

The Design and Implementation of Open ORB 2

Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski

Lancaster University

Established middleware platforms such as CORBA and DCOM are not flexible enough to meet the needs of emerging distributed applications. This article discusses the architecture of Open ORB 2, a middleware platform based on reflection and component technology.

Middleware has emerged as an important architectural component in modern distributed systems largely because it offers a high-level, platform-independent programming model that helps mask distribution problems. Examples of key middleware platforms include DCE, CORBA, DCOM, .NET, and the Java-based series of technologies, including RMI, Jini, and EJBs (for more information on these platforms, see the [Middleware Platforms](#) sidebar). Traditionally, developers have deployed such platforms in areas such as banking and finance to overcome heterogeneity and support the integration of legacy systems.

More recently, however, developers have applied middleware technologies in a wider range of areas, including safety-critical, embedded, and real-time systems. It is now becoming apparent that middleware technologies cannot respond to such diverse requirements or technical challenges because of the limitations of the black-box philosophy maintained by developers of most existing middleware platforms. In particular, existing middleware platforms offer fixed services to their users; it is typically impossible to view or alter the implementation of these services. Inevitably, the architecture of this platform then represents a compromise design that features, for example, general-purpose protocols and associated management strategies. It is not possible to specialize platforms to meet the needs of more specific target domains. Other researchers in the field have also recognized these problems.[1,2](#)

Middleware designers are aware of these problems and have responded with several initiatives. The Object Management Group, focusing on CORBA, introduced a series of platform specifications that includes real-time CORBA and Minimal CORBA. But these specifications are specific solutions for specific

domains, not general solutions to this problem. Modern middleware platforms also typically offer more flexibility through mechanisms such as interceptors and configurable protocol stacks. While these are important developments, they do not offer a complete solution to the problem.

We believe next generation middleware platforms should be

- configurable, to meet the needs of a given application domain,
- dynamically reconfigurable, to enable the platforms to respond to changes in their environment, and
- evolvable, to meet the needs of changing platform design.

Recently, several reflective middleware technologies have emerged in response to such requirements (see the [Related Work](#) sidebar). Reflection is a technology that has previously been deployed successfully in the design of languages and operating systems. The key to the approach is to offer a meta-interface supporting the inspection and adaptation of the underlying virtual machine. In the context of middleware, the meta-interface would support operations to discover the internal operation and structure of the middleware platform (such as the deployed protocols and management structures) and to make changes at runtime. The design of such a meta-interface is central to studies of reflection: The interface should be sufficiently general to permit unanticipated changes to the platform but should also be restricted to prevent the integrity of the system from being destroyed.

This article presents the design and implementation of Open ORB, a reflective middleware platform developed at Lancaster University. Specifically, we focus on Open ORB 2, a significant redesign that builds on our experience from the first implementation.³

Note that, for simplicity, we generally refer to Open ORB 2 as simply Open ORB in the rest of this article.

Open ORB 2 architecture

The Open ORB architecture builds on two complementary technologies: components and reflection. Component technologies are gaining widespread acceptance in the middleware community, as evidenced by the popularity of COM and JavaBeans and also the imminent publication of the component-based CORBA 3 specification. In the context of this article, we define a component as "a unit of composition with contractually specified interfaces and explicit context dependencies only."⁴ In addition, a component "can be deployed independently

and is subject to third-party composition."⁴

The middleware technologies we just mentioned adopt a component-based programming model to enhance the configurability, reconfigurability, and level of application reuse. Our model extends these capabilities to the design of the middleware platform itself. In particular, an instance of Open ORB is a particular configuration of components that can be selected at build and reconfigured at runtime. For example, we can opt for a minimal configuration of components (perhaps offering only client-side capabilities) to run on a PDA (Personal Digital Assistant). Similarly, we can replace a particular protocol component at runtime if network conditions change significantly.

Heavily influenced by previous work on the Sumo Project and also by the Computational Model from RM-ODP,⁵ our component model is designed to support multimedia programming. In our model, we describe components in terms of a set of required and provided interfaces, and we support interfaces for continuous media interaction. Our component model enables explicit bindings between compatible interfaces (the result being the creation of a binding component) and offers a built-in event notification facility.

Our model provides access to the underlying platform—and by implication the associated component structure—through reflection. In particular, every application-level component offers a meta-interface that provides access to an underlying metaspace, which is the support environment for the component. Metaspace is itself composed of components. Such metalevel components also have a meta-interface that offers access to their support environment. This approach is therefore recursive, potentially leading to infinite towers of reflection. To overcome the problem of infinite towers, our model instantiates metacomponents on demand; unless accessed, they exist in theory but not in practice.

In our design, metaspace is partitioned into several distinct metaspace models, an approach first advocated by the designers of AL-1/D.⁶ The benefit of this approach is to simplify the interface the metaspace offers by separating concerns between different system aspects. This is particularly important in distributed systems given the wide range of concerns that must be considered (in comparison to designing a single language, for example). [Figure 1](#) illustrates the structure of metaspace.

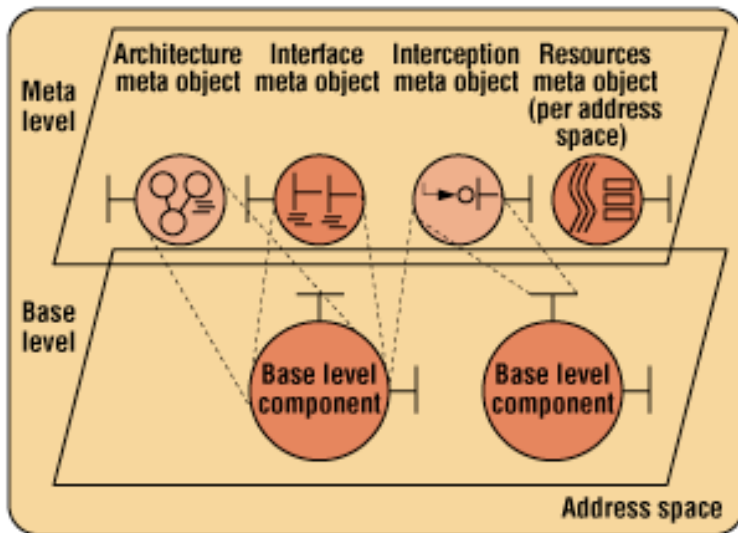


Figure 1. The structure of metaspace.

Metaspace models for structural reflection

In reflective systems, structural reflection deals with the content and structure of a given component.² In our architecture, this aspect of metaspace is represented by two distinct metamodels, namely the interface and architecture metamodels. The two metamodels represent a separation of concerns between the external view of a component (its set of interfaces) and the internal construction (its software architecture).

The interface metamodel provides access to the external representation of a component in terms of the set of provided and required interfaces. It is possible to enumerate all provided (or required) interfaces offered by a given component or to discover the type signature associated with a given interface. This metamodel therefore provides a capability similar to introspection facilities in the Java reflection API, allowing a programmer to interact with a component discovered dynamically in the environment. Unlike the previous design,³ it is not possible to access the internal implementation of an interface (as a set of methods and attributes); nor is it possible, for example, to add methods to this implementation. Rather, we enforce the strict separation between interface and implementation in keeping with this important principle of component-based programming.

The architecture metamodel provides access to the implementation of the component as a software architecture that consists of two key elements: a component graph and an associated set of architectural constraints. The concept of the component graph is central to this design and is represented by a set of components (more specifically interfaces) connected together by local bindings, where a local binding represents a mapping between a required and provided interface in a single address space. Distribution can be added into the model by

introducing distributed binding components into the graph. An extensible set of binding types can offer interaction models such as remote invocation, publish-subscribe, continuous media flows, and group communication. Normally, this structure would be hidden from a component user. However, the architecture metamodel can be used both to discover and to make changes to this structure at runtime.

If unconstrained, this architecture is a rather dangerous approach to advocate. Consequently, we extended the architecture to include a set of architectural constraints. The type management system offers one level of constraints: A new component must be a valid substitution of the old component. It is also important to take a more global view of the architecture in determining validity of adaptations. For example, changing a compression component may require a similar change to the peer decompression component. Similarly, it might be necessary to preserve a given architectural style over time. Our approach is to record such constraints explicitly in the architecture and to ensure that adaptations preserve the architectural rules before committing the changes. We are investigating two alternative approaches for use at runtime:

- an implicit approach, in which constraints are captured indirectly by the interface that an architecture offers (permitting only certain safe operations), and
- an explicit approach, in which constraints are encoded using an appropriate notation.

In both approaches, a mapping can be defined from the design-time representation to the runtime representation.

The architectural metamodel can be applied recursively in that components within a component graph might have architecture, accessed through its architecture metamodel (that is, at a meta-metalevel relative to the uppermost component). For example, a binding component within a graph might have a structure consisting of stubs and protocol components. This recursion terminates with primitive components that have no visible underlying structure and whose internal implementation details are inaccessible to the programmer. Access at this level would inevitably depend on the language of implementation and hence is deemed beyond the scope of a language-independent middleware architecture.

Note that, in the Open ORB approach, interfaces are immutable and represent irrevocable contracts with their environment. Evolution is supported by the addition of new components into an architecture that supports upgraded interfaces. Old clients can still rely on the previous interfaces to offer functional and nonfunctional guarantees, a technique that considerably simplifies the task of

type management.

Another benefit of the new design is that we can place strict controls on access rights for adaptation. More specifically, we can give all classes of users rights to access the interface metamodel. In contrast, rights to the architecture metamodel can be tightly constrained so that only trusted third parties can modify the system architecture. See the sidebar, "[The Design of the Four Metamodels](#)," for the interfaces (or meta-object protocols) for the interface and architecture metamodels.

Metaspace models for behavioral reflection

Behavioral reflection deals with activity in the underlying system.⁷ Open ORB distinguishes between actions taking place in the system and the resources required to support such activity. These two aspects are represented by the interception and resources metamodels.

The interception metamodel is arguably the most straightforward in the Open ORB design, and is a simplified version of the environmental metamodel from the first version of the architecture. In keeping with several reflective middleware proposals, this metamodel enables the dynamic insertion of interceptors. Such interceptors are associated with interfaces (specifically, local bindings) and enable the insertion of pre- and post-behavior, which applies equally to all styles of interface supported in Open ORB (operational, continuous media, and so forth). This mechanism is useful, for example, to dynamically introduce monitoring or accounting into a running system.⁸ Similarly, interceptors can be used to introduce additional nonfunctional behavior, such as security checks or concurrency control.

In contrast, the resources metamodel is a unique feature of the Open ORB design, offering access to underlying resources and resource management.⁹ We believe that for many classes of application (including multimedia applications) it is just as important to be able to adapt resource usage and management policies as to evolve the basic system structure.

The resources metamodel is based on the abstractions of resources and tasks. Resources can be either primitive (for example, raw memory or OS threads) or complex (for example, buffers or user-level threads multiplexed to kernel-level threads). Resource factories create them and resource managers manage them—the latter typically building complex resources by adding value to or combining primitive resource instances. For example, a user-level scheduler is a resource manager that builds user-level threads from OS threads. Tasks are then the logical unit of activity in the system with the precise granularity varying from

configuration to configuration. There could be a single task, for example, that deals with the arrival, filtering, and presentation of an incoming video stream. Alternatively, this task could be divided into several smaller tasks. Tasks can span component boundaries and are thus orthogonal to the system's structure. Tasks are essentially the unit of resource allocation. That is, tasks have a pool of resources to support their execution.

There is a resources metamodel per address space. That is, resources are associated with a particular address space and all components within that address space share the same metamodel. The metamodel provides access to a set of components that represent resource management. As with other metamodels, it is then possible to either inspect or adapt activity associated with resources. For example, you can insert monitors to capture statistics on the effectiveness of a thread scheduling policy and then possibly change this policy based on the information collected. In programming terms, the resources metamodel is represented as a graph structure that organizes resources, tasks, and managers into hierarchical structures.

To extend this work, we developed an enhanced architectural description language (ADL) called Xelha,¹⁰ which builds on the task model described above. We have implemented a tool for the interpretation of Xelha specifications. This tool generates Python object classes and low-level resource descriptions in a resource configuration description language. The main purpose of Xelha is to support the engineering of resource management concerns in distributed real-time systems.

In common with many ADLs, Xelha supports the specification of software architectures in terms of components, their interfaces, and connectors. Interestingly, the ADL also supports the overlaying of a task structure with associated quality of service requirements. The QoS requirements are then used to derive the underlying resource allocation policies for tasks. Finally, the ADL also (optionally) supports the introduction of dynamic QoS management structures in terms of monitoring and controlling components. [Figure 2](#) shows an example of the use of Xelha (omitting dynamic QoS management features).

```
Def connector <stream> AudioConnector_V1(string srcCapsule, string
sinkCapsule):
```

```
  components:
```

```
    srcStub: SrcStub, srcCapsule
```

```
    sinkStub: SinkStub, sinkCapsule
```

```
  connectors:
```

```
    streamConn: StreamConnector(srcCapsule, sinkCapsule)
```

```
  interfaces:
```

```
    interaction:
```

```
      IN: SrcStubIN, ( srcStub, IN )
```

```
      OUT: SinkStubOUT, ( sinkStub, OUT )
```

```
    control:
```

```
      CTRL: StreamConnCTRL, (streamConn, CTRL)
```

```
  composition graph:
```

```
    interfaces:
```

```
      OUT: ( srcStub, OUT )
```

```
      streamConnIN: ( streamConn, IN )
```

```
      streamConnOUT: ( streamConn, OUT )
```

```
      IN: ( sinkStub, IN )
```

```
    edges:
```

```
      ( OUT, streamConnIN )
```

```
      ( streamConnOUT, IN )
```

tasks:

Def task transmitAu.marshall:

switching points:

srcStub:CTRL:start [if taskx]

qos specifications:

delay(srcStub:IN:read, streamConn:IN:put) = 5

throughput(srcStub:OUT:put) = 64

Def task transmitAu includes transmitAu.marshall,
transmitAu.unmarshall:

importance: 5

qos specifications:

10 delay(streamConn:IN:put, streamConn:OUT:put) =

packet_loss(streamConn:IN:put,
streamConn:OUT:put) = 5

delay(srcStub:IN:read, sinkStub:OUT:write) = 20

jitter(srcStub:IN:read, sinkStub:OUT:write) = 1

qos management structure: ...

<not shown for simplicity>

Figure 2. An example of the use of Xelha.

This example is a composite audio connector that consists of source and sink stubs that are interconnected by a stream connector. The early parts of this IDL are fairly traditional. The task section, however, defines the task structure and its associated QoS requirements (used to derive resource allocations as discussed

above). We can see the definition of tasks for marshalling the audio and transmitting the audio. In the full version, there is also a task for unmarshalling the arriving audio packets, but we have omitted that here for brevity. The boundaries between tasks are defined by task switching points (that is, points where the system will switch to alternative pools of resources, such as different set of threads). Again, the meta-object protocols for behavioral reflection in Open ORB can be found in the "[The Design of the Four Metamodels](#)" sidebar.

Extensions

Open ORB 2 contains two other enhancements to the previous version: an integrated approach to meta-information management and the incorporation of group services.

Meta-information management. Meta-information management in Open ORB 2 centers on our type repository, which is an extension to the CORBA interface repository that offers additional features to support our component model. These additional features include, for example, stream interfaces, QoS annotations, media types, and primitive and composite components and bindings. The kind of meta-information the type repository manages includes both the type and template aspects that describe the elements of the component model. The repository therefore aims to provide support for type checking and for the definition, storage, and instantiation of platform configurations.

Another important goal of our meta-information management facility is to provide a unified approach that combines an intentional style of reflection (provided by the type repository) and the more extensional styles Open ORB already supports. In this approach, the type repository becomes the provider of structural meta-information for the meta-objects of the interface and architecture metaspace models. The approach is simplified considerably by the immutability of interfaces and primitive components in Open ORB, which reduces the problems of consistency maintenance between the same meta-information the repository and meta-objects keep. However, composite components and bindings can still change, for example, by featuring a new interface or by adapting the internal architecture.

The subtype relationship the repository supports can validate the reconfigurations before they take place (to guarantee that the changed configuration is a valid replacement for the previous one). After reconfiguration has taken place and the new configuration is stable, we can automatically derive type (and template) meta-information from the self-representation that the involved architecture meta-object maintains. We can then publish such meta-information in the repository for future use as a version of the original type.

Finally, the model of the type repository (or, more precisely, its metamodel) is defined in terms of the CORBA Meta-Object Facility, which introduces the ability to evolve the type system over time, although we have not yet explored this aspect in any depth. We are also investigating supporting the meta-information facility in environments with limited resources, such as embedded and mobile systems. One approach is to provide only the client side of the facility in such environments, with the actual repository residing elsewhere on the network. Another possibility is to define a lightweight implementation of the facility that is based on the structural meta-objects as the placeholder for local meta-information.[11](#)

Group services. We have extended communication capabilities in the platform to include configurable and reconfigurable group services through the provision of a group factory. The latest model supports both open and closed group services. The basic group factory supports a single method, `createGroup`, which takes an XML template as a parameter. This template defines the required software architecture of the group (as a component graph), one or more member types, and zero or more service types. A member type is essentially a template for future members of a group, given by their interface type and a per-member configuration.

Service types are used in conjunction with open groups and represent an externally visible view of the group. Crucially, the XML template is used to configure the group; for example, groups can be created with or without monitoring components in place. A group also features a default group management interface with the operations `join`, `leave`, and `getMembers`. The group service is also supported by a library of base components for IP multicast, reliable communication, ordering, and collation. The resultant group service can be accessed using the various metamodels defined earlier. For example, the interface metamodel can be used to discover the external interfaces offered by a group. Similarly, the architecture metamodel can be used to adapt the implementation at runtime (subject to the normal architectural constraints). Further details of these extensions can be found elsewhere.[12](#)

Efficient implementation

The general Open ORB architecture has evolved through a series of prototypes written in Python. Python was a natural choice for this prototyping work, given its intrinsic support for rapid prototyping and also its underlying reflective capabilities. Nevertheless, given Python's interpreted nature, it is not possible to fully investigate the performance characteristics of a reflective middleware platform. Consequently, we initiated a parallel activity to investigate the efficient

implementation of Open ORB using C++. The goal is to have a reflective middleware platform that (in standard configuration) performs at least as well as commercial ORBs but offers the additional benefits of reflective middleware.

The approach we adopted is to define a base reflective component model, Open COM,¹³ as an extension to Microsoft's COM architecture and to use it to implement a component-based middleware platform.

Open COM component model

Open COM is closely based on Microsoft's COM but is enhanced with richer reflective facilities. It relies only on the core of COM: on the provided interfaces, the IUnknown interface, and the basic language-independent binary-level standard that enables components to be dynamically composed within a single address space. Open COM avoids dependencies on other features of COM, such as distribution (through DCOM), persistence, security, and transactions. Our model does not retain interoperability with other COM components. Moreover, the binary-level nature of interconnections promises considerable performance benefits over other component models such as JavaBeans.

One limitation of COM is that there are no mechanisms to make the connections between components explicit. If one component depends upon the interface of another (we call this a required interface of the component) then it is accessed through a simple pointer variable whose type and location are lost at compile time. This clearly makes it impossible to track dependencies between components at runtime and consequently means that COM components cannot be dynamically reconfigured.

In our model, we define the receptacle data structure as a first class runtime entity that maintains pointer and type information for a connection between a component and a required interface. Connections are established explicitly so that they are made known to the system. The component developer implements an interface (IReceptacles) in order to allow the system to access the component's receptacles. Receptacles also contain other elements (locks, for example) to allow the system to prevent invocations through a receptacle when a reconfiguration on the connection is taking place.

Open COM provides low-level support for our metamodels as follows:

- The IMetaInterface interface provides meta-information relating to the interface and receptacle types of a component. This interface can also be used to support dynamic invocation of arbitrary methods as in Java core reflection.

- The IMetaArchitecture interface provides access to the underlying graph structure of components and their connections (assuming the component is not primitive).
- The IMetaInterception interface enables the dynamic attachment or detachment of interceptors as defined earlier.

This is only a subset of our reflective features. For example, we do not offer a resources metamodel at this level. Rather, we expect a resources metamodel to be constructed above this level (in the Open ORB implementation). Similarly, we only offer a partial implementation of the architecture metamodel because we assume that architectural constraints will be introduced at a higher level. [Figure 3](#) illustrates Open COM's architecture.

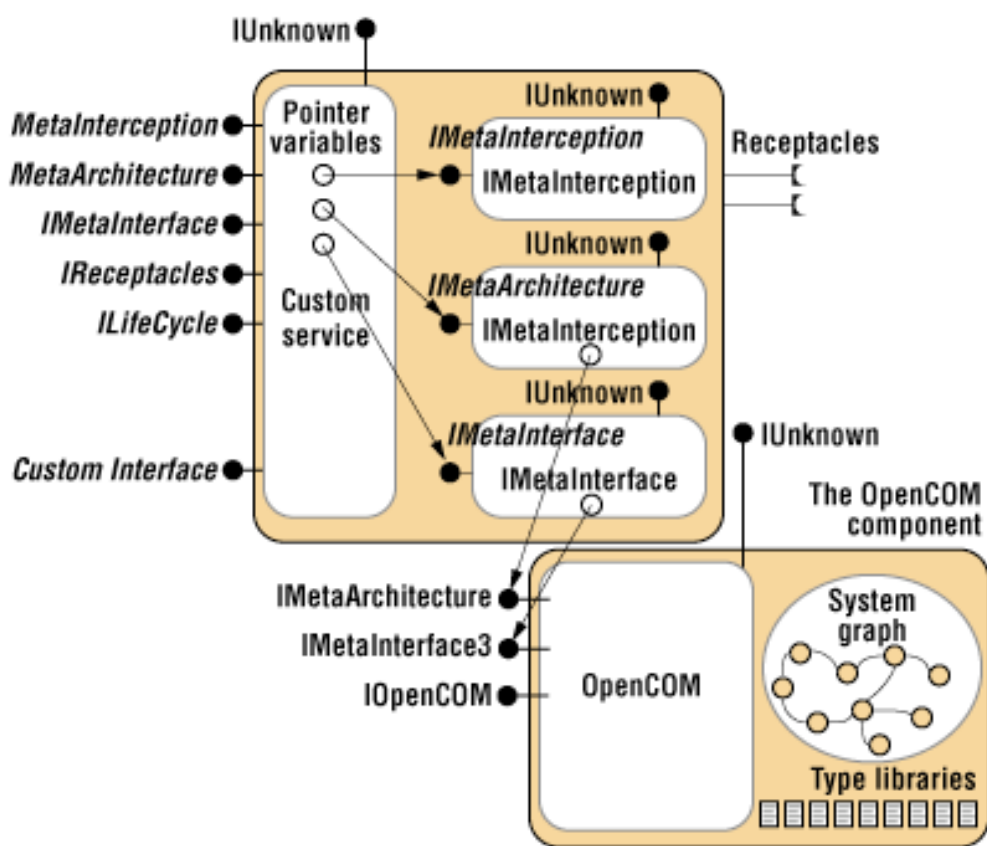


Figure 3. The Open COM architecture.

Middleware architecture

We have used the Open COM component model to develop an implementation of the Open ORB architecture. This platform can offer a CORBA-compliant interface but is both configurable and dynamically reconfigurable through the additional reflective features (for example, the CORBA interface could be replaced by a SOAP interface). The platform also provides support for multimedia, including streaming audio and video services. The GOPI platform heavily influences the design.¹⁴

The Open COM implementation exploits the concept of component frameworks—rules and contracts that govern the interaction of a set of components.⁴ The main motivation for using component frameworks is to constrain the design space and the scope for evolution. Moreover, component frameworks help simplify component development and assembly, enable lightweight components, and increase the system’s understandability and maintainability.

Component frameworks have often been used as a design-time concept. We have adopted this approach at runtime. In our approach, component frameworks define architectures (component graphs and constraints) for specific domains and provide partial support for our architecture metamodel. In particular, our component frameworks are explicitly represented as components (called component framework representatives) that are responsible for implementing the architecture meta-interfaces while enforcing the architectural constraints. The middleware architecture is then decomposed into an extensible set of specialized and focused domains of concern, such as buffer management and binding establishment, that are each based on a component framework.

Specifically, we organized the middleware architecture into three layers, as shown in [Figure 4](#).

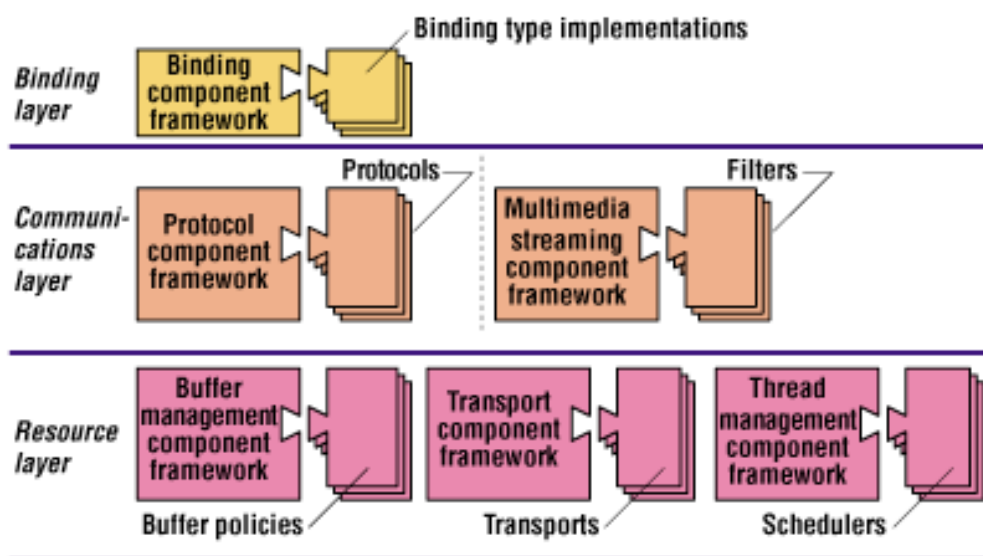


Figure 4. The component frameworks in Open ORB.

The binding layer contains the binding component framework that accepts a variety of binding type implementations. The communications layer contains the protocol component framework. Within this framework, a reconfiguration manager maintains information about the current protocol stack (organized as a component graph), which can then be adapted using the architecture meta-interface. The component framework ensures that the overall integrity of the

protocol structure is maintained during reconfiguration. This layer can be configured to include additional component frameworks, such as, for example, a multimedia-streaming framework featuring a filter style.

We are developing a component framework for multiparty communication protocols to support the group service just described. At the lowest level, the resources layer has several component frameworks for buffer, transport, and thread management. Again, adaptation can be tailored for the particular domain. For example, the thread management component framework enables the dynamic installation of scheduler components and the migration of existing threads between schedulers.

The binding layer is arguably this architecture's most interesting feature. In contrast to most existing middleware platforms, the Open ORB implementation supports an extensible set of binding types, including remote method invocation, publish-subscribe, message queuing, group communications, and media streaming. Binding types aim to separate communication and coordination aspects from computational aspects to simplify development. They also effectively implement software architecture connectors, which helps bridge the gap to software architecture research.

The API offered by the binding component framework is based on a small number of abstractions (iref, binder, resolver, generator, participant, and so forth), all designed to capture commonalities across diverse binding types. The API does not attempt to specify a uniform interface to all binding types (which is clearly infeasible) but to offer guidelines and generic interfaces that provide consistency for binding users and guidance for binding type implementers. Binding types can be configured statically and also changed at runtime by dynamic loading when an iref of a specific type arrives. More fine-grained changes to the binding type implementations can also be made using the reflective facilities of the component model. Further details of the binding component framework can be found in a forthcoming article.[15](#)

Overall, the implementation of the platform consists of six component frameworks, around 25 Open COM components, and in total around 50,000 lines of C++ code (including support for audio and video streaming). We are currently thoroughly investigating performance and we will report our findings in the forthcoming article cited above.[15](#) Early figures indicate that we will comfortably meet our target of outperforming commercial CORBA implementations. For example, [Figure 5](#) provides an indication of the relative performance of the platform against Orbacus 3.3.4 (one of the fastest commercially available ORBs) and GOPI 1.2.

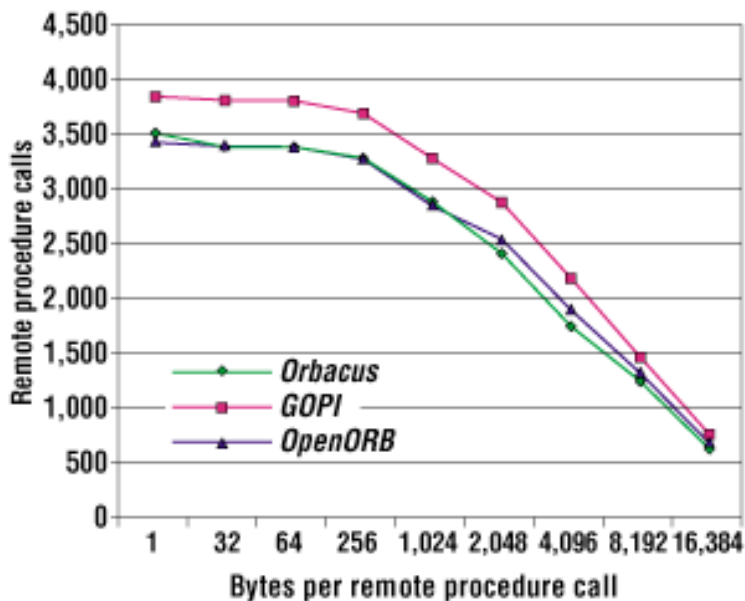


Figure 5. Performance measurements.

For packets of less than 1,024 octets, Open ORB performs about the same as Orbacus, with GOPI running around 10 percent faster. As might be expected, there is a diminishing difference between all three systems as packet size increases. This diminishing difference is presumably because the overhead of data copying begins to outweigh the cost of call processing. A comparison of Open ORB and GOPI also provides a first approximation of the overheads of supporting reflection in Open ORB. This comes out at around a 10 percent difference between GOPI and Open ORB in terms of invocation throughput. We feel this is very encouraging for the Open ORB project given the added capabilities of configurability and reconfigurability in Open ORB (which can both be exploited to improve the performance from this baseline figure by removing unnecessary components at runtime or introducing more specialist components for given network conditions).

We performed tests over the loopback interface on a Dell Precision 410 workstation with a 550-Mhz Pentium III processor and 256 Mbytes of RAM. We used Microsoft's cl.exe compiler with flags /MD /W3 /GX /FD /O2.

Experimental evaluation of Open ORB 2

The first experiments with Open ORB focused on using reflection to support dynamic reconfiguration. We experimented with adapting the structure of continuous services in response to fluctuations in the underlying network QoS and we developed a QoS management subsystem that offers both monitoring and controlling capabilities. In particular, we achieved QoS management by introducing management components into the component configuration (accessed through the architecture metamodel). Our architecture relies on different styles of management component, described in [Table 1](#).

Table 1. Styles of management components.

Monitoring	Role
Event collector	Observe behavior of underlying functional components and generate relevant QoS events.
Monitor	Collect QoS events and report abnormal behavior to interested parties.
Control	
Strategy selectors	Select an appropriate adaptation strategy (strategy activator) based on feedback from monitors.
Strategy activators	Implement a particular strategy (by manipulating component graph, for example).

Event collectors and strategy activators are at the lowest level of this architecture, interfacing directly with the managed components and using reflection to gain the required level of access to the underlying infrastructure (in terms of both introspection and adaptation). Monitors and strategy selectors, in contrast, are more abstract and effectively represent the QoS management policy to decide, for example, that the best course of action to be taken in response to increasing jitter in the network is to increase the buffer size of the receiver.

Our prototype system expresses the policies for monitoring and strategy selection as timed automata, which then map directly to management components that act as timed automata interpreters at runtime. These interpreters then interface to other components in the system using event notification. That is, they register for events of interest, receive events, react to them, and then emit events to interested parties (behaving in many ways like reactive objects¹⁶). This use of timed automata also allows us to carry out formal analysis of the behavior of the QoS management subsystem in isolation, and also when composed with a model of the rest of the system.

We have implemented two examples using this approach: a QoS-managed audio-streaming application and a synchronization protocol for stored video (both of which use the Python-based prototype). Experiments are underway to extend this work to allow the dynamic reconfiguration of resources and resource-management policies. Our experiences from this work so far have been extremely positive. In particular, the experiments largely confirm our hypothesis that reflection provides a principled approach to supporting adaptation in distributed systems (both in terms of the ability to monitor and also the ability to change the underlying configuration). It is heartening that QoS management can be introduced dynamically in our architecture, even if it has not been planned in advance. Given the recursive nature of the architecture, it is possible to monitor and adapt the management components to check, for example, whether a given policy is operating satisfactorily.

Additional details of the QoS management architecture and associated experiments can be found elsewhere,^{[17](#)} in addition to a more general treatment of adaptation in Open ORB.^{[18](#)}

Ongoing experiments

Work is now continuing on a series of experiments to evaluate the architecture more completely. To date, there has been very little work on exploiting the configurability inherent in the Open ORB architecture (apart from the experiments with the configuration of group services as reported above). We are currently investigating the use of Open ORB to configure middleware for minimal devices, such as the Palm Pilot. In addition, we are examining the use of the component-based approach, together with the reflective facilities of Open ORB, to support the development of cooperative visualisation environments (in collaboration with the Rutherford Appleton Labs).^{[19](#)} In both cases, the longer-term goals are also to support dynamic reconfiguration, broadening our experience in this area.

We are convinced that the Open ORB architecture also provides interesting support for the longer-term evolution of software as requirements change over time. For example, the architecture metamodel captures the initial software architecture of the system in terms of components, connectors, and architectural constraints. The designer can both access this software architecture, and then use this information as the basis for making changes to the design. We are currently investigating this premise in two application domains, namely digital libraries and banking.

For example, in the digital library setting, we are considering the support offered by Open ORB for a series of evolutionary steps such as introducing continuous media services, supporting mobile users, and enhancing the scalability of the

platform. As part of this work, we are also re-implementing the Open ORB architecture in Java. In this implementation, though, we are investigating an explicit approach to representing architectural rules. This approach enables us to evolve the architecture itself (by modifying the constraints over time, for example).

The experiments we've described will provide a relatively complete evaluation of reflective middleware's potential in general and Open ORB's potential in particular.

The most important work to be done in the future is to reach international consensus on the need for reflective middleware technology and on the interfaces to be offered by such platforms. Consensus is important to support the emergence of a next-generation of middleware platforms that are more configurable and reconfigurable, but that also offer portability and interoperability for applications that choose to exploit such advanced features.

Acknowledgments

The research described in this article is partly funded by France Telecom R&D (CNET Grant 96-1B-239). We thank Jean-Bernard Stefani and his group at France Telecom for many useful discussions on reflection and distributed systems. The work is also partly funded by the EPSRC together with BT Labs through grant GR/M04242). In particular, we acknowledge the contributions of the following people from BT: Steve Rudkin, Alan Smith, Paul Evans, Kashaf Khan, and Benjamin Bappu. The work on specification and formal verification of QoS properties is also funded by EPSRC (Research Grant GR/L28890). Fabio Costa, Hector Duran Limon and Katia Saikoski are funded by CNPq/ UFG (Brazil), UNAM (Mexico) and CAPES/ PUCRS (Brazil), respectively. Rui Moreira is also partially funded by UFP (Universidade Fernando Pessoa). We acknowledge the contributions of our partners on the CORBAng project (next generation CORBA) at UniK and the Universities of Oslo and Tromsø. Particular thanks to Frank Eliassen, Vera Goebel, Øyvind Hanssen, and Thomas Plagemann.

References

1. N. Wang et al., "Toward an Adaptive Reflective Middleware Framework for QoS-Enabled CORBA Component Model Applications," IEEE Distributed Systems Online, vol. 2, no. 5, 2001, http://dsonline.computer.org/0105/features/wan0105_1.htm (current 28 June).

2. M. Roman, F. Kon, and R.H. Campbell, "Reflective Middleware: From the Desk to your Hand," IEEE Distributed Systems Online, 2001, vol. 2, no. 5, 2001, http://dsonline.computer.org/0105/features/rom0105_1.htm (current 28 June).
3. G.S. Blair et al., "An Architecture for Next Generation Middleware," Proc. IFIP Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer-Verlag, New York, 1998, pp. 191-206.
4. C. Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley, Reading, Mass., 1998.
5. G.S. Blair and J.B. Stefani, Open Distributed Processing and Multimedia, Addison-Wesley, Reading, Mass., 1998.
6. H. Okamura, Y. Ishikawa, and M. Tokoro, "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," Proc. Int'l Workshop on Reflection and Meta-level Architectures, Tokyo, Japan, Nov. 1992, pp. 36-47.
7. T. Watanabe and A. Yonezawa, "Reflection in an Object-Oriented Concurrent Language," Proc. OOPSLA'88, ACM SIGPLAN Notices, ACM Press, San Diego, California, vol. 23, 1998, pp. 306-315.
8. M. Wegdam et al., "Using Reflection in a Management Architecture for CORBA," Proc. 11th IFIP/IEEE Int'l Workshop on Distributed Systems, Operations & Management (DSOM 2000), IEEE Press, Piscataway, N.J., Dec. 2000, pp. 230-242.
9. H. Duran-Limon and G.S. Blair, "The Importance of Resource Management in Engineering Distributed Objects," Proc. 2nd Int'l Workshop on Engineering Distributed Objects (EDO'2000), Springer-Verlag, Berlin, 2000, pp. 44-60.
10. H. Duran-Limon and G.S. Blair, "Specifying Real-time Behavior in Distributed Software Architectures," Proc. 3rd Australasian Workshop on Software and System Architectures, Monash University, Melbourne, Australia, Nov. 2000, pp. 44-54.
11. F. Costa and G.S. Blair, "Integrating Meta-Information Management and Reflection in Middleware," Proc. of the Int'l Symposium on Distributed Objects and Applications (DOA'00), IEEE Press, Piscataway, N.J., Sept. 2000, pp. 133-143.
12. K. Saikoski, G. Coulson, and G.S. Blair, "Configurable and Reconfigurable Group Services in a Component Based Middleware Environment," Proc. Int'l SRDS Workshop on Dependable System Middleware and Group Comm. (DSMGC'2000), 2000.
13. M. Clarke et al., "An Efficient Component Model for the Construction of Adaptive Middleware", To appear in Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware'01), Heidelberg, November 2001.
14. G. Coulson, "A Distributed Object Platform Infrastructure for Multimedia Applications," Computer Comm., vol. 21, no. 9, July 1998, pp. 802-818.

15. G. Coulson et al., "The Design of a Highly Configurable and Reconfigurable Middleware Platform", Lancaster University, Technical Report, MPG-01-04, Also submitted to Distributed Computing, ACM/ Springer-Verlag, 2001.
16. Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems, Springer-Verlag, New York, 1992.
17. G.S. Blair et al., "Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform," IEE Proceedings Software, vol. 147, no. 1, Feb. 2000, pp. 13–21.
18. G.S. Blair et al., "The Role of Software Architecture in Constraining Adaptation in Component-Based Middleware Platforms," Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms and Open Distributed Processing (Middleware 2000), ACM Press, New York, 2000, pp. 164-184.
19. T. Fitzpatrick et al., "Design and Application of TOAST: an Adaptive Distributed Multimedia Middleware Platform," To appear in Proc. 8th Int'l Workshop on Interactive Distributed Multimedia Systems (IDMS'01), Springer-Verlag, Lancaster, September 2001.

Gordon Blair is a member of the Distributed Multimedia Research Group at Lancaster University. Following the completion of his PhD at Strathclyde University, he moved to Lancaster in 1983. He currently holds a chair in distributed systems at this university, and is also an Adjunct Professor at the University of Tromsø in Norway. He has published over 150 papers in his field and in on the programme committees of many major international conferences in distributed systems and multimedia. He is on the steering committee of the Middleware series of conferences and was General Chair for this conference in 1998. He also co-organised a workshop on Reflective Middleware (with Roy Campbell, University of Illinois) held in conjunction with Middleware'2000). He is a member of the ACM and affiliate member of the IEEE. Contact him at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; gordon@comp.lancs.ac.uk.

Geoff Coulson is a Senior Lecturer at Lancaster University, a position he has held since gaining his PhD from Lancaster in 1993. His current research interests are in the area of distributed systems, with the engineering of middleware systems being a prime speciality. He is also interested in the application of reflective middleware techniques to network control and management. Contact him at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; geoff@comp.lancs.ac.uk.

Anders Andersen is a faculty member of the Department of Computer Science at the University of Tromsø (since November '99). He obtained his M.Sc in computer science in 1991 from the University of Tromsø. After working at Hedmark College as a lecturer, he returned to Tromsø in 1995. Between 1995 and 1999 he was working as a researcher at NORUT IT. In 1997 he was a visiting student at the Distributed Multimedia Research Group at Lancaster University. In 1998 he was engaged as a visiting researcher in the same group. His main interests are adaptable middleware, support for continuous media, quality of service management, and component-based application server platforms. He is a member of IEEE and ACM. Contact him at the Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway; aa@computer.org.

Lynne Blair is currently a lecturer in the Computing Department at Lancaster University. She completed her PhD at the same university in 1995. Following this, she was a post-doctoral researcher before taking up her current post in January 1999. Her research interests include the use of formal specification in distributed real-time systems, feature interaction beyond traditional telephony applications, and approaches to separation of concerns (including aspect-oriented programming and

specification). Contact her at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; lb@comp.lancs.ac.uk.

Michael Clarke is a Senior Research Associate on the Open ORB project at Lancaster University. He holds a BSc and PhD both attained while studying at Lancaster. His research interests are system oriented and have included networking and operating systems. His PhD thesis explored the principled dynamic reconfiguration of operating system kernel services. Building on this work, he is now involved with designing and implementing the reflective component model that enables principled dynamic reconfiguration in the Open ORB middleware. Contact him at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; mwc@comp.lancs.ac.uk.

Fabio Costa is currently a Ph.D. candidate in Computer Science at the University of Lancaster, UK, and also an assistant lecturer at the Federal University of Goias (UFG), Brazil. He received his MSc in Computer Science from the State University of Campinas (UNICAMP), Brazil in 1995. His research interests are in the area of dynamic and adaptable middleware platforms and reflective architectures. Contact him at the Instituto de Informatica, Universidade Federal de Goias, Campus II, UFG, Goiania, GO, 74001-970, Brazil. Tel: +55 62 851 1181; fmc@inf.ufg.br.

Hector Duran-Limon is currently a Ph.D. candidate in Computer Science at the University of Lancaster, UK sponsored by the Autonomous National University of Mexico (UNAM). He received his MSc in Computer Science from the UNAM, Mexico in 1996. His research interests include the area of resource management in adaptable middleware platforms and the role of reflection on such kind of platforms. He is also interested in the use of ADLs and AOP techniques for the construction of distributed real-time systems. Contact him at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; duranlim@comp.lancs.ac.uk.

Tom Fitzpatrick is a research associate on the Visual Beans Project at Lancaster University developing component-oriented multimedia middleware for distributed cooperative scientific visualization. He completed his PhD at Lancaster in 1999 as part of the Adapt Project, investigating middleware support for adaptive distributed multimedia applications in a heterogenous mobile environment. Contact him at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; tf@comp.lancs.ac.uk.

Lee Johnston is currently a PhD student at Lancaster University and Core Software Engineer at Venation Ltd. His research interests are reflective, component-based middleware with particular interest in exploiting such techniques to create M:N multicast configurable and reconfigurable toolkits. He is also interested in large scale multicast, customized multicast streaming and software architecture for the construction of generic, reusable code. Contact him at Venation, Aadastral Park, Martlesham Heath, Ipswich, Suffolk, IP5 3RE, UK; lee.Johnston@venation.com.

Rui Moreira has been a teaching assistant at UFP (Fernando Pessoa University), since March 1993. He has also been a researcher at INESC Porto (Institute of Computer and Systems Engineering of Porto) since August 1996. His main research interests include the study of software architecture and architectural style representation for adaptable component-based distributed systems such as distributed Digital Libraries. Mr. Moreira received the M.Sc. degree in Electrical and Computer Engineering - Telecommunications Area - from the Faculty of Engineering of Porto University in 1995. He is currently involved in his Ph.D. at Lancaster University. He is a member of the IEEE. Contact him at Praça 9 de Abril, 349, 4249-004 Porto, Portugal; rjm@inescporto.pt.

Nikos Parlavantzas is a research assistant and PhD candidate in the Computing Department at Lancaster University. He is currently working on Open ORB, a reflective middleware platform, focusing

on the application of component software concepts and techniques to the design and implementation of the platform. His research interests include distributed computing, component-based software engineering and software architecture. He received an M.Sc. degree in Distributed Interactive Systems from Lancaster University and a Dipl.Eng. degree in Computer Engineering and Informatics from the University of Patras, Greece. Contact him at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; parlavan@comp.lancs.ac.uk.

Katia Barbosa Saikoski is an assistant lecturer at the Pontifical University Catholic of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil. She is currently reading for a PhD in Computer Science in the Computing Department, Lancaster University sponsored by CAPES, Brazil. She received her B. Sc. (Electrical Engineering) UFRGS, Brazil in 1994 and her M. Sc. (Computing Science) UFRGS, Brazil in 1989. Her areas of research include group-based computing, distributed object computing and reflection. Contact her at the Department of Computing, Lancaster University, Bailrigg, Lancaster LA1 4YR, UK; saikoski@comp.lancs.ac.uk.

Middleware Platforms

DCE—Distributed Computing Environment, www.opengroup.org/dce

CORBA—Common Object Request Broker Architecture, www.corba.org

DCOM—Distributed Component Object Model,
www.microsoft.com/com/dcom.asp

.NET—<http://msdn.microsoft.com/net>

RMI—Remote Method Invocation,
<http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html>

Jini—www.sun.com/jini

EJBs—Enterprise JavaBeans Technology, <http://java.sun.com/products/ejb>

The Design of the Four Metamodels

The tables present the complete meta-object protocol for Open ORB 2. Tables A and B provide the appropriate MOPs for structural reflection, corresponding to the architectural and interface meta-models respectively (see the main text for definitions of these terms).

Table A. Architectural Meta-Object Protocol.

Operation signature	Description
Operations for introspection	
ObjectGraph get_obj_graph();	Returns the complete representation of the component graph that describes the structure of the base-level configuration.
IDSeq get_internal_components();	Returns a list with the identifiers of the components that constitute the base-level configuration.
BoundComponentSeq get_bound_components(In ID comp_id);	Returns a list with information (component id and interface names) of the all components bound to the one identified as the argument.
IDSeq get_internal_bindings();	Returns a list with the ids of all binding objects that are part of the base-level composition.
ArchStyle get_arch_style();	Returns the architecture style of a composite component.
RuleSeq get_style_rules();	Returns the sequence of rules of the composite architecture.
SymbioSeq get_symbiotic_constraints();	Gets the sequence of dependency constraints associated with the architecture, i.e. constraints between two or more components (such as if one is replaced another must also be replaced).
Operations for reconfiguration	

<pre>Void local_bind(In ID interf_id_1, In ID interf_id_2);</pre>	<p>Establish a local binding between the two identified interfaces.</p>
<pre>Void break_local_bind(In ID interf_id_1, In ID interf_id_2);</pre>	<p>Break the local binding between the two interfaces.</p>
<pre>Void insert_component(In OpenORB::RepositoryId new_comp_type, in Name new_comp_name, in InsertLocation location);</pre>	<p>Create and insert a new component into the base-level configuration, with the given name and in the specified location (given by zero or more interfaces to which the new component should be bound; if zero interfaces are given, the new component is left unbound).</p>
<pre>Void remove_component(In ID comp_id, In LBindSeq rebind_mapping);</pre>	<p>Delete the component from the configuration, re-binding the adjacent interfaces of neighbouring components, if appropriate and according to the given mapping of interfaces to be rebound.</p>
<pre>Void replace_component(In ID old_comp_id, In Name new_comp_name, In OpenORB::RepositoryId new_comp_type);</pre>	<p>Replace an existing component with a new component of the given type (the old component is deleted).</p>

Void expose_interf(In Name ext_interf_name, In ID interf_exposer_comp, In Name exposed_interf);	Map the interface of an internal component as a new interface of the composite component.
Void init_arch_transaction();	Creates the boundary for a new set of modifications that will be introduced in the configuration graph (starts the transaction).
Void commit_arch_transaction();	Completes the transaction.
Void rollback_arch_transaction();	Rolls back the transaction.
Void set_arch_style(in ArchStyle style);	Sets the architecture style which constraints the configuration.
Void add_rule(in Rule rule);	Inserts a new rule in the style constraints.
Void remove_rule(in string rule);	Removes a style rule from the style constraints.
Void change_rule(in string rule, In Rule new_rule)	Removes the specified rule and adds the new rule.
Void set_symbiotic_constraint(In NameSeq comps, In Prop property);	Sets a dependency constraint for a given architecture.

Table B. Interface Meta-Object Protocol (introspection only).

Operation signature	Description
---------------------	-------------

IDSeq get_interfaces();	Return the identifiers of all the interfaces supported by a component.
InterfStyle get_interf_style(In ID interf_id);	Return the style of the identified interface (i.e. either operational, stream or signal).
AttrSeq get_attr_list(In ID interf_id);	Return a list with the names and types (typecodes) of all attributes in the identified interface.
NameSeq get_interaction_list(In ID interf_id);	Return a list with the names of all interactions (either operations, flows or signals) in the identified interface.
InteractionDescription Get_interaction_description(In ID interf_id, In Name interaction);	Return the full description of the named interaction (operation, flow or signal) of the identified interface.
Any get_attribute_value(In ID interf_id, In Name attr_name);	Return the actual value of the named attribute in the current instance of the identified interface.
Void set_attribute_value(In ID interf_id, In Name attr_name, In any new_attr_value);	Set the value of the named attribute in the current instance of the identified interface.

<pre>Void call_operation(In ID interf_id, In Name op_name, In ArgValueSeq args);</pre>	<p>Enables a dynamic call to be made to an operation of the identified interface. (Does not work for stream and signal interfaces.)</p>
---	---

Tables C and D provide the appropriate MOPs for behavioural reflection, corresponding to the interception and resources meta-models respectively (again, see the main text for definitions of these terms).

Table C. Interception Meta-Object Protocol.

Operation signature	Description
<pre>Void add_pre_interceptor(In InterceptorDescr interceptor, In Name interceptor_name);</pre>	<p>Add an interceptor to the base-level interface, in order to trap incoming messages and introduce additional behavior to be executed before the interaction is actually processed.</p>
<pre>Void add_post_interceptor(In InterceptorDescr interceptor, In Name interceptor_name);</pre>	<p>Add an interceptor to the base-level interface, in order to trap incoming messages and introduce additional behavior to be executed after the interaction is actually processed.</p>
<pre>Void del_interceptor(In Name interceptor_name);</pre>	<p>Remove the named interceptor from the base-level interface.</p>

Table D. Resources Meta-Object Protocol.

Operation signature	Description
<p>Operations on Factories, Managers and Abstract Resources</p>	

Abstraction getLL();	Get the lower level abstraction of this entity.
Void setLL(In Abstraction abst);	Set the lower level abstraction of this entity.
Abstraction getHL();	Get the higher level abstraction of this entity.
Void setHL(In Abstraction abst);	Set the higher level abstraction of this entity.
Operations on Factories	
Resource newResource(In Size size, In Policy mgntPolicy, In Param schedParam);	Create an abstract resource of a given size and associates a management policy with it; scheduling parameters are passed in case of the creation of processing resources.
Resources getResources();	Get the resources created by this factory.
Operations on Managers	
Int setPolicy(Policy policy);	Set the management policy of this manager.
Int admit(In ResourceAmount amount);	Perform admission control test for a resource request.
Int reserve(In ResourceAmount amount);	Reserve amount of resources.
Int expel(In ResourceAmount amount);	Liberate an amount of reserved resources.
Resources getResources();	Get resources mapped or multiplexed by this manager.

Int addResource(In Resource resource);	Add resource for being mapped or multiplexed by this manager.
Int removeResource(In Resource resource);	Remove this resource so that it is not longer mapped or multiplexed by this manager.
Operations on Schedulers	
Void suspend(In Job job);	Suspend an abstract processing resource.
Void resume(In Job job);	Resume an abstract processing resource.
Int schedule(Int unitOfExec) ;	Determine the order of execution of processing resources.
Operations on Abstract Resources	
Manager getManager();	Get the manager of this resource.
Void setManager(In Manager newMgr, In Param schedParam);	Set the manager of this resource.
Factory getFactory();	Get the factory of this resource.
Void setFactory(in Factory fact);	Set the factory of this resource.
Operations on Jobs	
Int run(In Operation op, In Parameters param, Out Result result);	Execute an operation associated with given parameters and provide the result of the operation.
SchedParam getSchedParam();	Get the scheduling parameters of this job.

```
Int setSchedParam(In  
SchedParam param);
```

Set the scheduling parameters of this job.

Related Work

Pioneering work in the general area of reflection in distributed systems was carried out by Jeff McAffer,¹ who developed the CodA platform, which features extensive support for behavioral reflection through the reification of several facets of communication. There is also now a growing corpus of work in the area of reflective middleware. For example, researchers at the University of Illinois at Urbana-Champaign have experimented with the use of reflection to introduce more dynamic reconfigurability into the Tao middleware platform (dynamicTao).² They achieve this reconfigurability through the use of configurators that maintain dependencies between components and provide a set of hooks for the attachment or detachment of components dynamically. They are also currently interested in configurability of platforms for mobile devices (the LegORB Project).³ In associated work, they have developed a task control model to support QoS management in their platforms.⁴ In general, their approach takes a fairly coarse-grained view of reflection by supporting the customization of key parts of the platform.

Researchers at Trinity College Dublin have investigated the use of a reflective language, Iguana, to develop a more open and extensible middleware platform.⁵ The resultant platform can then be accessed and modified using the reflective facilities offered by the language. Ongoing experiments are also investigating the use of reflection in the development of a minimal CORBA implementation.⁵ Like our work, this approach is based on the use of reflection together with a component model. OpenCorba, developed by researchers at the Ecole des Mines de Nantes, is another example of an open, dynamically adaptable ORB that depends on a reflective language (NeoClasstalk).⁶ Researchers at APM have developed an experimental middleware platform called FlexiNet,⁷ which allows the programmer to tailor the underlying communications infrastructure by inserting and removing layers. Their solution is, however, language-specific; applications must be written in Java. Other middleware platforms featuring aspects of reflection include QuO⁸ and Tao.⁹

Our design has been influenced by several specific reflective languages and systems. We derived the concept of multimodels from AL/1-D. However, the underlying models of AL/1-D are quite different; the language supports six models, namely operation, resource, statistics, migration, distributed environment, and system.¹⁰ This resource model supports scheduling and

garbage collection of objects (but in a relatively limited way compared to our approach). In addition, our architecture metamodel is similar to architectural reflection as proposed by Walter Cazzola and colleagues.¹¹ In their approach, architectural reflection is decomposed into topological reflection, which involves the manipulation of structure (in terms of components and connectors), and strategical reflection, which involves the manipulation of behavior (as a set of rules). They do not explicitly address the issue of architecture integrity, but rather leave this to the designer of the behavioral rules. Similarly, they do not consider other forms of reflection.

Our use of component graphs is inspired by researchers at JAIST in Japan.¹² In their system, they handle adaptation through the use of control scripts written in TCL. Although related to our proposals, the JAIST work does not provide access to the internal details of communication components. Furthermore, the work is not integrated into a middleware platform. The designers of the VuSystem¹³ and Mash¹⁴ advocate similar approaches. Microsoft's ActiveX software¹⁵ also uses component graphs. This software, however, does not address distribution of component graphs. In addition, the graph is not reconfigurable during the presentation of a media stream.

1. J. McAffer, "Metalevel Architecture Support for Distributed Objects," Proc. Reflection 96, Dept of Information Science, Tokyo University, 1996, pp 39-62.
2. F. Kon et al., "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," Proc. of the IFIP/ACM Int'l Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), ACM Press, New York, Apr. 2000, pp. 121-143.
3. M. Roman, F. Kon, and R.H. Campbell, "Reflective Middleware: From the Desk to your Hand," IEEE Distributed Systems Online, 2001, vol. 2, no. 5, 2001, http://dsonline.computer.org/0105/features/rom0105_1.htm (current 28 June).
4. B. Li and K. Nahrstedt, "Dynamic Reconfiguration for Complex Multimedia Applications," Proc. IEEE Int'l Conf. on Multimedia Computing and Systems (IEEE Multimedia Systems 99), IEEE Press, Piscataway, N.J., 1999, pp. 165-170.
5. J. Dowling and V. Cahill, "Building a Dynamically Reconfigurable minimumCORBA Platform with Components, Connectors, and Language-Level Support," Proc. Workshop on Reflective Middleware (RM 2000), ACM Press, New York, Apr. 2000.
6. T. Ledoux, "OpenCorba: A Reflective Open Broker," Proc. Reflection 99, Springer-Verlag, New York, vol. 1616, 1999, pp. 197-214.
7. R. Hayton and A. Herbert, "FlexiNet: A Flexible Component-oriented Middleware System," In S. Krakowiak, S. Shrivastava (Eds.), Advances in

- Distributed Systems - Advanced Distributed Computing: From Algorithms to Systems, LNCS, vol 1752, Springer-Verlag, Heidelberg, 2000, pp. 497 ff.
8. J. Zinky, D. Bakken and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," Theory and Practice of Object Systems, John Wiley and Son, vol. 3, no. 1, Jan. 1997, pp 1-20.
 9. W. Cazzola et al., "Rule-based Strategic Reflection: Observing and Modifying Behavior at the Architectural Level," Proc. 14th IEEE Int'l Conf. Automated Software Engineering (ASE 99), IEEE Press, Piscataway, N.J., Oct. 1999, pp. 263-266.
 10. H. Okamura, Y. Ishikawa, and M. Tokoro, "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework," Proc. Int'l Workshop on Reflection and Meta-level Architectures, Tokyo, Japan, Nov. 1992, pp. 36-47.
 11. N. Wang et al., "Toward an Adaptive Reflective Middleware Framework for QoS-Enabled CORBA Component Model Applications," IEEE Distributed Systems Online, vol. 2, no. 5, 2001, http://dsonline.computer.org/0105/features/wan0105_1.htm (current 28 June).
 12. A. Hokimoto, K. Kurihara and T. Nakajima, "An Approach for Constructing Mobile Applications using Service Proxies," Proc. 16th Int'l Conf. on Distributed Computing Systems (ICDCS 96), IEEE Press, Piscataway, N.J., May 1996, pp. 726-733.
 13. C.J. Lindblad and D.L. Tennenhouse, "The VuSystem: A Programming System for Computer-Intensive Multimedia," IEEE J. of Selected Areas in Communications, vol. 14, no. 7, 1996, pp. 1298-1313.
 14. S. McCanne et al., "Towards a Common Infrastructure for Multimedia-Networking Middleware," Proc. 7th Int'l Conf. on Network and Operating System Support for Digital Audio and Video (Nossdav 97), St. Louis, Missouri, May 1997, pp. 39-49.
 15. Microsoft, "Microsoft ActiveX," www.microsoft.com/com/tech/activex.asp (current 28 June 2001).

