An Architecture for Next Generation Middleware

G.S. Blair, G. Coulson, P. Robin, M. Papathomas
Distributed Multimedia Research Group
Computing Department, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK
Tel: +44 1524 65201, Fax: +44 1524 593608
E-mail: {gordon, geoff, pr, michael}@comp.lancs.ac.uk

Abstract
This paper proposes an approach to the design of configurable and open middleware platforms based on the concept of reflection. More specifically, the paper introduces a language-independent reflective architecture featuring a per-object meta-space, the use of meta-models to structure meta-space, and a consistent use of object graphs for composite components. This is complemented by a component framework supporting the construction of meta-spaces. The paper also reports on experiences of implementing the architecture (with emphasis on experiments with open bindings).

Keywords
Middleware, (re-)configuration, reflection, open implementation, open bindings.

1 INTRODUCTION

Middleware has emerged as an important 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 standardisation activities such as the ISO/ITU-T Reference Model for Open Distributed Processing (RM-ODP), OMG's CORBA, the Java RMI, Microsoft's DCOM and the Open Group's DCE.

Although middleware is now well established, it is crucial that the standards remain responsive to new challenges such as groupware, multimedia, real-time, and increasingly mobility. In our opinion, such challenges require new approaches to the engineering of middleware platforms in terms of being able to *configure* the underlying support offered by the middleware platform. For example, multimedia applications require very specific support in terms of specialised communications protocols and resource management. Similarly, real-time applications may require lightweight, small footprint Object Request Brokers (ORBs). Although essential, configurability is not enough. It is also important to adopt an *open engineering* approach allowing *inspection* and *adaptation* of underlying components at run-time. For example, mobile systems need to detect and adapt to drastic changes in connectivity; this may involve changes at a number of different levels in the system (e.g. introducing header compression, altering the protocol stack, inserting filtering components, etc).

Support for configurability and open engineering is not available in the current generation of middleware platforms, which typically adopt a *black box* philosophy to their design (thereby hiding implementation details from the applications). Although there are good reasons for this approach, there are already signs 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. 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 RM-ODP architecture does address this issue by distinguishing between computational and engineering concerns. However, we believe this approach is still not sufficient to meet the demands of the next generation of distributed applications (see section 3.3 below).

This paper proposes an approach to the design of configurable and open middleware platforms based on the concept of *reflection*. More specifically, the paper presents a reflective architecture for next generation middleware platforms, supplemented by an open and extensible component framework. We view reflection as a principled (as opposed to ad hoc) means of achieving our desired characteristics.

2. BACKGROUND ON REFLECTION

The concept of reflection was first introduced by Smith (1982). 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 (Kiczaleset al, 1991) (Watanabe and Yonezawa, 1988) (Agha, 1991). More recently, the work has diversified with reflection being applied in operating systems (Yokote, 1992) and, more recently, distributed systems (see section 5).

The primary motivation of a reflective language or system is to provide a principled (as opposed to ad hoc) means of achieving open engineering. 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 object 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 3.3).

3. AN ARCHITECTURE FOR REFLECTIVE MIDDLEWARE

3.1. General principles

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 (Kiczales et al, 1991), 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 also important given the predominance of such models in open distributed processing (Blair and Stefani, 1997a). 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 can have multiple interfaces, ii) operational, stream and signal interfaces are supported, and iii) 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 (Blair and Stefani, 1997a). In contrast, with RM-ODP, however, we adopt a consistent object model throughout the design (see section 3.3).

The second principle behind our design is to have *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 change (see discussion in section 3.3). 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. This aspect of the design is discussed in more depth in section 3.2.2.

As our third principle, we adopt a *procedural approach* to reflection, i.e. the meta-level (selectively) exposes the actual program that implements the system (Maes, 1987). This approach has a number of inherent advantages over a more declarative approach. For example, a procedural approach is more primitive in the sense that it is possible to support declarative interfaces on top of procedural reflection but not vice versa. In addition, the required property of *causal connection* is automatically maintained (as the implementation itself is directly manipulated) (Maes, 1987). 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 important in our design and is revisited in section 3.2.1 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 (Okamura et al, 1992). 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 three aspects currently employed are: *composition*, *encapsulation* and *environment*. This is however not a closed list and the range of models may be expanded later. Further details of each of the models can be found below.

3.2. Design

3.2.1. The structure of meta-space

The overall design of meta-space is illustrated in figure 1. The meta-space is structured as three distinct meta-space models, covering encapsulation, composition and environment. To support this, reflective objects must support a set of operations to reify each of the meta-space models. These operations are `encapsulation()`, `composition()` and `environment()` respectively. Crucially, these operations provide language *independent* access to the meta-space, although the level of access to each model may be language *dependent* (we return to this issue below when we look at the details of each individual model).

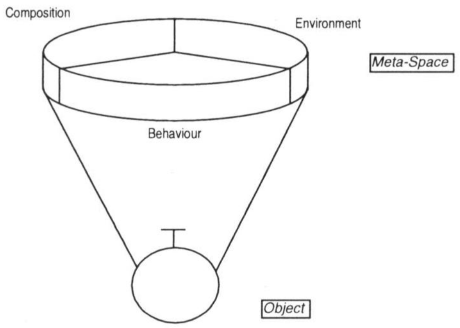

Figure 1 Overall structure of meta-space.

It is important to realise that, in our approach, objects can have multiple interfaces. Given this, meta-spaces are actually associated with *each interface* (as opposed to with each object). More precisely, each interface has an associated encapsulation and environment meta-model. The compositional model is however treated differently. In this case, the meta-model is associated with the object itself and is thus *common* to all interfaces. The compositional meta-model is accessed by calling the `composition()` operation on any of the interfaces associated with the object. This distinction should become clearer when the precise details of the different meta-models are considered below.

The *compositional meta-model* provides access to the *object* in terms of its constituent objects. The composition of an object is represented as an *object graph*, in which the constituent objects are connected together by *local bindings*. Local bindings are crucial in our design; they provide a language-independent means of implementing the interaction point between interfaces. Importantly, some objects in this graph can be (distributed) binding objects, allowing distributed configurations to be created (we return to this aspect in section 4.1 below). In practice, the `composition()` operation returns a graph object with operations to inspect and adapt the composite object, i.e. to view the structure of the graph, to access individual objects in the graph, and to adapt the graph structure and content. For objects that are not composite, the `composition()` operation returns null.

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. The level of access provided by the encapsulation model is clearly language dependent. For example, with compiled languages such as C access may be limited to inspection of the associated IDL interface. With more open (interpreted) languages, such as Java or Python (see section 4.2), 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 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.

Finally, 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, dispatching, unmarshalling, thread creation and scheduling (plus the equivalent on the sending side) (Watanabe and Yonezawa, 1988) (McAffer, 1996). Again, different levels of access are supported. For example, a simple meta-model may only deal with the arrival and dispatching of messages at the particular interface. More complex meta-models will allow the insertion of additional levels of transparency or control over thread creation and scheduling. As with the encapsulation meta-model, this level of heterogeneity is accommodated within an open and extensible type hierarchy. Crucially, the environment meta-model is represented as a composite object. Hence, this aspect of the meta-space is inspected and adapted at the meta-meta-level using the graph manipulation objects, i.e. by calling operations on `composition(environment(object))`. Such operations, can be used, for example, to insert a QoS monitor at the required point in the graph.

Note that there is a high level of recursion in the above definition. In particular, the meta-level is realised using object-oriented techniques. Hence, objects/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, objects/interfaces at the meta-meta-level have an associated meta-meta-meta-space. This process continues providing an *infinite tower of reflection*. 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. This provides a finite representation of an infinite structure (a similar approach is taken in ABCL/R (Watanabe and Yonezawa, 1988)). Access to different meta-levels is important in our design although most access will be restricted to the meta- and meta-meta-levels.

An initial implementation of the three meta-models is described in (Costa et al, 1998).

3.2.2. Reflective groups

As stated earlier, there are situations where it is useful to be able to access the meta-spaces of sets of interfaces in one action. Indeed, a number of reflective languages provide such a facility as core functionality (e.g. ABCL/R2 (Matsuoka et al, 1991)). We propose to accommodate this requirement through the general mechanism of group bindings (or simply groups). Groups provide a uniform mechanism for invoking a set of interfaces whether they are at the base-level, meta-level, meta-meta-level, etc. For example, groups can be used, at the base level, to multicast new share price information to a set of interested parties or, at the meta-level, to insert a new QoS monitoring method in a group of objects.

Importantly, groups are themselves reflective, in that it is possible to access each of the meta-models associated with the group. More specifically, groups are composite objects, where the compositional meta-model provides access to the individual interfaces comprising the group. Through access to the object graph, it is possible to tailor and adapt the semantics of group message distribution, e.g. by introducing a new ordering protocol. It is also possible to manipulate management aspects of the group, e.g. membership policies or collation policies.

3.2.3. Component framework

To complement our open architecture, we provide an open and extensible library of *components* which can be configured to build middleware platforms (i.e. meta-spaces) for a range of applications. The component framework is essential to populate the abstract architecture described above. Note that components can be used at all levels of the reflective architecture.

The component framework consists of both primitive components and composite components, each supported by associated *factory* objects (extending the component framework simply involves introducing a new factory object). Primitive components include:

- a range of low-level (indivisible) communications protocols, such as IP and IP-multicast, and, where appropriate, micro-protocol components to support the construction of protocol stacks, e.g. header compression, flow control, rate control, etc;
- a range of end-system components, such as filters, buffers, dispatchers, demultiplexers, threads, etc;
- a range of management components such as scheduling policies, dispatching policies, buffer allocation policies, and QoS management components;

- one or more implementations of the local binding operation, e.g. same address space or cross address space, same language or cross language, etc.

Composite components then represent off-the-shelf configurations of components (which may in turn include other composite components). The associated factories therefore embody a policy with respect to the selection and configuration of components. Examples of composite components include:

- a range of transport bindings supporting stream and operational interaction;
- end-to-end stream bindings featuring different filtering components (e.g. MPEG compression and decompression);
- pre-configured environment meta-spaces offering access to buffering, dispatching and scheduling components.

Crucially, *groups* are simply (composite) components in the component framework. A range of group bindings, with different group semantics, can be provided. This implies that groups can be exploited without the need for explicit architectural extension.

An initial set of primitive and compound components is currently being created in the GOPI project (Coulson, 1998).

3.3. Discussion

In our opinion, the reflective architecture described above provides a strong basis for the design of future middleware platforms and overcomes the inherent limitations of technologies such as CORBA (as discussed in section 1). In particular, the architecture offers principled and comprehensive access to the engineering of a middleware platform. This compares favourably with CORBA which, as stated above, 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. This approach is complemented by an open and extensible component framework, containing a range of re-usable services such as dispatching policies, filters and communication protocol or micro-protocols. The fine-grained nature of the component framework further enhances the flexibility of the approach.

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 objects 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 object 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 object/interface has its own meta-space. Thus changes to a meta-space can only affect a single object. Furthermore, the meta-space is highly structured, again minimising the scope of changes. An additional level of safety is provided by the strongly typed object model.

In contrast, the issue of performance remains a matter for further research. Nevertheless, our initial experiences with open bindings (see section 4.1 below) has convinced us that there is not necessarily an unfavourable trade-off between flexibility and performance.

4. EXPERIMENTS IN REFLECTIVE MIDDLEWARE

4.1. Open bindings

4.1.1. Introducing open bindings

As an initial experiment, we focused our attention on the open engineering of binding objects (*open bindings*). This work was carried out as part of the Adapt Project, a collaboration between Lancaster University and BT Labs, with the aim of exploiting open bindings to support mobile multimedia applications. In terms of our architecture, an open binding is an object which provides access to its compositional meta-model. As explained above, this meta-model is represented by a graph of objects, connected together by *local bindings*. For open bindings, objects in the graph may themselves be binding objects. The use of binding objects in graphs allows open bindings to span multiple address spaces or multiple nodes, i.e. bindings are distributed objects.

Note that binding objects in the graph can themselves be open bindings and hence also be composed in terms of object graphs. The nesting bottoms out by offering a set of *primitive bindings* whose implementation is closed. For example, a particular platform might offer RTP or IP services as primitive bindings (depending on the level of openness in the platform). This nested structure provides access to lower levels of the implementation (if required). At a finer granularity, each object in the graph can offer a meta-interface to control its individual behaviour.

The concept of (nested) open binding is illustrated in figure 2 below.

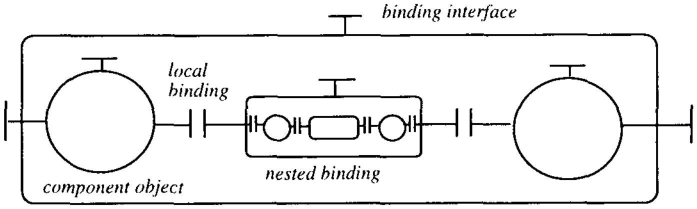

Figure 2 A nested open binding.

4.1.2. Implementation of open bindings

We have implemented an experimental middleware platform featuring the concept of explicit open bindings. This platform is based on a CORBA implementation from Chorus Systems (now Sun Microsystems), called COOL-ORB (Habertet al, 1990). This runs over a variety of computers and operating systems; our experimental testbed consists of laptop PCs running Window-NT. These are interconnected by a variety of networks, including Switched Ethernet, WaveLAN and GSM.

In order to support open bindings, the COOL platform has been extended as shown in figure 3.

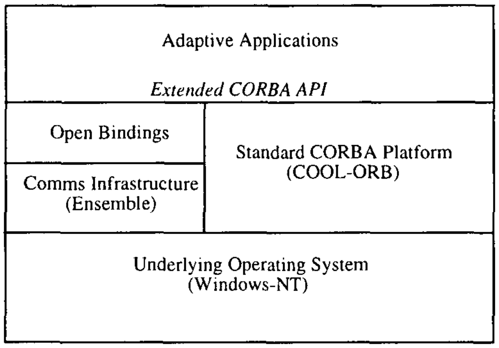
Adaptive Applications
Extended CORBA API
Open Bindings
Comms Infrastructure (Ensemble)
Standard CORBA Platform (COOL-ORB)
Underlying Operating System (Windows-NT)

Figure 3 The extended CORBA platform.

Currently CHORUS/COOL-ORB support two communications infrastructures: TCP/IP and CHORUS IPC. In order to support higher degree of configurability, we have extended COOL with a third communications infrastructure based on Ensemble (Hayden, 1997) (the successor to Horus (van Renesse et al, 1996)). Ensemble enables the programmer to select a particular protocol profile at *bind time* by providing a list of the component modules. In addition, the software supports *run-time* adaptation, in terms of both modification of modules and dynamic reconfiguration. For example, the parameters for flow control can be modified at run-time. Similarly, Ensemble allows the programmer to *switch* to an alternative protocol stack at any point during an interaction. We are currently extending this capability to allow new modules to be introduced dynamically (via the compositional meta-model); such modules could either be an additional layer in the protocol stack or could implement a management function.

For operational bindings, the standard CORBA `bind()` call has been extended i) to enable the specification of the initial protocol graph, and ii) to return an interface providing access to the meta-space (i.e. the object graph representing the configuration of Ensemble modules). Crucially, Ensemble-based operational bindings provide a direct realisation of our concept of reflective groups (as discussed in section 3.2.2). In particular, they support the multicasting of messages to groups of objects (whether at the base-level, the meta-level, etc) and also allow the user to inspect or adapt the underlying protocol stack at run-time (thus altering the semantics of group message passing).

For stream bindings, we provide a set of binding factories offering pre-configured Ensemble stacks for continuous media interaction. Note that it is relatively straightforward to extend the range of bindings by constructing new binding factories from existing components. Prominent among the components are a range of filter objects for common media formats.

Further details of the Adapt Project can be found in the literature (Blair et al, 1997b) (Fitzpatrick et al, 1998).

4.2. Other aspects

In Adapt, the scope of the work is limited to open bindings and to the compositional meta-model. Nevertheless, the project has provided some invaluable experience in constructing open middleware platforms and has also demonstrated the potential of reflection in achieving much greater degrees of openness and adaptivity in such systems. We are now expanding the scope of our work to consider all aspects of middleware design and to develop a complete implementation of our reflective architecture. This work is being carried out using some reflective facilities developed in Python (Watters et al, 1996).

In a related development, we are investigating the potential of formal languages for the expression of scripts for management objects. More specifically, we are considering the role of timed automata in providing a direct representation of scripts for key QoS management functions such as QoS monitoring and adaptation. This work is part of the V-QoS project (a collaboration between Lancaster University and the University of Kent at Canterbury). Finally, we are also investigating the potential of object graphs to support the construction of open and extensible operating systems (Clarke and Coulson, 1998).

The longer term objective of this research is to develop a complete implementation of a reflective middleware platform, based on CORBA. The expected outcome from this research is the definition of a meta-object protocol for CORBA (i.e. a CORBA MOP).

5. RELATED WORK

There is growing interest in the use of reflection in distributed systems. Pioneering work in this area was carried out by McAffer (McAffer, 1996). With respect to middleware, researchers at Illinois have carried out initial experiments on reflection in Object Request Brokers (ORBs) (Singhai et al, 1997). 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 (Hayton, 1997). 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 (Manola, 1993), 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 (Ledoux, 1997).

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 (Okamura et al, 1992). Our ongoing research on the environment and encapsulation meta-models is also heavily influenced by the designs of ABCL/R (Watanabe and Yonezawa, 1988) and CodA (McAffer, 1996). 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 (Matsuoka et al, 1991). 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 object graphs is inspired by researchers at JAIST in Japan (Hokimoto and Nakajima, 1996). 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 (Lindblad and Tennenhouse, 1996) and Mash (McCanne et al, 1997). The same criticisms however also apply to these designs. Microsoft's ActiveX software also uses object graphs. This software, however, does not address distribution of object graphs. In addition, the graph is not re-configurable during the presentation of a media stream.

6. CONCLUDING REMARKS

This paper has presented our design for a next generation middleware platform. This design exploits the concept of reflection to provide the desired level of configurability and openness. We believe that this approach addresses the needs of emerging application areas such as multimedia, real-time systems and mobile computing. Equally importantly, by adopting a principled approach, we believe that the design is sufficiently extensible to meet future demands.

The most important features of our design are i) the ability to associate a meta-space with every object/interface, ii) the sub-division of meta-spaces into three orthogonal models, and iii) the consistent use of object 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 complemented by a component framework featuring an open and extensible set of primitive and composite components. Such components support the construction of meta-spaces.

We already have considerable experience from the Adapt Project in constructing configurable and open middleware platforms based on our architecture (focusing on open bindings). We are now extending this work to look at other aspects of our design (including groups and management). As stated above, the longer term aim of this research is to define a meta-object protocol for future middleware platforms such as CORBA.

7. ACKNOWLEDGEMENTS

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 Andrew Grace, Alan Smith and Steve Rudkin at BT Labs. Finally, we would like to acknowledge the contributions a number of researchers at Lancaster to the ideas described in this paper, namely Anders Andersen (also of the University of Tromsø, Norway), Lynne Blair, Mike Clarke, Fabio Costa, Hector Duran, Nikos Parlavantzas and Katia Saikoski.

8. REFERENCES

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

Blair, G.S. and Stefani, J.B. (1997a) Open Distributed Processing and Multimedia. Addison-Wesley.

Blair, G.S., Coulson, G., Davies, N., Robin, P. and Fitzpatrick, T. (1997b) Adaptive Middleware for Mobile Multimedia Applications. *Proc. 7th International Conference on Network and Operating System Support for Digital Audio and Video (Nossdav'97),* St Louis, Missouri, USA., 259-273.

Clarke, M. and Coulson, G. (1998) An Architecture for Dynamically Extensible Operating Systems. *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, USA.

Costa, F.M., Blair, G.S. and Coulson, G. (1998) Experiments with Reflective Middleware, Internal Report MPG-98-11, Lancaster University Computing Dept., Lancaster LA1 4YR, England. Submitted to Workshop on Reflective Object-Oriented Programming and Systems (in ECOOP'98), Brussels, Belgium, 20-24 July, 1998.

Coulson, G. (1998) A Distributed Object Platform Infrastructure for Multimedia Applications", To appear in *Computer Communications.*

Fitzpatrick, T., Blair, G.S., Coulson, G., Davies, N. and Robin, P. (1998) Supporting Adaptive Multimedia Applications Through Open Bindings. *Proceedings of the 4th International Conference on Configurable Distributed Systems*, IEEE.

Habert, S., Mosseri, L. and Abrossimov, V. (1990) COOL: Kernel Support for Object-Oriented Environments. Proceedings of ECOOP/ OOPSLA Conference, Ottawa, Canada, *ACM SIGPLAN Notices*, **25**, 269-277, ACM Press.

Hayden, M. (1997) The Ensemble System. *PhD Dissertation*, Dept. of Computer Science, Cornell University, USA.

Hayton, R. (1997) FlexiNet Open ORB Framework. *APM Technical Report 2047.01.00*, APM Ltd, Poseidon House, Castle Park, Cambridge, UK.

Hokimoto, A. and Nakajima, T. (1996) An Approach for Constructing Mobile Applications using Service Proxies. *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS'96)*, IEEE.

Kiczales, G., des Rivières, J. and Bobrow, D.G. (1991) The Art of the Metaobject Protocol. MIT Press.

Ledoux, T. (1997) Implementing Proxy Objects in a Reflective ORB. *Proc. ECOOP'97 Workshop on CORBA: Implementation, Use and Evaluation*, Jyväskylä, Finland.

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

Maes, P. (1987) Concepts and Experiments in Computational Reflection. Proceedings of OOPSLA'87, *ACM SIGPLAN Notices*, **22**, 147-155, ACM Press.

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

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

McAffer, J. (1996) Meta-Level Architecture Support for Distributed Objects. *Proceedings of Reflection 96*, G. Kiczales (ed), 39-62.

McCanne, S., Brewer, E., Katz, R., Rowe, L., Amir, E., Chawathe, Y., Coopersmith, A., Mayer-Patel, K., Raman, S., Schuett, A., Simpson, D., Swan, A., Tung, T-K. and Wu, D. (1997) Towards a Common Infrastructure for Multimedia-Networking Middleware. *Proc. 7th International Conference on Network and Operating System Support for Digital Audio and Video (Nossdav'97),* St Louis, Missouri, USA.

Okamura, H., Ishikawa, Y. and Tokoro, M. (1992) AL-1/d: A Distributed Programming System with Multi-Model Reflection Framework. *Proceedings of the Workshop on New Models for Software Architecture*, November 1992.

Singhai, A., Sane, A. and Campbell, R. (1997) Reflective ORBs: Supporting Robust, Time-critical Distribution. *Proc. ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems*, Jyväskylä, Finland.

Smith, B.C. (1982) Procedural Reflection in Programming Languages. *PhD Thesis*, MIT, Available as MIT Laboratory of Computer Science Technical Report 272, Cambridge, Mass.

van Renesse, R., Birman, K.P. and Maffeis, S. (1996) Horus: A Flexible Group Communications Service. *Communications of the ACM*, **39** (4).

Watanabe, T. and Yonezawa, A. (1988) Reflection in an Object-Oriented Concurrent Language. Proceedings of OOPSLA'88, *ACM SIGPLAN Notices*, **23**, 306-315, ACM Press.

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

Yokote, Y. (1992) The Apertos Reflective Operating System: The Concept and Its Implementation. Proceedings of OOPSLA'92, *ACM SIGPLAN Notices*, **28**, 414-434, ACM Press.

9. BIOGRAPHY

Gordon Blair is a Professor of Distributed Systems in the Computing Department at Lancaster University. He has been responsible for a number of research projects at Lancaster in the areas of distributed systems and multimedia support and has published over a hundred papers in his field. He also recently completed a book with Jean-Bernard Stefani (CNET) on "Open Distributed Processing and Multimedia". His current research interests include distributed multimedia computing, open distributed processing, the impact of mobility on distributed systems, and the use of formal methods in distributed systems.

Geoff Coulson is a lecturer in the Computing Department at Lancaster University. Dr. Coulson has published many papers in his field and has contributed to a number of books in the area. He is a member of the ACM and the BCS and has served on program committees and been involved in the organisation of a number of workshops on systems support for multimedia. His current research interests are operating system support for continuous media, distributed systems architectures and high speed networking.

Philippe Robin is a research assistant at Lancaster University. He previously worked at Chorus Systèmes in the development of the CHORUS/MiX operating system and subsequently for a company developing fault-tolerant platforms based on Chorus technology. His areas of research include support of Quality of Service by distributed operating systems, real-time systems and multimedia systems. He is currently working on the design of protocol adaptation mechanisms for mobile platforms.

Michael Papathomas received his MSc and PhD in Computer Science from the University of Geneva, Switzerland in 1985 and 1992 respectively. Following a period as a postdoctoral researcher at Geneva, he joined Lancaster University in 1994 as a Visiting Research Fellow supported by a grant from the Swiss FNRS. On completion of this research, Michael worked as a CaberNet funded Visiting Research Fellow at IMAG, Grenoble, before returning to Lancaster in early 1997. His research interests include pragmatic and formal aspects of concurrent and distributed systems, reflection and object-oriented programming language design, and multimedia.