A Principled Approach to Supporting Adaptation in Distributed Mobile Environments

Gordon S. Blair, Geoff Coulson, Anders Andersen*, Lynne Blair,
Mike Clarke, Fabio Costa, Hector Duran, Nikos Parlavantzas and Katia Saikoski
Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Bailrigg, Lancaster,
LA1 4YR, U.K.
Tel: +44 1524 65201
Fax: +44 1524 593608
gordon@comp.lancs.ac.uk

Abstract

To support multimedia applications in mobile environments, it will be necessary for applications to be aware of the underlying environmental conditions, and also to be able to adapt their behaviour and that of the underlying platform as such conditions change. Many existing distributed systems platforms support such adaptation only in a rather ad hoc manner. This paper presents a principled approach to supporting adaptation through the use of reflection. More specifically, the paper introduces a language-independent, component-based reflective architecture featuring a per-component meta-space, the use of meta-models to structure meta-space, and a consistent use of component graphs to represent composite components. The paper also reports on a quality of service management framework, providing sophisticated support for monitoring and adaptation functions. Finally, the paper describes a prototype implementation of this architecture using the object-oriented programming language Python.

1. Introduction

Future distributed systems will consist of a range of end-systems, from PDAs through to workstations, which will either be fully connected, weakly connected by low speed wireless networks such as GSM, or indeed disconnected. Furthermore, it can be anticipated that the precise level of connectivity will vary over time as a consequence of the mobility of the future computer user. Other aspects can also vary over time in such an environment including the cost of the underlying connection, the availability of resources (such as processor cycles), and the battery life. Given this, it is well recognised that mobility requires support for *adaptation*. Katz summarises this argument rather succinctly [19]:

> *"Mobility requires adaptability. By this we mean that systems must be location- and situation-aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion."*

In heterogeneous distributed environments, the other important factor is *openness*. In particular, it is crucial that open systems standards are exploited wherever possible to enable both interoperability and portability of (mobile) applications. In this paper, we are therefore concerned with support for adaptation in open distributed systems. We present a principled solution to this problem based on ideas from the fields of open implementation and reflection. We are particularly interested in supporting distributed *multimedia* applications in such environments.

The paper is structured as follows. Section 2 takes a closer look at adaptation, considering what parts of the system require adaptation, how adaptation should be carried out and where this adaptation should be placed in the overall architecture. It is argued that middleware is the right place to support adaptation. Section 3 then considers the potential role of reflection in supporting adaptation in middleware platforms. Following this, section 4 presents our architecture for reflective middleware platforms. Section 5

* Also of the University of Tromso, Norway

0-7695-0634-8/00 $10.00

then extends this architecture with quality of service (QoS) management capabilities. Our implementation of the reflective architecture (including QoS management) is then briefly discussed in section 6; this implementation uses the interpreted, object-oriented language Python. Sections 7 and 8 then evaluate the architecture and consider related work respectively. Finally, section 9 presents some concluding remarks.

2 A Closer Look at Adaptation

2.1 What do we Adapt?

Adaptation is required at various *levels* in a distributed system. For example, at the network level, it might be necessary to switch between available network links depending on the quality of service currently on offer. Higher up the protocol stack a range of adaptations are also possible such as switching to a different transport protocol (or set of micro-protocols [14]), changing the parameters of a particular protocol element, or introducing new protocol elements. It might also be useful to change the encoding of data, e.g. by introducing compression and de-compression elements. Crucially, many aspects of adaptation are also carried out either in the application or with the involvement of the application. For example, an application might choose to restructure its activity by off-loading some processing to a fixed portion of the network.

More generally, adaptation applies to a wide range of *aspects* at different levels of the system. These aspects include the communications aspects described above but also include issues such as the resources allocated to an activity and the set of external services currently being used. Furthermore, this adaptation should be driven by awareness of a wide range of issues including communications performance, resource usage, location, cost, battery life and application preference.

2.2 How do we Adapt?

Adaptation as described above (if it is not to be done on a purely ad hoc basis) requires a sophisticated infrastructure to expose and manage the various aspects described above. In more detail, this requires:

- Open access to various components in the underlying system, corresponding to the various aspects under consideration;
- The ability to monitor the current environment (communications subsystem, resources, battery life, etc) and adapt the various components and configurations of components to suit the characteristics of the current environment.

Open access is the cornerstone of adaptive systems. However, existing distributed systems platforms often fail to provide this level of openness, or, if they do provide openness, it tends to be in an ad hoc manner. As will be seen in section 4, our approach offers a principled means of achieving openness tackle through the application of reflection.

The second requirement implies dynamic quality of service (QoS) management and should be addressed by a general and sophisticated QoS management architecture subsuming QoS functions such as monitoring and adaptation.

2.3 Where do we Adapt?

As stated above, adaptation is required at all levels of a system, from the application level potentially right down to the operating system level. However, this immediately introduces a number of problems. For example, adaptation at the operating system level can be quite dangerous in terms of affecting integrity and performance. In addition, the programmer would inevitably have to rely on operating system specifics to achieve adaptation, thus compromising the portability of applications. The opposite extreme of leaving all adaptation to the application is also clearly unacceptable, as this would introduce an unacceptable burden for the application writer. Our solution is to introduce a framework for managing adaptation at the middleware level of the system.

In distributed systems, middleware is defined as a layer of software residing on every machine and sitting between the underlying (heterogeneous) operating system platforms and distributed applications/services, offering a platform-independent programming model to programmers (see figure 1). Examples of middleware platforms include OMG CORBA, Microsoft's DCOM, and the Open Group's DCE [4].

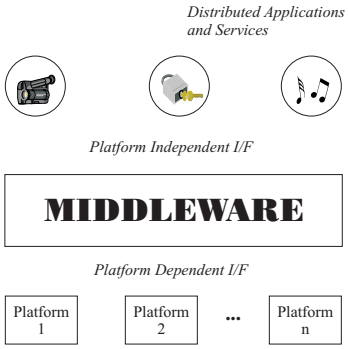

Figure 1. The role of middleware.

Note that this approach is supported by recent research in the operating systems community which advocates that operating systems should be smaller and more policy free [12]. This implies that many functions initially contained in the

operating system kernel will be available as open services in user space accessible through a middleware platform. Similar results have also emerged in the communications community, most notably with the emergence of application level framing [6]. Again, this implies that key functions such as transport protocols may be available as open services. The role of the middleware platform is then to support the configuration (and possibly re-configuration) of such services on behalf of the application.

3 The Role of Reflection

The concept of reflection was first introduced by Smith in 1982 [35]. In this work, he introduced the reflection hypothesis which states:

> *"In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures"*.

The importance of this statement is that a program can access, reason about and alter its own interpretation. Access to the interpreter is provided through a *meta-object protocol* (MOP) which defines the services available at the *meta-level*. Examples of operations available at the meta-level include altering the semantics of message passing and inserting before or after actions around method invocations. Access to the meta-level is provided through a process of *reification*. Reification effectively makes some aspect of the internal representation explicit and hence accessible from the program. The opposite process is then *absorption* where some aspect of meta-system is altered or overridden.

Smith's insight has catalysed a large body of research in the application of reflection. Initially, this work was restricted to the field of programming language design [20, 37, 1]. More recently, the work has diversified with reflection being applied in operating systems [39] and, more recently, distributed systems (see section 8).

The primary motivation of a reflective language or system is to provide a principled (as opposed to ad hoc) means of achieving open access. For example, reflection can be used to inspect the internal behaviour of a language or system. By exposing the underlying implementation, it becomes straightforward to insert additional behaviour to monitor the implementation, e.g. performance monitors, quality of service monitors, or accounting systems. Reflection can also be used *to adapt* the internal behaviour of a language or system. Examples include replacing the implementation of message passing to operate more optimally over a wireless link, introducing an additional level of distribution transparency in a running computation (such as migration transparency), or inserting a filter component to reduce the bandwidth requirements of a communications stream. Although reflection is a promising technique, there are a number of potential drawbacks of this approach; in particular, issues of *performance and integrity* must be carefully addressed (we return to these issues in section 7).

In contrast, the standard approach to developing middleware platforms is generally to adopt a *black box* philosophy, whereby implementation details are hidden from the platform user (cf. distribution transparency). There is increasing evidence though that the black box philosophy is becoming untenable. For example, the OMG have recently added internal interfaces to CORBA to support services such as transactions and security. The recently defined Portable Object Adapter is another attempt to introduce more openness in their design. Nevertheless, their overall approach can be criticised for being rather ad hoc (as discussed above). Similarly, a number of ORB vendors have felt obliged to expose selected aspects of the underlying system (e.g. filters in Orbix or interceptors in COOL). These are however non-standard and hence compromise the portability of CORBA applications and services.

The authors believe that the solution is to provide flexible middleware platforms through application of the principle of reflection.

4 An Architecture for Reflective Middleware

4.1 General Principles

In our reflective architecture, we adopt a *component-based* model of computation [36]. A middleware platform is then viewed as a particular configuration of components, which can be selected at build-time and re-configured at run-time. We therefore provide an open and extensible library of components, and component factories, supporting the construction of such platforms, e.g. protocol components, schedulers, etc. The use of components is important given the trend towards the application of this technology in open distributed processing, e.g. CORBA v3 [30] and Microsoft's DCOM. Note however that these technologies exploit component technology at the application level; we extend this approach to the structuring of the middleware platform itself. Our particular component model includes features to support multimedia applications, and is derived from previous work on the Computational Model from RM-ODP. The main features of our component model are: i) components are described in terms of a set of *required* and *provided* interfaces, ii) interfaces for *continuous media* in-

teraction are supported, iii) *explicit bindings* can be created between compatible interfaces (the result being the creation of a *binding component*), and iv) components offer a built-in *event notification service*. The component model also has a sophisticated model of quality of service including QoS annotation on interfaces. Further details on the underlying RM-ODP Computational Model can be found in [4]. In contrast, with RM-ODP, however, we adopt a consistent computational model throughout the design.

A second principle behind our design is to support *per interface* (or sometimes per component) *meta-spaces*. This is necessary in a heterogeneous environment where components 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 change. We look at the design of meta-spaces in detail in the following sub-section.

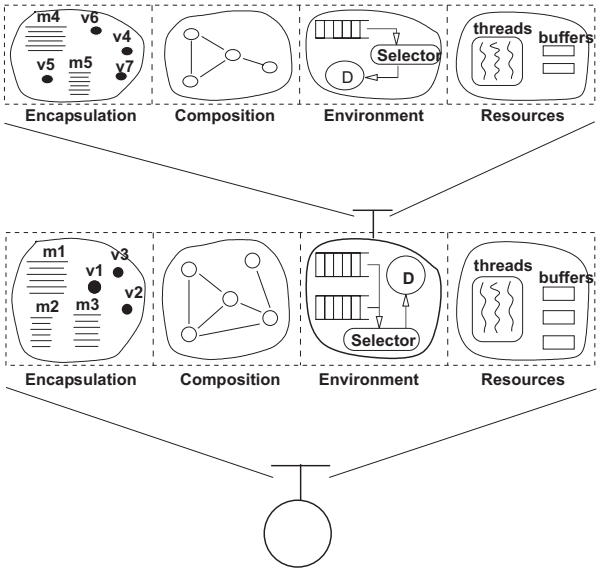

Figure 2. Overall structure of meta-space.

4.2 A Multi-model Approach

In our design, every component has an associated *meta-space* supporting inspection and adaptation of the underlying infrastructure for the component. More precisely, because of the nature of our component model, a meta-space is actually associated with each interface. Crucially, this meta-space is organised as a number of closely related but distinct *meta-space models*. This approach was first advocated by the designers of AL-1/D [31]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects. The four aspects currently employed are: *composition*, *encapsulation*, *environment* and *resource* (see figure 2).

We consider each model briefly in turn below. Further details of this reflective architecture can be found in the literature [3, 7], including detailed descriptions of the meta-object protocols (MOP) offered by each of the meta-models [7].

Firstly, the *composition meta-model* provides access to the component in terms of its constituent (base-level) components, represented as a *component graph*, in which the constituent components are connected together by efficient primitive bindings referred to as *local bindings*. This meta-model is particularly useful when dealing with binding components [13]. In this context, the composition meta-model reifies the internal structure of the binding component in terms of the components used to realise the end-to-end communication path. For example the component graph could feature an MPEG compressor and decompressor and an RTP protocol component. The structure can also be exposed recursively; for example, the composition meta-model of the RTP protocol component might expose the peer protocol entities for RTP and also the underlying UDP/IP protocol.

Secondly, the *environment meta-model* represents the execution environment for each interface as traditionally provided by the middleware platform. In a distributed environment, this corresponds to functions such as message arrival, enqueing, selection, unmarshalling and dispatching (plus the equivalent on the sending side) [37, 27]. As with the composition meta-model, this activity is represented as a component graph.

Thirdly, 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.

Finally, the *resources meta-model* provides access to the underlying resources and resource management subsystems provided by the middleware platform. For example, this meta-model can be used to discover the allocation of threads or memory to a particular task or to alter the allocation of such resources. In addition, it is possible to change the management policies for particular resources, e.g. to change the scheduling of threads from fixed priority to earliest deadline first [8].

Note that there is a high level of recursion in the above definition. In particular, the meta-level is realised using component-based techniques. Hence, components/interfaces at the meta-level are also open to reflection and have an associated meta-meta-space. As above, this meta-meta-space is represented by three (meta-meta-) models. Similarly, components/interfaces at the meta-meta-

level have an associated meta-meta-meta-space. As in ABCL/R, this is realised in our design 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 [37]. This provides a finite representation of an infinite structure.

4.3 Examples

In order to illustrate how adaptation takes place using the meta-models, we present an example scenario. Figure 3 shows a two-level binding component which provides the simple service of connecting the interfaces of two application components. The purpose of such a binding may be, for example, to transfer a continuous stream of media from one component to the other. At run-time, some external monitoring mechanism notices a drop in the network throughput, demanding a reconfiguration of the binding component in order to support the negotiated quality of service. This reconfiguration may be in terms of inserting compression and decompression filters at both sides of the binding, hence reducing the actual amount of data to be transfered.

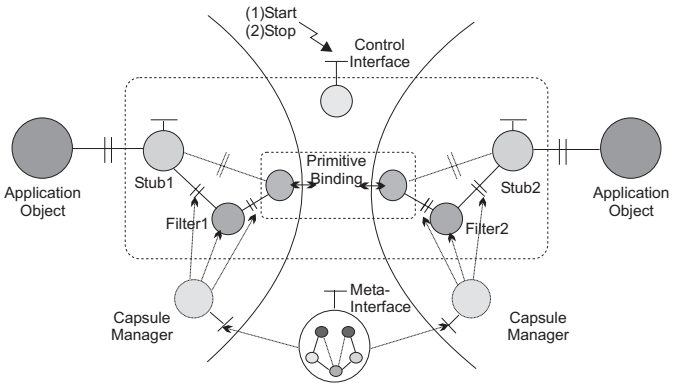

Figure 3. Adaptation using the meta-models

As the picture shows, the compositional meta-object maintains a representation of the binding configuration (the component graph mentioned in section 4.2). The compositional MOP provides operations to manipulate this representation, and any results are reflected in the actual configuration of components in the binding component.

A QoS control mechanism would call methods to add new components (`Filter1` and `Filter2` in the figure) to the binding. The effect produced by each call is:

1. the previously existing local bindings between the stub (`Stub1`) and the primitive binding is broken;
2. a new filter component is created (`Filter1`);
3. new local bindings are established to connect the interfaces of the filter to the interfaces of the stub and the primitive binding.

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

5 Introducing QoS Management

As implied in section 2, we are largely concerned with the dynamic aspects of QoS management, namely QoS monitoring and adaptation. In terms of our architecture, this equates to inspecting and adapting the corresponding component graphs depending on the actual QoS attained and the actual QoS offered by the environment (which might of course change in a mobile environment).

Dynamic QoS management is achieved by introducing *management components* into the *component* graph structure (accessed via meta-space). To maintain a clean separation of concerns between management *components and components* being managed, communication between the two is achieved by using the event notification mechanism included as part of our component model (see section 4.1). Separation is achieved because *components* do not need to know in advance if they are to be managed. In other words, managers can be introduced at any time and can then register for events of interest (and then receive call-backs when the specific events occur).

Different styles of management *component* are identified in our architecture (see table 1).

Table 1. Different styles of management component.

Monitors	**Controllers**	
	Strategy Selectors	**Strategy Activators**
Collect statistics on QoS achieved and report abnormal events to interested parties.	Select an appropriate adaptation strategy depending on feedback from monitors.	Implement a particular strategy, e.g. by manipulating an component graph.

The role of monitors is to collect statistics on the level of QoS attained by the running system and to raise events when problems occur, e.g. to collect information on the latency and throughput of a video presentation and raise exceptions should they fall outside given thresholds. Controllers are then responsible for implementing adaptation policies in response to such events. We distinguish between

strategy selectors and strategy activators (which together realise the adaptation policy). Strategy selectors decide on which approach should be taken in response to QoS degradation, e.g. degrading the quality of the video presentation or providing additional resources. In contrast, strategy activators are responsible for the detailed implementation of this strategy, typically by manipulating the component graph, e.g. video compression/ decompression components might be introduced. The rationale for this division is to provide a cleaner architecture and to promote re-use of management components. In addition, selectors and activators can be written in different languages. It should be stressed though that, in implementation, the different functions may well be composed together.

At a first glance, this might appear to be a fairly traditional approach to QoS management. However, when combined with the capabilities of a reflective architecture, some interesting properties emerge. Firstly, the approach is completely dynamic. New management components can be introduced at any time and at any place in the underlying configuration. Similarly, they can be removed when no longer needed. Both these actions are initiated by reconfiguring the component graph. Secondly, the QoS management functions can operate over any of the meta-spaces. For example, it is possible to monitor both the composition of an open binding and also its associated resource usage and hence make adaptation decisions based on resource availability. Similarly, it is possible to carry out adaptations both in terms of manipulating the component graph of an open binding and also in terms of allocating additional resources. Thirdly, the policy for management is itself open to inspection and adaptation through reification of management components. To enable this, we assume that management components consist of a policy written in an appropriate scripting language, together with an in-built interpreter for that scripting language. Reflection can then be used to access or modify this policy, e.g. by downloading a new management script or altering some parameters for the script. More specifically, we can introduce meta-managers (consisting of monitors and controllers) to effectively manage the management structure. Whereas managers implement policy, meta-managers implement meta-policy. As an example, consider the use of header compression in a low bandwidth environment. A manager could have the role of switching header compression on or off based on a given bandwidth threshold (an attribute of the manager component). A meta-manager would then be able to alter this threshold depending on the current processor load. We argue that this is a useful facility to have in highly dynamic environments.

The architecture is open in that any scripting language can be used, although we do provide support for one particular style of language (i.e. timed automata). We believe that timed automata provide a concise and natural means of expressing QoS management policies. As an illustration, a simple timed automaton is shown in figure 4.

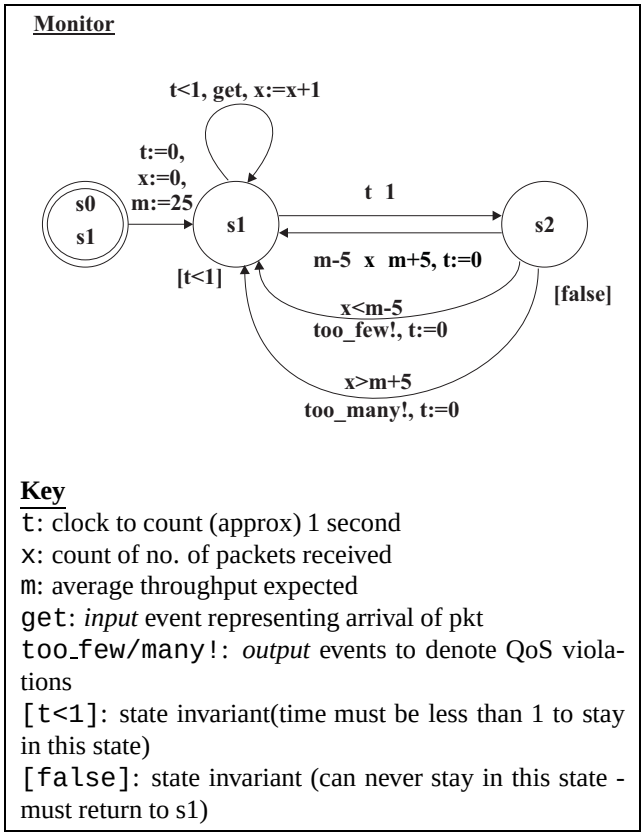

Figure 4. Monitor for throughput/quality management.

This automaton monitors a video stream for throughput (expected to be 25 frames per second). If the actual throughput, differs from this ideal by $\pm$ a given threshold (5), the quality of service of each frame may need to be increased or lowered accordingly. It is assumed in this example that a strategy selector (and subsequently an activator) will respond to the `too_few` and `too_many` events in an appropriate manner.

By using timed automata, we have the added advantage that we can formally verify QoS management subsystems. Indeed, through the use of a multi-paradigm specification technique we can specify QoS management subsystems in timed automata and the rest of the system in an alternative formal notation (or notations), and can then verify global properties of the complete system. We have developed a tool suite to support this process. Further description of this work is beyond the scope of this paper; the interested reader is referred to [5] for more details.

6 Implementation

A prototype implementation [7] of the reflective middleware architecture has been developed in Python 1.5 [38], an object-oriented interpreted language that provides several reflective features, including the ability to inspect and alter the set of methods associated with an object. The aims of this prototype are firstly, to investigate the practicality of the reflective architecture and, secondly, to investigate the (meta-) interfaces to be offered to the programmer. Performance is not a prime concern at this stage in the research (as witnessed by the choice of an interpreted language). Ongoing research is however investigating the efficient implementation of our reflective architecture using lightweight (reflective) components in a COM/ C++ environment.

The platform implements the distributed programming model described in section 4.1 above, i.e. components with multiple interfaces connected by explicit bindings. Crucially, the platform supports a set of operations to reify each of the meta-space models supported by a particular interface. These are `encapsulation()`, `composition()`, `environment()` and `resource()`. These operations can also be combined, e.g. to enter meta-meta-space. For example, `iref.environment().composition()` provides access to the component graph structure of the environmental meta-model associated with the interface `iref` (see the discussion in section 4.2). This set of operations provides language-independent access to meta-space (although as discussed above the level of access to each model may be dependent on the particular language in use).

To support QoS management, we have introduced management components into our prototype. Management components are interpreters for timed automata which interact with their environment using asynchronous events. For generality, timed automata are specified using the interoperable FC2 format [23]. They can be generated using a range of automata based tools such as Autograph and Eucalyptus. In addition, our own tool-suite supports FC2. Thus QoS management components can initially be verified using the tool suite and then down-loaded into the running system using the available reflective facilities.

To evaluate the approach, we have implemented a simple adaptive QoS management strategy for the transmission of audio over an unpredictable network (such as a wireless network). A monitor component observes the occupancy of the buffer. It then flags if it detects that the buffer appears to be “full too often” or “empty too often”. Depending on the circumstances, the corresponding controller automaton then can increase or decrease the size of the buffer, or can alter the quality of service transmitted by the source. A full description of this example can be found in the literature [2].

7 Evaluation

In our opinion, the reflective architecture described above provides a strong basis for the design of future middleware platforms to operate in mobile environments, and overcomes the inherent limitations of technologies such as CORBA (as discussed in section 3). In particular, the architecture offers principled and comprehensive access to the engineering of a middleware platform. This compares favourably with CORBA which, as stated earlier, generally follows a black box philosophy with minimal, ad hoc access to internal details. More generally, we are proposing a concept of middleware as a customisable set of components which can be tailored to the needs of an application. Furthermore, the configuration can be adapted at run-time, should the initial environmental assumptions change.

We also believe that the reflective approach generalises the viewpoints approach to structuring advocated by RM-ODP. As stated above, RM-ODP distinguishes between the Computational Viewpoint (focusing on application-level components and their interaction) and the Engineering Viewpoint (which considers their implementation in a distributed environment). Crucially, each viewpoint also has its own set of object modelling concepts (for example, the Computational Viewpoint features objects, interfaces and bindings, whereas the Engineering Viewpoint has basic engineering objects, capsules and protocol objects). Consequently, as the models are different, the mapping between the two viewpoints is not always clear. In addition, this approach enforces a two-level structure, i.e. it is not possible to analyse engineering objects in terms of their internal structure or behaviour. Our approach overcomes these limitations by offering a consistent component model throughout, supporting arbitrary levels of openness.

Another benefit of our approach is that it minimises problems of maintaining integrity. This is due to our approach to scoping whereby every component/interface has its own meta-space. Thus changes to a meta-space can only affect a single component. Furthermore, the meta-space is highly structured, again minimising the scope of changes. An additional level of safety is provided by the strongly typed component model. In contrast, the issue of performance remains a matter for further research. This issue will be resolved once we have completed our re-implementation of the reflective architecture using lightweight components.

8 Related Work

There is growing interest in the use of reflection in distributed systems. Pioneering work in this area was carried out by McAffer [27]. With respect to middleware, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) [34]. The

level of reflection however is coarse-grained and restricted to invocation, marshalling and dispatching. In addition, the work does not consider key areas such as support for groups or, more generally, bindings. Researchers at APM have developed an experimental middleware platform called FlexiNet [15]. 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 [25], 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 [21].

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 [31]. From this list, it can be seen that AL/1-D does however support a resources model. This resource model supports reification of scheduling and garbage collection of objects (but in a relatively limited way compared to our approach). Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R [37] and CodA [27]. 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 [26]. 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.

Our use of component graphs is inspired by researchers at JAIST in Japan [16]. In their system, adaptation is handled through the use of control scripts written in TCL. Although similar to our proposals, the JAIST work does not provide access to the internal details of communication objects. Furthermore, the work is not integrated into a middleware platform. Similar approaches are advocated by the designers of the VuSystem [22] and Mash [28]. The same criticisms however also apply to these designs. Microsoft's ActiveX software also uses component graphs. This software, however, does not address distribution of component graphs. In addition, the graph is not re-configurable at runtime.

Finally, a number of researchers have considered the impact of emerging application areas on middleware. For example, a number of middleware platforms have been developed to support multimedia [9, 17, 18, 29]. Most notably, researchers at CNET have developed an extended CORBA platform to support multimedia [4]; this platform features the concept of recursive bindings and has been highly influential in our research (a related platform is also reported in [11]). Studies have also been carried out in the areas of mobility [10, 32], real-time [33] and group support [24]. While many of these designs support a level of configurability, they do not offer the level of openness and adaptivity that we seek in our research.

9 Concluding Remarks

This paper has addressed the issue of distributed systems support for mobile applications. It is now well recognised that this equates to support for adaptation. However, in our opinion, many current solutions address adaptation in a rather ad hoc manner; we believe a more principled approach is required in terms of i) open access to underlying systems components, and ii) QoS management of such components. We also believe that middleware is the right place to place such functionality. The paper has presented the design and implementation of a reflective middleware architecture that, we argue, provides such principled support for mobile applications[1]. The most important features of this architecture are i) the ability to associate a meta-space with every component/interface, ii) the sub-division of meta-spaces into four orthogonal models, and iii) the consistent use of component graphs to represent composite components in the architecture. Crucially, the architecture also provides a language-independent model of reflection (as required by the field of open distributed processing).

The architecture is supported by a component framework featuring an open and extensible set of primitive and composite components. In addition, our approach to QoS management has a number of interesting properties in terms of i) offering support for the dynamic insertion and adaptation of QoS management components, ii) allowing policies to operate over all meta-models (including crucially the resource meta-model), and iii) enabling the creation of sophisticated management structures featuring, for example, policies and meta-policies.

We now have considerable experience in constructing configurable and open middleware platforms based on our architecture and feel that this is a highly promising approach. Ongoing research is now addressing the re-engineering of the architecture in terms of a lightweight (reflective) component model in order to address issues of performance in reflective middleware platforms.

[1] Given the degree of openness and extensibility inherent in the design, we also believe the architecture can support a diverse range of application requirements

Acknowledgments

The research described in this paper is partly funded by the CNET, France Telecom (CNET Grant 96-1B-239) and by the EPSRC together with BT Labs (Research Grant GR/K72575). We would like to thank our collaborators for their support. Particular thanks are due to Jean-Bernard Stefani and his group at CNET, and also Ian Fairman, Alan Smith and Steve Rudkin at BT Labs.

References

[1] G. Agha. The Structure and Semantics of Actor Languages. *Lecture Notes in Computer Science*, 489:1–59, 1991. Springer-Verlag.

[2] G. Blair, A. Andersen, L. Blair, and G. Coulson. The Role of Reflection in Supporting Dynamic QoS Management Functions. Technical Report MPG-99-03, Computing Department, Lancaster University, 1999.

[3] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An Architecture for Next Generation Middleware. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer, 1998.

[4] G. Blair and J. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.

[5] L. Blair and G. Blair. Composition in Multi-paradigm Specification Techniques. In *3rd International Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'99)*, Florence, Italy, February 15th-18th 1999. Kluwer.

[6] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation Protocols. *Computer Communication Review*, 20(4):200–208, September 1990. SIGCOMM'90.

[7] F. Costa, G. Blair, and G. Coulson. Experiments with Reflective Middleware. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems*, Brussels, Belgium, 20, July 1998. Springer-Verlag.

[8] G. Coulson. A Distributed Object Platform Infrastructure for Multimedia Applications. *Computer Communications*, 21(9):802–818, July 1998.

[9] G. Coulson, G. Blair, F. Horn, L. Hazard, and J. Stefani. Supporting the Real-Time Requirements of Continuous Media in Open Distributed Processing. *Computer Networks and ISDN Systems*, 27(8), 1995.

[10] N. Davies, A. Friday, G. Blair, and K. Cheverst. Distributed Systems Support for Adaptive Mobile Applications. *ACM Mobile Networks and Applications*, 1(4), 1996. Special Issue on Mobile Computing - System Services.

[11] B. Dumant, F. Horn, F. Dang Tran, and J. Stefani. Jonathan: An Open Distributed Processing Environment in Java. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*. Springer, September 1998.

[12] D. Engler, M. Kaashoek, and J. O'Toole jnr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *15th ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[13] T. Fitzpatrick, G. Blair, G. Coulson, D. N., and P. Robin. A Software Architecture for Adaptive Distributed Multimedia Systems. *IEE Proceedings on Software*, 145(5):163–171, October 1998.

[14] M. Hayden. *The Ensemble System*. PhD thesis, Dept. of Computer Science, Cornell University, USA, 1997.

[15] R. Hayton. FlexiNet Open ORB Framework. Technical Report 2047.01.00, APM Ltd., Poseidon House, Castle Park, Cambridge, CB3 ORD, UK, October 1997.

[16] A. Hokimoto and T. Nakajima. An Approach for Constructing Mobile Applications using Service Proxies. In *16th International Conference on Distributed Computing Systems(ICDCS'96)*, May 1996.

[17] Interactive Multimedia Association. Multimedia System Services - Part 1: Functional Specification (2nd Draft). IMA Recommended Practice, September 1994.

[18] Interactive Multimedia Association. Multimedia System Services - Part 2: Multimedia Devices and Formats (2nd Draft). IMA Recommended Practice, September 1994.

[19] R. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1):6–17, Quarter 1994.

[20] G. Kiczales, J. des Rivières, and D. Bobrow. *The Art of Meta Object Protocol*. MIT Press, 1991.

[21] T. Ledoux. Implementing Proxy Objects in a Reflective ORB. In *ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation*, Jyväskylä, Finland, June 1997.

[22] C. Lindblad and D. Tennenhouse. The VuSystem: A Programming System for Computer-Intensive Multimedia. *IEEE Journal of Selected Areas in Communications*, 14(7):1298–1313, 1996.

[23] E. Madelaine and R. de Simone. *FC2: Reference Manual Version 1.1.*, 1994. See `http://www.inria.fr/meije/verification/doc.html`.

[24] S. Maffeis and D. Schmidt. Constructing Reliable Distributed Communication Systems with CORBA. *IEEE Communications Magazine*, 14(2), February 1997.

[25] F. Manola. 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.

[26] S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *European Conference on Object-Oriented Programming (ECOOP'91)*, LNCS 512, pages 231–250, Geneva, Switzerland, 1991. Springer-Verlag.

[27] J. McAffer. Meta-Level Architecture Support for Distributed Objects. In G. Kiczales, editor, *Reflection 96*, pages 39–62, San Francisco, 1996. Available from Dept of Information Science, Tokyo University, 1996.

[28] S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T.-K. Tung, and D. Wu. Toward a Common Infrastructure for Multimedia-Networking Middleware. In *7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97)*, St. Louis, Missouri, May 1997.

[29] Object Management Group. AV Strems RTF, 1998. Available from `http://www.omg.org/library/schedule/AV_Streams_RTF.htm`.

[30] Object Management Group. CORBA Components Final Submission. Document `orbos/99-02-05`, February 1999. Available from `http://www.omg.org/cgi-bin/doc?orbos/99-02-05`.

[31] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/d: A Distributed Programming System with Multi-Model Reflection Framework . In *Workshop on New Models for Software Architecture*, Tokyo, November 1992.

[32] A. Schill and S. Kümmel. Design and Implementation of a Support Platform for Distributed Mobile Computing. *Distributed Systems Engineering*, 2(3):128–141, 1995.

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

[34] A. Singhai, A. Sane, and R. Campbell. Reflective ORBs: Supporting Robust, Time-critical Distribution. In *ECOOP'97 - Workshop on Reflective Real-Time Object-Oriented Programming and System*, Jyväskylä, Finland, June 1997.

[35] B. Smith. *Procedural Reflection in Programming Languages*. PhD thesis, MIT Laboratory of Computer Science, 1982. Technical Report 272.

[36] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[37] T. Watanabe and A. Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings of OOPSLA'88*, volume 23 of *ACM SIGPLAN Notices*, pages 306–315. ACM Press, 25-30 September 1988. Also available as Chapter 3 of Object-Oriented Concurrent Programming, A. Yonezawa, M. Tokoro (eds), pp 45-70, MIT Press, 1987.

[38] A. Watters, G. van Rossum, and J. Ahlstrom. *Internet Programming with Python*. Henry Holt MIS/M&T Books, September 1996.

[39] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of OOPSLA'92*, volume 28 of *ACM SIGPLAN Notices*, pages 414–434. ACM Press, October 1992.