A Resource Management Framework for Adaptive Middleware

Hector A. Duran  
Computing Department, Lancaster University,  
Bailrigg, Lancaster LA1 4YR, UK  
duranlim@comp.lancs.ac.uk

Gordon S. Blair  
Computing Department, Lancaster University,  
Bailrigg, Lancaster LA1 4YR, UK  
gordon@comp.lancs.ac.uk

Abstract

In this paper, we introduce a reflective resource management framework that offers facilities for resource awareness and dynamic reallocation of resources for an adaptive middleware platform.

1. Introduction

Middleware has emerged as a means of solving problems of heterogeneity (e.g. heterogeneous hardware, operating systems, programming languages, etc) in distributed environments. One of the main functions of middleware is to provide a platform independent programming model. In these systems a common programming interface is achieved regardless of the hardware, the operating system and the language the application is relying on [8, 10].

The advent of new technologies like the Internet, multimedia applications and mobile computing is imposing new requirements on such middleware platforms. Traffic on the network may also lead to difficulties for real-time applications. In addition, mobile computing implies the possibility of changing from having an excess of resources to a more hostile environment; such a situation may lead an application to be disrupted. Therefore, there is a need to include adaptation in the middleware so that it is able to deal with the above demanding issues.

Resource management plays an important role in this adaptation process in terms of both resource awareness and dynamic reallocation of resources. That is, adaptive middleware platforms should provide facilities that allow a system to be aware of the availability of computational resources, the management policies being used and how resources are allocated among the activities the system performs.

We believe that a general solution to these problems is to introduce more openness and flexibility into middleware platforms. In particular, we use reflection [11, 5] as a principled means of achieving these goals (see next section).

In this paper, we introduce an architecture for reflective middleware with introspection and adaptation capabilities. The main emphasis of the paper is the design and implementation of a resource management framework for such an architecture. Note that we are not carrying out research in specific resource management algorithms; rather, we focus on providing a general framework which can accommodate best practice in the area. Such facilities ease the management of resources by providing a complete and consistent model of resources in the system at varying levels of abstractions. In addition, activities in the system are represented in a task model which provides a higher level of resource management.

The paper is structured as follows. Section 2 presents an overview of reflection. Section 3 introduces the overall approach of our reflective middleware platform. Section 4 then focuses on the description of the resource model. Following this, section 5 describes the implementation of a specific instantiation of the framework. Finally, some concluding remarks are presented in section 6.

2. Background on Reflection

Reflection is a means by which a system is able to inspect and change its internals in a principled way [6]. Basically, a reflective system is able to perform both self-inspection and self-adaptation. To accomplish this, a reflective system has a representation of itself. This representation is causally connected to its domain, i.e. any change in the domain must have an effect in the system, and vice versa.

A reflective system is mainly divided into two parts: the base-level and the meta-level. The former deals with the normal aspects of the system whereas the latter regards the system’s representation. The meta-level interface is often referred to as the Meta-Object Protocol (MOP) [5]. In addition, the process of gaining access to the meta-level is called reification. Reification is the means of making explicit an aspect of the system whereas absorption is the opposite, i.e. the process in which an explicit aspect of the
system is made implicit. There are two styles of reflection [6]: procedural and declarative. In the former, the representation of the system is a program that it is generally written in the same language of the system. The latter uses a set of declarative statements as a representation of a system.

Most research has been focused on object-oriented languages (OOL) [6, 12, 5, 7]. The meta-object of an object can be seen as a virtual machine since it interprets and executes the object. In addition, since meta-objects are also objects, they may have meta-objects as well. This opens the possibility of an infinite tower of reflection to exist in theory.

3. Our System Architecture

3.1 Overall Approach

The reflective middleware architecture is underpinned by four general principles. First, we adopt the use of the RM-ODP computational model. Objects within this model have several interfaces. Not only operational interfaces but also stream and signal interfaces are supported. In addition, explicit bindings are supported which offer more control over the communication path between object interfaces.

Second, we advocate the use of procedural reflection. Our choice is based on the number of advantages procedural reflection offers over its counterpart (i.e. declarative reflection) [1].

The third principle refers to the fact that meta-spaces support objects on a per-object (or per interface) basis. The advantages provided by this approach are twofold. A fine level of control can be obtained and changes to the meta-level are localised. The latter enforces consistency since modifications to the meta-level are not spread beyond the limits of an object interface.

Finally, the meta-space is structured as a set of four orthogonal meta-models (compositional, environmental, encapsulation and resources). This approach was first introduced by AL-1/D [7] in order to simplify the meta-interface by maintaining a clear separation of concerns among different system aspects. Each meta-model is introduced in turn below.

3.2 The Four Meta-models

The compositional meta-model deals with the way a composite object is composed, i.e. how its components are inter-connected, and how these connections are manipulated. There is a compositional meta-model per object.

The environmental meta-model is in charge of the environmental computation of an object interface, i.e. how the computation is done. It deals with message arrivals, message selection, dispatching, marshalling, concurrency control, etc.

The encapsulation meta-model relates to the set of methods and associated attributes of a particular object interface. Methods scripts can be inspected and changed by accessing this meta-model.

Finally, and of central importance to this paper, the resource meta-model is concerned with both the resource awareness and resource management of objects in the platform. The main elements of this meta-model are resources, resource managers and resource factories. A more comprehensive description of the resource meta-model is presented below.

The complete architecture is summarised in figure 1. Further details of the overall architecture can be found in the literature. For example, detailed descriptions of the four meta-models can be found in [3, 2].

4. The Resource Meta-model

4.1 Modelling Resources

System resources are explicitly represented in the resource model [9, 2]. Resources are represented as objects and may be accessed through their interfaces. Objects representing resources are called abstract resources. The resource model is recursive in that high-level abstract resources are constructed on top of lower-level resources. The result of this approach is a hierarchy of resource abstractions.

![Figure 1. The Four Meta-model Structure.](image)

![Figure 2. The Overall Resource Management Structure.](image)
The overall management structure is shown in figure 2. As can be seen, *managers* provide high level resources out of lower level resources. A manager allows its users to share the resources it manages by providing an abstract view of them. For this purpose, managers are responsible for either multiplexing or mapping higher-level resources on to lower-level resources.

*Factories* are then in charge of creating new instances of abstract resources. The main role of factories is to create higher-level resources as abstractions of lower-level resources. To accomplish this, higher-level factories use lower-level factories. Therefore, a corresponding factory hierarchy is also defined. *Schedulers* are the specialisation of managers which are in charge of multiplexing higher-level processing resources, like threads or virtual processors, on lower-level processing resources.

Finally, we introduce *Virtual Task Machines (VTM)* as the top-level abstractions. VTMs represent units of resource management for the activities performed in a system. Basically, a VTM encompasses both an abstraction of the resources allocated to a particular task in the system (e.g. a team of threads and buffer memory) and information about the management policies of such resources. A more comprehensive description of the role of a VTM, is given in section 4.2.

![Figure 3. A Particular Instantiation of the Resource Meta-model.](image)

As an example, consider a particular instantiation of the resource framework shown in figure 3. At the top of the abstract resource hierarchy are VTMs which are built on top of abstract resources representing both CPU and memory resources. Firstly, a two-level thread structure is represented in figure 3(a) where user-level threads run on top of *virtual processors* (kernel-level threads). We use a *team* abstraction to represent two or more user-level threads within a VTM. At the bottom of the hierarchy we have a representation of a physical processor. Secondly, a VTM has an abstract resource representing a memory buffer pool which in turn is a higher abstraction of physical memory. Similarly, there is a hierarchy of abstract resource factories as shown in figure 3(b). The VTM factory is composed by both the team factory and the memory factory whereas the virtual processor factory supports the thread factory.

### 4.2 Task-oriented Resource Management

We take a task-oriented approach for managing resources, i.e. resources are managed on a per task basis. That is, a system is broken into tasks according to the type of activities it performs. We define a *task* as a logical activity that a system performs, e.g. transmitting audio over the network or compressing a video image. Each task in the system has a representation in the resource model. Thus, a VTM represents the abstract resources that a task uses for execution. There is a one-to-one mapping between tasks and VTMs. In addition, VTMs represent a unit of resource management. VTMs may be seen as virtual machines in charge of executing their associated tasks. VTMs may span both object and address space boundaries. That is, a task may involve more than one object and may be distributed. In the latter case, a distributed VTM is a composite VTM integrated by the corresponding local VTMs.

The task-oriented resource management presented here eases the customisation and the dynamic reconfiguration of the resources of a system. The task-model provides a means to achieve both fine- and coarse-grained reconfiguration of resources. That is:

a) **Fine-grained** resource reconfiguration of the system is obtained by accessing the various levels of abstraction from which a VTM is build up. For instance, the management policy of a low-level resource may be changed, e.g. the scheduling policy of a particular thread of a task may be changed from an EDF to a least-laxity policy.

b) **Coarse-grained** resource reconfiguration of a system may be easily achieved by manipulating all resources involved in a task as single unit of resource management. For instance, a whole activity may be suspended.

An additional novel feature of the task-oriented approach is that changes to the underlying resources of a system are localised. That is, changes to the resource management system are only spread to the tasks involved in the modification operation.

### 5. Implementation

We have implemented a prototype of the resource management framework, programmed in Python and running on a Sun SPARC/Solaris 2.5 platform. The implementation also makes use of a micro ORB platform called GOPI [4]. This platform offers low level services for the support of distributed multimedia applications.
Python has been layered on top of GOPI in order to have access to such services.

We implemented a particular instantiation of the framework which includes CPU and buffer management as shown in figure 3. In particular, various levels of abstraction were used for modelling resources as depicted in figure 3(a). At the top are VTMs which run on top of teams of threads. A two-level thread model was implemented where user-level threads run on top of kernel-level threads. In addition, VTMs also include memory resources abstractions. The factory hierarchy shown in figure 3(b) was also implemented. Lower-level factories support higher-level factories.

To evaluate the platform, ongoing work is being carried out on the implementation of an audio application on top of the resource framework. In addition, we are also addressing the issue of broadening the number of computational resources managed by the system. Future work will also address the integration of the resource model with a QoS management framework.

6. Concluding Remarks

A dynamic resource management framework for an adaptive middleware platform has been presented. In this framework, computational resources have a direct representation in the system as a means of providing a more natural way of accessing resources. In addition, our framework encompasses a model that defines various levels of resource abstractions as well as hierarchies of both resource managers and resource factories. This provides a means to achieve fine-grain adaptation by allowing the user to choose the desired abstraction level of resource reconfiguration.

We have used reflection as a principled means of achieving adaptation. Through this, we have separated the non-functional behaviour (i.e. resource management) from the functional behaviour. The functional behaviour of any object in the system may be accessed through the base-level interface, whereas the non-functional behaviour may be accessed through the meta-level interface. We believe that reflection in general provides a powerful means of constructing adaptive behaviour.

Acknowledgements

Hector A. Duran would like to thank his sponsor, the Autonomous National University of Mexico (UNAM). We would like to thank our collaborators for their support. Particular thanks are due to Jean-Bernard Stefani and his group at CNET.

References


