Consistency

In reflective architectures, care must be taken to ensure the consistency of the underlying virtual machine. This is particularly true in the reflective middleware architecture described above, as the internal structure of the underlying platform can be changed significantly by using the meta-object protocols associated with the meta-models. For example, using the encapsulation meta-model, it is possible to introduce new methods or attributes into a running object. Similarly, the compositional and environmental meta-models allow the programmer to change the construction of a composite object and the interpretation of method invocation respectively. Given this, steps must be taken to ensure that consistency constraints are maintained.

As can be seen from the architecture, the most common style of adaptation is to alter the structure of an object graph. Consequently, it is crucial that such run-time alterations can be made in a safe manner. Some support is provided by the strongly-typed object model in that local bindings will enforce type correctness across interacting objects. This is however not a complete solution to the problem. The following steps are also necessary[3]:

1. Any activity in the affected region of the graph must be suspended;
2. The state of such activity must be preserved;
3. Replacement components must maintain the desired level of quality of service.

In addition, it is also necessary to maintain global consistency across the graph, i.e. changes of one component may require changes to another component (as in the case when replacing a compression and decompression object).

Responsibility for enforcing consistency constraints may be left to the application in our architecture. However, it is much more likely that the application will devolve responsibility to QoS management components that monitor the underlying object graph and implement an appropriate adaptation policy. The design of such a QoS management infrastructure for our reflective architecture is discussed in [Blair99].

We return to this important issue in section 4 below.

3. Resource Meta-Model

3.1. The Resource Framework

3.1.1. The Overall Approach

The main aim of the resource framework is to provide a complete and consistent model of all resources in the system at varying levels of abstraction. For example, a consistent approach should be taken to the management of processing capacity, memory, and network resources. Similarly, the framework should encompass more abstract resources such as virtual memory, persistent virtual memory, connections, teams of threads, and more general collections of resources.

[3] This is similar to the problem of (nested) atomic transactions, but with the added requirement of maintaining the quality of service constraints [Mitchell98].

This is achieved by recursively applying the structure illustrated in figure 1.

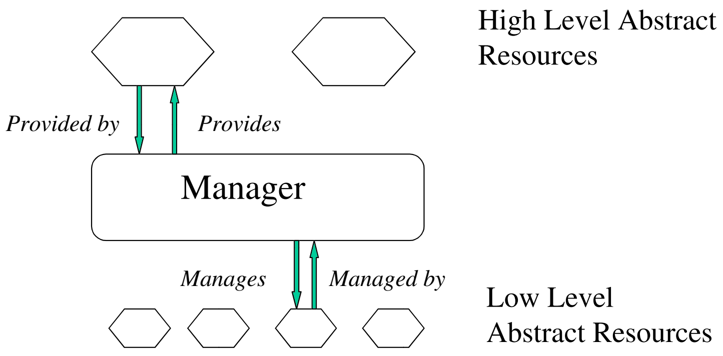

Fig. 1. The Overall Resource Framework

This diagram shows how higher level resources are constructed out of lower level resources by resource manager objects [ReTINA99]. Managers *provide* the higher level resources and *manage* the lower level resources. Similarly, higher level resources are *provided by* managers and lower level resources are *managed by* managers (we return to this in section 3.3). This structure maps naturally on to facilities offered by most modern operating systems. Note that there is an assumption that access to resource management is possible. At the very least, this will be possible in terms of user level resources such as user level threads. The level of access can also be enhanced through techniques such as split-level resource management [Blair97].

Both the high level and low level resources are of type `AbstractResource` as defined by the following interface:

```
interface AbstractResource {
 Manager provided_by();
 Manager managed_by();
 void release();
}
```

The `provided_by()` operation enables an application to locate the manager for a high level resource. Similarly, the `managed_by()` operation can be used to locate the manager for a given low level resource. Finally, `release()` should be called when the (abstract) resource is no longer needed.

Different styles of manager can be created. For example, some managers may multiplex high level resources on to the set of low level resources (e.g. schedulers), or they can map directly on to low level resources (e.g. memory allocation), with the set of available resources either being static or dynamic. Similarly, they can compose low level resources to create a more abstract high level resource (e.g. to create a Unix-like process), or they can provide direct access to a low level resource (cf binders).

In general, managers offer the following interface:

```
interface Manager {
 void register(Interface ref,...);
 void unregister(Interface ref);
 AbstractResource[] manages(AbstractResource high);
 AbstractResource[] provides();
}
```

The `register()` and `unregister()` operations allow a programmer to associate additional low level resources with the manager. The precise interpretation of this will vary between different styles of manager as seen when we consider factories and binders below. The `manages()` and `provides()` operations return the high level and low level abstract resources associated with this manager respectively. The `high` parameter to `manages()` enables the manager to return only the low level resources supporting that given high level resource.

Note that this approach treats resource as a relative term. An object *r* is a resource *for* another object *u* - a *user* -, that itself may be a resource for a third object *v*, or possibly for *r*. In other words, resource and user are roles, similar to server and client.

3.1.2. Factories and Binders

Overview

Suppose now that an object requires access to an abstract resource. There are two ways in which this can happen. Firstly, it can create a new instance of the required abstract resource. Secondly, it can locate and bind to an existing instance of an abstract resource. These two scenarios are supported in the framework by *factories* and *binders* respectively.

As will be seen, both factories and binders are specialisations of `Manager` in the architecture in that they both have the role of providing an `AbstractResource` for an object (the `Manager()` class described above should be viewed as an abstract class, which is realised by factories, binders, or indeed other styles of manager). In this way, they can be viewed as different *patterns* of object management.

We look at each in turn below.

Factories

As stated above, factories support the creation of new objects of type `AbstractResource`. In our architecture, they also have a second role, i.e. performing the run-time management of the resultant resource(s). Combining the two roles has a number of advantages, e.g. in integrating policies for static and dynamic resource management.

A factory must conform to the following interface:

```
interface Factory extends Manager {
 AbstractResource new(...);
}
```

The `new()` method returns a reference to an interface of an abstract resource. Its arguments may contain information needed by the factory to build and manage the new resource, like specifications of the type and quality of service of the expected resource, indications of lower level resources or factories to be used, etc. The `register()` operation defined in the super class effectively provides the factory with additional raw material to create the new high level abstract resource, e.g. in terms of providing a new virtual processor to support threads. The `unregister()` operation has the opposite effect of removing the low level abstract resource from the factory.

In a typical environment, an extensible range of factories will be provided corresponding to the various levels of abstract resources discussed in section 3.1.1.

Binders

As stated earlier (section 2.1), an object is only aware of other objects through knowledge of interface references. However, holding an interface reference does not mean that it is possible for the holder of the reference to interact with the corresponding object. The object may be inaccessible or unusable in its current form.

The role of a binder object is to make identified interfaces accessible to their user. More specifically, in the resources framework, they support the location of appropriate resources and also undertake the necessary steps to make the interface accessible. For example, if an interface is located on a remote machine, the binder will establish an appropriate communications path to the object or indeed migrate the remote object to the local site.

The interface for a binder object is shown below:

```
interface Binder extends Manager {
 AbstractResource bind(Interface ref,...);
}
```

To be managed by a binder, an interface must register itself using the inherited `register()` method. In this context, this is similar to registering an interface with a name server or trading service. This means that the binder is now entitled to let users access and interact with that interface. The binder may also take steps to make this resource accessible. The precise policies used for allocation may be implicit in the binder, encapsulated in the provided interface reference, or made explicit in parameters to the `register()` method. Additional parameters may also specify management policies (cf the policies provided in CORBA in the Portable Object Adapter specification). The corresponding `unregister()` method then removes this abstract resource from control of that binder.

The role of the `bind()` method is to allow the specified interface to be used. This method returns an interface reference that should then be used to interact with that object. This is likely to be an interface reference for the object itself. The architecture however also allows this to refer to an intermediary interface that will provide indirect access to the appropriate object (cf proxy mechanisms).

3.2. Application to Processing Resources

3.2.1. A Resource Framework for Processing

To illustrate the use of the resource management framework, we now discuss a particular instantiation of the framework to support the management of processing resources. As an example, we present a two-level structure whereby lightweight threads are constructed using virtual processors, which are themselves built on top of the underlying processor(s). The corresponding structure is illustrated in figure 2.

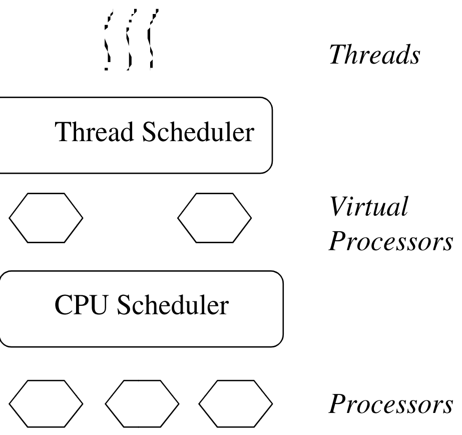

Fig. 2. An Example Scheduling Framework

From this diagram, it can be seen that *schedulers* fulfil the role of managers (more specifically factories). The CPU scheduler provides the abstraction of virtual processors out of the underlying physical processors. Similarly, the thread scheduler creates the abstraction of threads and maps them on to the underlying virtual processor(s). In both cases, the scheduling policy is encapsulated within each of the schedulers but may be changed dynamically (see below).

Processors, virtual processors and lightweight threads can all be viewed as a particular type of abstract resource that we refer to as a *machine*. In general terms, machines consume messages, act on these messages (possibly changing the state of the corresponding object), and then emit further messages. We also introduce a specialisation of a machine, called a *job*, which can be configured with scheduling parameters.

3.2.2. The Major Objects Revisited

Machines and Jobs

As mentioned above, machines are abstract resources that are responsible for carrying out some activity. They may be processors or sets of processors, or may represent more abstract execution environments such as virtual processors or indeed interpreters for high level languages. They may also be sequential or parallel.

Machines offer the following interface:

```
interface Machine {
 void run(Message);
}
```

The `run()` method provides a target machine with a new message (assuming the machine is in an appropriate state to accept the message). A `Job` is then an abstract machine that uses lower-level machines controlled by the scheduler to execute its messages. It offers the following interface:

```
interface Job extends Machine, AbstractResource {
 SchedParams getSchedParams();
 void setSchedParams(SchedParams params);
}
```

The two methods, `getSchedParams()` and `setSchedParams()` can be used to inspect the current scheduling parameters used by the underlying scheduler and to change these parameters dynamically.

Schedulers

The precise definition of a scheduler is that it is a machine factory that is able to create abstract machines (jobs) and then multiplex them on to lower-level machines. Its interface is as follows:

```
interface Scheduler extends Factory {
 Job new(SchedParams params);
 void notify(Event event);
 void suspend(...);
 void resume(...);
}
```

As can be seen, the interface supports a `new()` method that returns an object of type `job`. The `params` argument represents the initial *scheduling parameters* that will be used by the scheduler to control the multiplexing. These may be expressed in terms of priorities, deadlines, periods, etc. The scheduler implements a function to decide at any moment how jobs should be allocated to the various machines under its control[4] (according to these scheduling parameters). For instance, threads may be understood as jobs offered by a scheduler; the scheduler will then allocate the processor(s) (the lower-level machine(s)) to them according to their priority (the scheduling parameter). The scheduler may also use some dynamic information, like the timestamp on specific events, to control the multiplexing. Obviously, if there are fewer machines than jobs, the scheduler must maintain a job queue.

The decisions taken by a scheduler to multiplex jobs on machines may be triggered by different events such as interruptions or timer notifications. More generally, an event is an asynchronous occurrence in the system (message reception, message emission, object creation, etc).

The remaining methods defined on `Scheduler` provide a level of event management. The `notify()` method informs the scheduler of the occurrence of a specific event. The `suspend()` method then specifies that the execution of one or more jobs, as given by the arguments of the call, should be stopped and blocked. Finally, the `resume()` method is called to make blocked jobs eligible for re-

[4] Note that, in our framework, we enforce a restriction that a machine can be controlled by one and only one scheduler, this being the sole entity allowed to call the `run()` method. In contrast, a single scheduler may manage and use several machines.

execution. Parameters may include events that should trigger the suspend/resume mechanism. Locks may also be provided as specific instantiations of these operations.

3.3. Incorporation into Open ORB

The resource management framework is incorporated into the Open ORB architecture as an additional meta-model, referred to as the *resources meta-model*. The complete Open ORB architecture therefore consists of four meta-models as shown in figure 3.

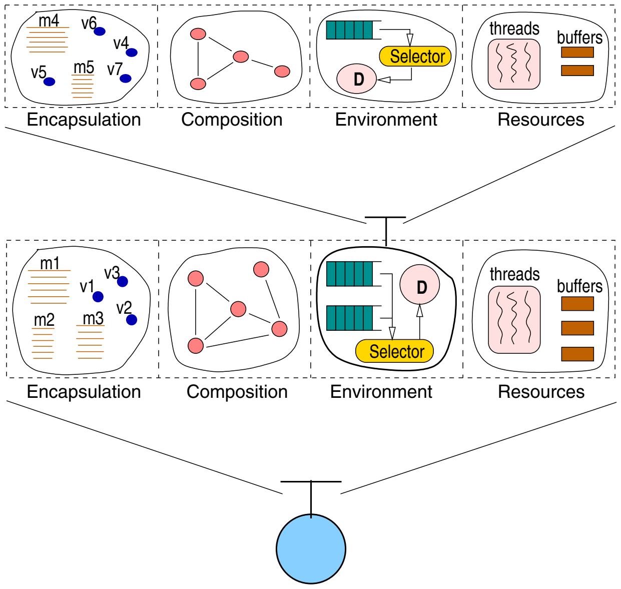

Fig. 3. The Open-ORB Reflective Architecture

It is important to stress that this meta-model is orthogonal to the others. For example, the environmental meta-model identifies the steps involved in processing an arriving message; the resources meta-model then identifies the resources required to perform this processing. More generally, the resources meta-model is concerned with the allocation and management of resources associated with any activity in the system. There is an arbitrary mapping from activity to resources. For example, all objects in a configuration could share a single pool of resources. Alternatively, each object could have its own private resources. All other combinations are also possible including objects partitioning their work across multiple resource pools and resources spanning multiple objects.

As an example, consider a configuration of objects providing an environmental model. One object could deal with the arrival and enqueuing of all messages from the network. This object could have a single high-priority thread and a dedicated pool of buffers associated with it. A further set of objects could deal with the selection, unmarshalling and dispatching of the message. In this case, the various objects could all share a pool of threads and a pool of buffers. Note that, once allocated to deal with a message, the thread (and associated buffer) will complete the entire activity, spanning multiple object boundaries. Synchronisation with other threads (and buffers) is only required when switching between abstract resources, e.g. when handing over from the arrival activity to the selection activity.

The resources meta-model supports the reification of the various objects in the resource framework. More specifically, the meta-model provides access to the (one or more) top-level abstract resources associated with a given interface. For example, a given interface might have a single abstract resource containing a global pool of threads and a dedicated buffer pool. It is then possible to traverse the management structure from the abstract resource(s). In particular, we can locate the manager for a given abstract resource by following a *provided by* link (see section 3.1.1). In turn, each manager maintains a list of resources that it *manages*, providing access to abstract resources one level down. This process can then be applied recursively to reach any particular object of the management structure. Similarly, from a resource, it is possible to follow a *managed by* link to locate the appropriate manager. It is then possible to trace the abstract resource that this manager *provides* (see figure 1).

Using such traversal mechanisms, it is possible to inspect and adapt the management structure associated with such abstract resources. We distinguish between fine-grained adaptation whereby the management structure remains intact and coarse-grained adaptation that makes changes to this structure. Examples of both styles of adaptation are given below.

4. Explanatory Examples

4.1. Introspection and Fine-grained Adaptation

This example demonstrates introspection and fine-grained adaptation for processing resources. We consider a single object, offering a single interface `iref`. This object is supported by a set of abstract resources that are bundled into a single top level abstract resource. In particular, this bundle contains a *team of threads* and some associated *virtual memory*. We assume a two-level management structure for threads as defined in section 3.2 above. We note that the threads are, in general, under-performing and decide to alter the priority of the underlying virtual processor (we assume that all threads in the team map on to one virtual processor).

This level of adaptation is achieved as follows:

```
top  = iref.Resources()[BUNDLE];
team = top.provided_by().manages(top)[THREADS];
vp   = team.provided_by().manages(top)[VP];
vp.setSchedParams (new_params);
```

The first line enters the resources meta-model, using a Resources() method as defined on all (reflective) objects. Note that this operation returns an array of all the top level abstract resources associated with the interface. We thus provide an index to select the appropriate resource (in this case the bundle, denoted by BUNDLE). From this, we can access the underlying team of threads and then the underlying virtual processor by using the provided_by() and manages() links. Note that we must also provide an index to select the appropriate low level resource from the array returned by the manages() operation. From there, it is straightforward to change the scheduling parameters on the appropriate virtual processor.

4.2. Coarse-grained Adaptation

In this example, we assume the same top-level abstract resource as before. This time, however, we want to change virtual memory to persistent virtual memory. This implies some fundamental changes to the management structure. In addition, this example raises important issues of consistency, i.e. the new persistent memory should mirror the original virtual memory and the change should be atomic.

We assume that virtual memory is provided by a single resource manager that maps to both the underlying physical memory and secondary storage resources. We also assume that we have access to a factory, denoted by `pvm_factory`, which can be used to create a new segment of persistent virtual memory.

The first few steps are similar to the example above:

```
top  = iref.Resources()[BUNDLE];
vm   = top.provided_by().manages(top)[VM];
team = top.provided_by().manages(top)[THREADS];
vp   = team.provided_by().manages();
```

In this case, however, we locate the virtual memory abstract resource *and* the underlying virtual processor. The latter is required as it is necessary to suspend all activity on the object while the changes are made, i.e. to make the change atomic. Note that this only works if we assume that no other threads will operate on the abstract resource. If this is not the case, then we must lock the resource instead.

We proceed as follows:

```
vp.suspend();
pvm   = pvm_factory.new(vm);
vm.release();
meta.provided_by().unregister(vm);
meta.provided_by().register(pvm);
vp.resume();
```

As can be seen, we use the state of the old virtual memory in the creation of the new persistent virtual memory (we assume that the factory supports this initialisation capability). The old virtual memory abstract resource can now be released. It then remains to alter the top level abstract resource (via its manager) to use the persistent virtual memory rather than the previous virtual memory.

4.3. Multi-faceted Adaptation

The final example illustrates how adaptation strategies can span multiple meta-models. In particular, we consider adaptation of a continuous media binding object, referring to the compositional and resources meta-models (the former provides access to the components used in the construction of the binding, and the latter provides information on underlying resource usage). The binding object is responsible for the delivery of audio in a mobile environment, where the available bandwidth may change significantly on roaming between different network types. The object has an initial structure as shown in figure 4.

As can be seen, the configuration contains a GSM encoder and decoder with a throughput requirement of approx. 13kbits/sec. The role of the monitoring object is to report if throughput drops below this threshold. Should this happen, the compression and decompression objects must be changed to a more aggressive scheme such as LPC-10 (which only requires 2.4kbits/sec). This is only feasible, however, if there is spare processing capacity as the computational overhead of LPC-10 is significantly greater than GSM. To support this, we assume the existence of a monitoring object in the resources meta-model that collects statistics on the available processing resources.

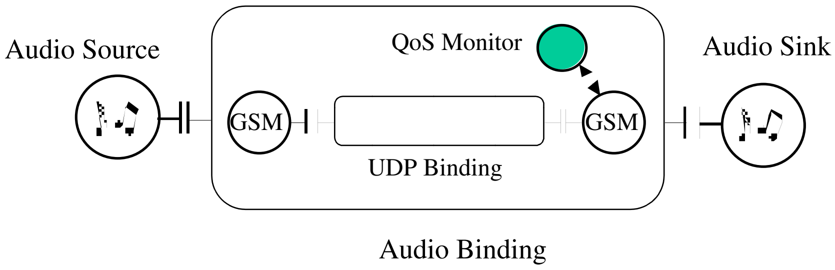

Fig. 4. The Composite Binding Object

The solution is outlined below. (As this is a more complex example, we do not go to the level of providing code.)

Firstly, it is necessary to locate the relevant monitoring objects in the compositional and resource meta-spaces. On detecting a drop in available bandwidth, it is then necessary to check if there is spare processing capacity. If there is, then adaptation can proceed (if not, it may be possible to obtain additional processing capacity, e.g. by increasing the priority of the underlying virtual processor as described in section 4.1. above[5]).

To achieve adaptation, it is first necessary to suspend activity in the binding (as in section 4.2 above). With this being a distributed object, however, this will involve the suspension of activity in more than one abstract resource. We can then use graph manipulation operations to replace the encoder and decoder with the new LPC-10 components[6]. Following this, activity can be resumed. Note that suspension of activity may not be sufficient to ensure consistency, i.e. some GSM packets may remain in transit until after the change-over. For the sake of this example, we assume that decoders will discard invalid packets.

In general, this is an example of a *resource aware QoS management* strategy. As an extension of this strategy, we can implement adaptation policies that monitor and

[5] Strictly speaking, such a step could violate QoS contracts in other parts of the system and so should involve QoS renegotiation [ReTINA99].

[6] Again, this step could involve QoS re-negotiation.

adapt one or more meta-models. Further discussion of such multi-faceted adaptation and QoS management can be found in [Blair99].

5. Related Work

5.1. Reflective Architectures

As mentioned earlier, reflection was first introduced in the programming language community, with the goal of introducing more openness and flexibility into language designs [Kiczales91, Agha91, Watanabe88]. More recently, the work in this area has diversified with reflection now being applied to areas such as operating system design [Yokote92], atomicity [Stroud95] and atomic transactions [Barga96], windowing systems [Rao91], CSCW [Dourish95]. There is also growing interest in the use of reflection in distributed systems. Pioneering work in this area was carried out by McAffer [McAffer96]. With respect to middleware, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [Singhai97]. The level of reflection however is coarse-grained and restricted to invocation, marshalling and dispatching. In addition, the work does not consider the reification of resource management. Researchers at APM have developed an experimental middleware platform called FlexiNet [Hayton97]. This platform allows the programmer to tailor the underlying communications infrastructure by inserting/ removing layers. Their solution is, however, language-specific, i.e. applications must be written in Java. Manola has carried out work in the design of a "RISC" object model for distributed computing [Manola93], i.e. a minimal object model which can be specialised through reflection. Researchers at the Ecole des Mines de Nante are also investigating the use of reflection in proxy mechanisms for ORBs [Ledoux97].

Our design has been influenced by a number of specific reflective languages. As stated above, the concept of multi-models was derived from AL/1-D. The underlying models of AL/1-D are however quite different; the language supports six models, namely operation, resource, statistics, migration, distributed environment and system [Okamura92]. Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [Watanabe88] and CodA [McAffer96]. Both these systems feature decompositions of meta-space in terms of the acceptance of messages, placing the message in a queue, their selection, and the subsequent dispatching. Finally, the design of ABCL/R2 includes the concept of groups [Matsuoka91]. However, groups in ABCL/R2 are more prescriptive in that they enforce a particular construction and interpretation on an object. The groups themselves are also primitive, and are not susceptible to reflective access.

5.2. Reflective Resource Management

In general, existing reflective architectures focus on structural reflection, behavioural reflection, or a combination of both. Most architectures do not address the reification of resource management. There are three notable exceptions, namely AL/1D, the ABCL/R* series of languages and Apertos. We consider each in turn below.

As mentioned above, Al/1D [Okamura92] offers a resource model as one of its six meta-models. This resource model focuses on resources on the local host and does not consider issues of distribution. AL1/D, like the other systems considered in this section, has an active object model. In terms of resources, this maps on to execution environments, called *contexts*, and memory abstractions, referred to as *segments*. These are managed by *schedulers* and *garbage collectors* respectively (with an object having precisely one of each). In terms of reflection, the resource model supports reification of contexts, segments, schedulers and garbage collectors. Consequently, this meta-model supports reification of only selected aspects of resource management; it does not attempt to provide as comprehensive a framework as we offer.

In ABCL/R2 [Matsuoka91], resource management is closely related to their concept of *groups* (see section 5.1). In their design, structural reflection is accessed through a meta-object, but behavioural reflection is controlled by a meta-group. This meta-group is implemented by a group kernel object offering a meta-object *generator* (for the creation of meta-objects), an evaluator (which acts as a sequential interpreter for the language), and a group manager (which bears the identity of the group and has reference to the meta-object generator and the evaluator). It is assumed that objects belonging to a group share a pool of common resources. Resource management policies are then contained within the various components of the group kernel object. For example, evaluators are responsible for scheduling of activity relating to the group. Such meta-behaviour can be changed dynamically. For example, as the evaluator is an object itself, the scheduling policy can be changed by inserting a call to a user-defined scheduler in the code of its meta-object. In contrast to ABCL/R2, our approach separates out the concepts of group and resource management, and also offers a more explicit resource management framework.

The successor, ABCL/R3 [Masuhara94] has a quite different resource model. In particular, the designers have added the concepts of node and node management to make the locality of computations more explicit at the meta-level (this is transparent to the base-level). A node is the meta-level representation of a processor element. Node management can then be used, for example, to control where objects are created, e.g. to introduce an element of load balancing. The language also makes node scheduling more explicit (again as a meta-level object on each node). This is an interesting development but again is less comprehensive than our proposed approach.

Finally, as a reflective operating system, one of the primary goals of Apertos is to provide open access to resource management [Yokote92]. In Apertos, every object has an associated meta-space offering an execution environment for that object (in terms of for example a particular scheduling policy, implementation of virtual memory, etc). This meta-space is accessed via a reflector object, which acts as a façade for the meta-space and offers a set of primitives for message passing, memory management, etc. Apertos adopts the computational model of active objects and pushes it to the limits. Every object managed by the system owns its computational resources, called a context), which includes, for example, memory chunks or disk blocks. This leads to prohibitive overheads in the system due to constant context switching. Our approach is more flexible with respect to active/ passive resource association, allowing high level active objects to co-exist with passive but efficient low level resources. In addition, in Apertos meta-space is structured as a set of interacting meta-objects, including resource objects and resource managers. These are not accessed directly; rather a form of coarse-grained adaptation is achieved by migrating an object from one meta-space to another (equivalent) meta-space (e.g. one that offers persistent virtual memory). The validity of migration is checked by calling a primitive defined on all reflectors called `canSpeak()`. The granularity of adaptation is therefore fixed by the class of the reflectors. In our framework, objects can gain direct access to the resources they use at the desired level of abstraction (e.g. an object may want to know at different times all the time-slices, threads or transactions executing its methods). Thus, we achieve dynamic reconfiguration at any level of expressiveness, the validity of adaptations being verified by type checking.

6. Concluding Remarks

This paper has discussed the design of next generation middleware platforms which we believe should be more configurable and re-configurable and should also provide access to the underlying resource management structure. We argue that this is best achieved through the use of reflection. More specifically, we advocate a reflective architecture where every object has its own meta-space. This meta-space is then structured as a series of orthogonal meta-models offering structural and behavioural reflection, and also reification of the resource management framework. Associated meta-object protocols support introspection and adaptation of the underlying middleware environment in a principled manner. This reflective architecture is supported by a component framework, offering a re-usable set of services for the configuration and re-configuration of meta-spaces. In general, re-configuration is achieved through reification and adaptation of object graph structures.

The main body of the paper concentrated on the resource model. The underlying resource management framework provides a uniform mechanism for accessing resources and their management at different levels of abstraction. We also presented a specialisation of this framework for activity management. We demonstrated, through a set of worked examples, that the approach can support introspection, fine- and coarse- grained adaptation, and also multi-faceted adaptation.

The work presented in this paper is supported by a number of prototype implementations. For example, the reflective middleware architecture has been implemented in the object-oriented, interpreted language Python [Watters96]. Currently, this particular implementation supports a full implementation of structural and behavioural reflection, together with an initial implementation of the resources meta-model. An earlier version of this platform (without the resources meta-model) is described in detail in [Costa98]. The resources framework is inspired by the underlying structure of Jonathan [Dumant97]. A more complete implementation of the resources framework is also being developed in Jonathan v2. A similar approach is also taken in GOPI [Coulson98].

Ongoing research is addressing two main distinct areas: i) further studies of consistency and adaptation, and ii) an examination of the support offered by the architecture for QoS-driven transaction mechanisms. This latter activity, in particular, should provide a demanding examination of the approach.

Acknowledgements

The research described in this paper is funded by CNET, France Telecom (Grant 96-1B-239). We would also like to thank a number of other people at CNET and Lancaster University who have contributed to the work described in this paper, including Anders Andersen, Lynne Blair, Mike Clarke, Tom Fitzpatrick, Lee Johnston and Katia Saikoski.

References

[Agha91] Agha, G., “The Structure and Semantics of Actor Languages”, Lecture Notes in Computer Science, Vol. 489, pp 1-59, Springer-Verlag, 1991.

[Barga96] Barga, R., Pu,, C., “Reflection on a Legacy Transaction Processing Monitor”, In Proceedings of Reflection 96, G. Kiczales (ed), pp 63-78, San Francisco; Also available from Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, P.O. Box 91000, Portland, OR 97291-1000, 1996.

[Blair97] Blair, G.S., Stefani, J.B., “Open Distributed Processing and Multimedia, Addison-Wesley, 1997.

[Blair98] Blair, G.S., Coulson, G., Robin, P., Papathomas, M., “An Architecture for Next Generation Middleware”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), pp 191-206, Springer, 1998.

[Blair99] Blair, G.S., Andersen, A., Blair, L., Coulson, G., “The Role of Reflection in Supporting Dynamic QoS Management Functions”, Internal Report MPG-99-03, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K., 199 January 1999.

[Costa98] Costa, F., Blair, G.S., Coulson, G., “Experiments with Reflective Middleware”, Proceedings of the ECOOP’98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP’98 Workshop Reader, Springer-Verlag, 1998.

[Coulson98] Coulson, G., "A Distributed Object Platform Infrastructure for Multimedia Applications", Computer Communications, Vol. 21, No. 9, pp 802-818, July 1998.

[Dourish95] Dourish, P., “Developing a Reflective Model of Collaborative Systems”, ACM Transactions on Computer Human Interaction, Vol. 2, No. 1, pp 40-63, March 1995.

[Dumant97] Dumant, B., Horn, F., Dang Tran, F., Stefani, J.B., “Jonathan: An Open Distributed Processing Environment in Java”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), Springer, September 1998.

[Hayton97] Hayten, R., “FlexiNet Open ORB Framework”, APM Technical Report 2047.01.00, APM Ltd, Poseidon House, Castle Park, Cambridge, UK, October 1997.

[Hokimoto96] Hokimoto, A., Nakajima, T., “An Approach for Constructing Mobile Applications using Service Proxies”, Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS’96), IEEE, May 1996.

[Fitzpatrick98] Fitzpatrick, T., Blair, G.S., Coulson, G., Davies N., Robin, P., “A Software Architecture for Adaptive Distributed Multimedia Systems", IEE Proceedings on Software, Vol. 145, No. 5, pp 163-171, October 1998.

[Kiczales91] Kiczales, G., des Rivières, J., and Bobrow, D.G., “The Art of the Metaobject Protocol”, MIT Press, 1991.

[Ledoux97] Ledoux, T., “Implementing Proxy Objects in a Reflective ORB”, Proc. ECOOP’97 Workshop on CORBA: Implementation, Use and Evaluation, Jyväskylä, Finland, June 1997.

[Manola93] Manola, F., "MetaObject Protocol Concepts for a "RISC" Object Model", Technical Report TR-0244-12-93-165, GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA, December 1993.

[Masuhara94] Masuhara, H., Matsuoka, S., Yonezawa, A., “An Object-Oriented Concurrent Reflective Language for Dynamic Resource Management in Highly Parallel Computing”, IPSJ SIG Notes, Vol. 94-PRG-18 (SWoPP'94), pp. 57-64, July 1994.

[Matsuoka91] Matsuoka, S., Watanabe, T., and Yonezawa, A., “Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming”, In Proceedings of the European Conference on Object-Oriented Programming (ECOOP’91), Geneva, Switzerland, LNCS 512, pp 231-250, Springer-Verlag, 1991.

[McAffer96] McAffer, J., “Meta-Level Architecture Support for Distributed Objects”, In Proceedings of Reflection 96, G. Kiczales (ed), pp 39-62, San Francisco; Available from Dept of Information Science, Tokyo University, 1996.

[Mitchell98] Mitchell, S., Naguib, H., Coulouris, G., Kindberg, T., “Dynamically Reconfiguring Multimedia Components: A Model-based Approach”, Proc. 8th ACM SIGOPS European Workshop, Lisbon, Sep. 1998.

[Okamura92] Okamura, H., Ishikawa, Y., Tokoro, M., “AL-1/d: A Distributed Programming System with Multi-Model Reflection Framework”, Proceedings of the Workshop on New Models for Software Architecture, November 1992.

[Rao91] Rao, R., “Implementational Reflection in Silica”, Proceedings of ECOOP’91, Lecture Notes in Computer Science, P. America (Ed), pp 251-267, Springer-Verlag, 1991.

[ReTINA99] ReTINA, “Extended DPE Resource Control Framework Specifications”, ReTINA Deliverable AC048/D1.01xtn, ACTS Project AC048, January 1999.

[Schmidt97] Schmidt, D.C., Bector, R., Levine, D.L., Mungee, S., Parulkar, G., “Tao: A Middleware Framework for Real-time ORB Endsystems”, IEEE Workshop on Middleware for Real-time Systems and Services, San Francisco, Ca, December 1997.

[Singhai97] Singhai, A., Sane, A., Campbell, R., “Reflective ORBs: Supporting Robust, Time-critical Distribution”, Proc. ECOOP’97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems, Jyväskylä, Finland, June 1997.

[Smith82] Smith, B.C., “Procedural Reflection in Programming Languages”, PhD Thesis, MIT, Available as MIT Laboratory of Computer Science Technical Report 272, Cambridge, Mass., 1982.

[Stroud95] Stroud, R.J., Wu, Z., “Using Metaobject Protocols to Implement Atomic Data Objects”, Proceedings of the European Conference on Object-Oriented Programming (ECOOP’95), pp 168-189, Aarhus, Denmark, August 1995.

[Vanegas98] Vanegas, R., Zinky, J., Loyall, J., Karr, D., Schantz, R., Bakken, D., “QuO’s Runtime Support for Quality of Service in Distributed Objects”, Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware’98), pp 207-222, Springer, 1998.

[Watanabe88] Watanabe, T., Yonezawa, A., “Reflection in an Object-Oriented Concurrent Language”, In Proceedings of OOPSLA’88, Vol. 23 of ACM SIGPLAN Notices, pp 306-315, ACM Press, 1988; Also available as Chapter 3 of “Object-Oriented Concurrent Programming”, A. Yonezawa, M. Tokoro (eds), pp 45-70, MIT Press, 1987.

[Watters96] Watters, A., van Rossum, G., Ahlstrom, J., "Internet Programming with Python", Henry Holt (MIS/M&T Books), September 1996.

[Yokote92] Yokote, Y., “The Apertos Reflective Operating System: The Concept and Its Implementation”, In Proceedings of OOPSLA’92, ACM SIGPLAN Notices, Vol. 28, pp 414-434, ACM Press, 1992.