THE DESIGN AND IMPLEMENTATION OF OPEN ORB V2


Distributed Multimedia Research Group,
Department of Computing,
Lancaster University,
Bailrigg, Lancaster,
LA1 4YR, U.K.
Tel: +44 1524 65201
Fax: +44 1524 593608
E-mail: reflection@comp.lancs.ac.uk

ABSTRACT

Middleware has emerged as an important architectural component in modern distributed systems. However, it is now recognised that established middleware platforms such as CORBA and DCOM are not flexible enough to meet the needs of the emerging distributed applications, featuring for example access to multimedia services and also support for mobile users. Rather, we argue that next generation middleware platforms should be i) more configurable, ii) dynamically reconfigurable, and iii) should support the longer term evolution of the design of the platform. This paper discusses the design of the Open ORB v2 middleware platform, which addresses these requirements through an approach based on reflection together with the use of component technology. Important issues addressed in the paper include the use of architectural constraints to maintain the integrity of reflective middleware platforms, together with studies of efficient implementation of such technology. The capabilities of the Open ORB platform are also demonstrated through a series of application-level experiments.

1. INTRODUCTION

Middleware has emerged as an important architectural component in modern distributed systems. The role of middleware is to offer a high-level, platform-independent programming model to users (e.g. object-oriented or component-based) and to mask out problems of distribution. Examples of key middleware platforms include DCE, CORBA, DCOM, .NET, and the Java-based series of technologies (RMI, JINI, EJB). Traditionally, such platforms have been deployed (with considerable success) in application domains such as banking and finance as a means of tackling problems of heterogeneity, and also supporting the integration of legacy systems. However, more recently, middleware technologies have been applied in a wider range of areas including safety critical systems, embedded systems, and real-time systems. In addition, middleware is required to respond to emerging technical challenges as offered for example by multimedia or mobile computing. It is now becoming apparent however that middleware technologies such as the ones listed above are not able to respond to such diversity or to such technical challenges. The main reason for this is the black-box philosophy of existing platforms. In particular, existing middleware platforms offer a fixed service to their users, and it is not possible to view or alter the implementation of this service. Inevitably, the architecture of this platform then represents a compromise design featuring,

1 Also of the University of Tromsø, Norway.
2 Also of UFP (Universidade Fernando Pessoa) and INESC-Porto.
for example, general-purpose protocols and associated management strategies. It is then not possible to specialise platforms to meet the needs of more specific target domains.

Middleware designers are aware of this problem and have responded with a number of initiatives. Focussing on CORBA for example, the OMG have introduced a series of platform specifications including real-time CORBA and Minimal CORBA. These are however specific solutions to specific domains and are not a general solution to this problem. Modern middleware platforms also typically offer more flexibility through mechanisms such as interceptors and configurable protocol stacks. These are important developments but, in our opinion, they do not offer a complete solution to the problem. Rather, we believe that next generation middleware platforms should have the following properties:

1. they should be *configurable* to meet the needs of a given application domain (e.g. to provide a minimal configuration for embedded systems),

2. they should be *dynamically reconfigurable* to enable the platforms to respond to changes in their environment (e.g. to respond to dramatic fluctuations in network quality of service as experienced in mobile computing), and

3. they should support the *evolution* of the design of the platform as requirements change over time (e.g. to respond to the need to support continuous media interactions).

Recently, a number of *reflective middleware* technologies have emerged in response to such requirements. Reflection is a technology that has previously been deployed successfully in the design of programming languages and operating systems (amongst other areas). The key to the approach is to offer a meta-interface supporting the inspection and adaptation of the underlying virtual machine. In terms of middleware, this implies that the meta-interface should support operations to discover the internal operation and structure of the middleware platform (e.g. protocols and management structures being deployed) and to make changes at run-time. 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 (we return to this crucial issue later in the paper).

This paper presents the design and implementation of *Open ORB*, a reflective middleware platform developed at Lancaster University. More specifically, the paper focuses on Open ORB v2, a significant re-design building on our experiences from the use of our first implementation of the platform (descriptions of this earlier version of the platform can be found in the literature [Blair98a, Costa98, Costa00a, Andersen00]). The paper presents a broad view of our work ranging from studies of reflective architecture, through studies of efficient implementation, to a series of validation experiments for the technology.

The paper is structured as follows. Section 2 discusses the overall architecture of Open ORB, highlighting the complementary role of reflection and component technologies, and also the use of a multi-model approach. Section 3 then presents some work on the efficient implementation of Open ORB, using an extension to the COM component model (Open COM). Following this, section 4 describes a variety of application experiments intended to validate the architecture. These experiments demonstrate the capabilities of the architecture with respect to each of the properties introduced above (configurability, dynamic reconfiguration and evolution). Related work in the area of reflective middleware is discussed in section 5, with some overall conclusions drawn in section 6.
2. THE OPEN ORB ARCHITECTURE

2.1. Overall Approach

The Open ORB architecture builds on two complementary technologies namely *components* and *reflection*. Component technologies are gaining widespread acceptance in the middleware community, as witnessed by the popularity of COM and JavaBeans, and also the imminent publication of the component-based CORBA v3 specification. For the purposes of this paper, we define a component to be “a unit of composition with contractually specified interfaces and explicit context dependencies only” [Szyperski98]. In addition, a component “can be deployed independently and is subject to third-party composition” [Szyperski98]. The middleware technologies mentioned above adopt a component-based programming model to enhance the configurability, reconfigurability and level of re-use in applications. We extend this capability to the design of the middleware platform itself. In particular, an instance of Open ORB is a particular configuration of components, which can be selected at build-time and reconfigured at run-time. For example, we can opt for a minimal configuration of components, perhaps offering only client-side capabilities, to run on a PDA. Similarly, we can replace a particular protocol component at run-time should network conditions change significantly.

Note that our particular component model is heavily influenced by previous work on the Sumo Project at Lancaster [Blair98b], and also by the Computational Model from RM-ODP (again see [Blair98b]). In particular, the component model is designed to support multimedia programming. The main features of our component model are: i) components are described in terms of a set of *required* and *provided* interfaces, ii) interfaces for continuous media interaction are also supported, iii) *explicit bindings* can be created between compatible interfaces (the result being the creation of a *binding component*), and iv) components offer a built-in *event notification facility*.

Access to the underlying platform, and by implication the associated component structure, is provided by reflection. In particular, every application-level component offers a *meta-interface* providing access to an underlying *meta-space* which is in effect the support environment for this component (cf the middleware platform). Crucially, meta-space is itself composed of components. Such (meta-level) components also have a meta-interface, offering access to their support environment. This approach is therefore recursive, leading to an *infinite tower* of reflection. In order to overcome this, meta-components are instantiated on demand; unless accessed, they exist in theory but not in practice.

In our design, meta-space is partitioned into a number of distinct meta-space models. This approach was first advocated by the designers of AL-1/D [Okamura92]. The benefit of this approach is to simplify the interface offered by meta-space by maintaining a separation of concerns between different system aspects. This is particularly important in distributed systems given the wide range of concerns that must be considered (in comparison to the design of a single programming language for example). The structure of meta-space is captured by figure 1 below.
We consider this structure in more detail below.

2.2. The Meta-Space Models

2.2.1. Supporting Structural Reflection

In reflective systems, structural reflection is concerned with the content and structure of a given component [Watanabe88]. In our architecture, this aspect of meta-space is represented by two distinct meta-models, namely the interface and architecture meta-models. The two meta-models represent a separation of concerns between the external view of a component (i.e. its set of interfaces), and the internal construction (i.e. its software architecture). We introduce each model in turn below.

The interface meta-model provides access to the external representation of a component in terms of the set of provided and required interfaces. In particular, 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 meta-model 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 [Blair98a], 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 referred to above between interface and implementation, in keeping with this important principle of component-based programming. Important benefits of this approach will emerge from the discussion below.

The architecture meta-model then provides access to the implementation of the component as a software architecture, consisting of two key elements: a component graph and an associated set of architectural constraints (cf components frameworks as introduced in section 3.3 below). 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 (cf connectors in the software architecture literature). An extensible set of binding types can be supported offering interaction models such as remote invocation, publish/subscribe, continuous media flows, group communication, etc. Normally this structure would be hidden from a user of a component. However, the architecture meta-model can be used to both discover and also make changes to
this structure at run-time.

If unconstrained, this is a rather dangerous approach to advocate. Consequently, we extend the software architecture to include a set of architectural constraints. The type management system offers one level of constraints, i.e. a new component must be a valid substitution of the old component (cf subtyping of the respective interfaces). This is however not enough; 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 may be necessary to preserve a given architectural style over time such as pipes-and-filters. Our approach is to record such constraints explicitly in the architecture and to ensure that adaptations preserve the architectural rules before committing the changes (cf atomic transactions). At design time, we adopt a combination of UML, OCL and natural language to capture such constraints. At run-time, we are investigating two alternative approaches (see sections 3.3 and 4.2):

1. An implicit approach whereby constraints are captured indirectly by the interface offered by an architecture (i.e. only certain safe operations are permitted);
2. An explicit approach where constraints are encoded using an appropriate notation.

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

Note that the approach described above is applied recursively in that components within a component graph may themselves have architecture, accessed via its architecture meta-model (i.e. at a meta-meta-level relative to the uppermost component. For example, a binding component within a graph may itself have a structure consisting of stubs and protocol components. This recursion terminates with primitive components, which 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 (e.g. a reflective language may enable access to methods), and hence this is deemed beyond the scope of a (language-independent) middleware architecture.

Note also that with this approach interfaces are immutable, and represent irrevocable contracts with their environment. In other words, it is not possible to change the implementation behind an interface. Evolution is supported by the addition of new components into an architecture supporting upgraded interfaces. Old clients can however still rely on the previous interfaces to offer both functional and indeed their non-functional guarantees. This also considerably simplifies the task of type management as discussed in section 2.3 below.

A further benefit of the new design is that strict controls can be placed on access rights for adaptation. More specifically, all classes of users can be given rights to access the interface meta-model. In contrast, rights to the architecture meta-model can be tightly constrained so that only trusted third parties can modify the architecture of the system at run-time.

The interfaces (or meta-object protocols) for the interface and architecture meta-models can be found in appendix A.

2.2.2. Supporting Behavioural Reflection

Behavioural reflection is concerned with activity in the underlying system [Watanabe88]. More specifically, the design of 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 meta-models respectively.
The interception meta-model is arguably the most straightforward in the Open ORB design, and is a simplified version of the environmental meta-model from the first version of the architecture. In keeping with a number of reflective middleware proposals, this meta-model enables the dynamic insertion of interceptors. Such interceptors are associated with interfaces (more specifically, local bindings) and enable the insertion of pre- and post-behaviour. This applies equally to all styles of interface supported in Open ORB (operational, continuous media, etc). This mechanism is useful, for example, to dynamically introduce monitoring or accounting into a running system [Wegdam00]. Similarly, interceptors can be used to introduce additional non-functional behaviour, such as security checks or concurrency control.

The resources meta-model in contrast is quite unique to the Open ORB design, offering access to underlying resources and resource management [Duran-Limon00a]. We strongly 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 structure of the system, e.g. when now operating in a mobile environment.

The resources meta-model is based around the abstractions of resources and tasks. Resources can be either primitive (e.g. raw memory or OS threads) or complex (e.g. buffers or user-level threads multiplexed on to kernel-level threads). They are created by resource factories and managed by resource managers, 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. For example, there could be a single task dealing with the arrival, filtering and presentation of an incoming video stream, or alternatively this could be divided into a number of smaller tasks. Important, tasks can span component boundaries and are thus orthogonal to the structure of the system. Tasks are essentially the unit of resource allocation, i.e. tasks have a pool of resources to support their execution.

There is a resources meta-model per address space, i.e. resources are associated with a particular address space and all components within that address space share the same meta-model. The meta-model provides access to a set of components representing resource management. As with other meta-models, it is then possible to either inspect or adapt activity associated with resources. For example, it is possible to 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 meta-model is accessed as a graph structure that organises resources, tasks and managers into hierarchical structures.

As an extension to this work, we are currently experimenting with an enhanced architectural description language (ADL) called Xelha [Duran-Limon00b], building on the task model described above. In common with many ADL’s, 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 (QoS) 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. An example of the use of Xelha is given below (omitting dynamic QoS management features):

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

3 This aspect of the work is being developed along with our collaborators at France Telecom R&D. In particular, they have contributed to the resource/task model presented in this section.
This example is of a composite audio connector, consisting of source and sink stubs interconnected by a stream connector. The early parts of this IDL are fairly traditional and hence we do not comment on them further. The task section however defines the task structure and its associated QoS requirements (used to derive resource allocations as discussed above. In this case, 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 this is omitted for brevity. The boundaries between tasks are defined by task switching points, i.e. points where the system will switch to alternative pools of resources, e.g. different set of threads.

Again, the meta-object protocols for behavioural reflection in Open ORB can be found in appendix A.

2.3. Extensions

Open ORB v2 contains two further enhancements to the previous version:

1. An integrated approach to meta-information management. Meta-information management in Open ORB v2 centres on the Open ORB type repository. This is an extension to the CORBA interface repository, but with additional features to support our component model (e.g. stream interfaces, QoS annotations, media types, and primitive and composite components and bindings). The kind of meta-information managed by the type repository includes both the type and template aspects that describe the elements of the component model. The repository therefore provides support for type-checking and for the definition and instantiation of platform configurations. The model of the type repository (or more precisely its meta-model) is defined in terms of the CORBA Meta-Object Facility (MOF), thus achieving a level of integration between this intensional style of reflection (relating to the type system) and the more extensional styles already supported by Open ORB. The use of MOF technology also introduces the additional capability of being able to evolve the type system over time (although, to date we have not explored this aspect in any depth).

We have also achieved a level of integration between meta-information in the type repository and the typing and template information in the interface and architecture meta-space models. This task is simplified considerably by the immutability of
interfaces in Open ORB. However, components and bindings can still change, e.g. by featuring a new interface or by adapting the internal architecture. This introduces the need to manage type evolution. Essentially, the type repository becomes the source of information on all stable (i.e. published) types (and templates) in the system, whereas the structural meta-spaces become the source of the meta-information (type and template) that describes the current (possibly evolved) structure of components. Eventually, such meta-information can also be turned into proper types and published in the repository for future use.

2. The incorporation of group services. Communication capabilities in the platform have been extended to include configurable and reconfigurable group services through the provision of a group factory. Both open and closed group services are supported. The basic group factory supports a single method, i.e. `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 (e.g required protocol stacks, filters, etc). 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, collation, etc. The resultant group service can be accessed using the various meta-models defined above. For example, the interface meta-model can be used to discover the external interfaces offered by a group. Similarly, the architecture meta-model can be used to adapt the implementation at run-time (subject to the normal architectural constraints).

Further details of these important extensions can be found in [Costa00b] and [Saikoski00] respectively.

3. EFFICIENT IMPLEMENTATION

3.1. Overall Approach

The architecture described above has evolved through a series of prototypes written in the scripting language Python. Python was a natural choice for this prototyping work given the intrinsic support for rapid prototyping and also the underlying reflective capabilities of the language. Nevertheless, given the interpreted nature of this language, 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 stated goal of this work is to have a reflective middleware platform that (in standard configuration) performs at least as well as commercial ORBs but with the additional benefits of reflective middleware. This is of course a worst case scenario for the technology as reduced configurations should exhibit much better performance.

In more detail, the approach adopted is to define a base reflective component model, Open COM, as an extension to Microsoft’s COM architecture, and then to use this to implement a component-based middleware platform (i.e. the middleware platform has itself a component-based architecture). We look at Open COM and the associated middleware implementation in turn below.
3.2. The Open COM Component Model

As mentioned above, Open COM is closely based on Microsoft’s COM but enhanced with richer reflective facilities. Open COM relies only on the core of COM, i.e. 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; it avoids dependencies on other features of COM such as distribution (via DCOM), persistence, security and transactions. Crucially, we do 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 term 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 run-time and consequently means that COM components cannot be dynamically reconfigured. In our model, we define the receptacle data structure as a first class run-time 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, e.g. locks, to allow the system to prevent invocations through a receptacle when a reconfiguration on the connection is taking place.

Open COM then provides low level support for our meta-models as follows:

1. 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);

2. The IMetaArchitecture interface provides access to the underlying graph structure of components and their connections (assuming the component is not primitive);

3. The IMetaInterception interface enables the dynamic attachment or detachment of interceptors as defined above.

Note that this is only a subset of our reflective features defined above. For example, we do not offer a resources meta-model at this level; rather we expect this to be constructed above this level (in the Open ORB implementation). Similarly, we only offer a partial implementation of the architecture meta-model; again, it is assumed that architectural constraints are introduced at a higher level.
The architecture of Open COM is summarised in figure 2 above.

3.3. The Middleware Architecture

The Open COM component model has been used 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 design is heavily influenced by the GOPI platform, previously developed at Lancaster [Coulson98].

The implementation exploits the concept of component frameworks in its construction. A component framework is defined as a collection of rules and contracts that govern the interaction of a set of components [Szyperski98]. The main motivation for such component frameworks is to constrain the design space and the scope for evolution. Moreover, they simplify component development and assembly, enable lightweight components and increase the understandability and maintainability of the system.

Component frameworks have often been used as a design time concept. We also adopt this approach at run-time. In our approach, component frameworks define architectures (component graphs and constraints) for specific domains and provide (partial) support for our architecture meta-model. In particular, our component frameworks are explicitly represented as components (called component framework representatives or CFRs), that are responsible for implementing the architecture meta-interfaces while enforcing the architectural constraints. This is achieved using the implicit approach to capturing constrains as defined in section 2.2.1). The middleware architecture is then decomposed into an extensible set of specialised and focused domains of concern, such as buffer management and binding establishment each based on a component framework /CFR.

More specifically, the middleware architecture is organised into three layers as shown in figure 3 (a more fine-grained view of the implementation is given in appendix B).
The binding layer contains the binding component framework that accepts a variety of binding type implementations (as discussed below). The communications layer then contains the protocol component framework. Within this framework, a reconfiguration manager maintains information about the current protocol stack (organised as a component graph), which can then be adapted using the architecture meta-interface; the component framework will ensure that the overall integrity of the protocol structure is maintained during reconfiguration. This layer can be configured to include additional component frameworks, e.g. a multimedia streaming framework featuring a filter style. Currently, we are developing a component framework for multiparty communication protocols to support the group service described in section 2.3 above (the latter would appear as a specific binding type). At the lowest level, the resources layer has a number of component frameworks for buffer, transport and thread management. Again, adaptation can be tailored for the particular domain, e.g. 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 the most interesting feature of this architecture. 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/coordination aspects from computational aspects and hence simplify applications development. They also effectively implement software architecture connectors, thus bridging 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, etc) designed to capture commonalities across diverse binding types. It does not attempt to specify a uniform interface to all binding types (this is clearly infeasible) but rather it offers guidelines and generic interfaces which provide consistency for binding users and guidance for binding type implementers. The configuration of binding types can be configured statically and also changed at run-time 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 paper [Coulson01].

Overall, the implementation of the platform consists of 6 component frameworks, around 25 Open COM components and in total around 50,000 lines of C++ code (including support for audio and video streaming). A thorough investigation of performance is currently being carried out and will be reported in the above forthcoming paper [Coulson01]. Nevertheless,
early figures indicate that we will comfortably meet our target of outperforming commercial CORBA implementations. For example, the following figure provides an indication of the relative performance of the platform against Orbacus (one of the fastest commercially available ORBs) and GOPI (see figure 4).

![Figure 4: Performance measurements.](image)

As can be seen GOPI performs best for invocations of up to 1024 bytes, then Orbacus takes over as the most efficient. The figures though for Open ORB are extremely encouraging. For smaller invocations the performance of Open ORB and Orbacus are practically identical, and it is only at larger invocation sizes that a (small) difference becomes apparent. This difference clearly comes from the underlying GOPI implementation strategies (also used in Open ORB) rather than any overhead of reflection. A comparison of Open ORB and GOPI also provides a first approximation of the overheads of reflection in Open ORB. This comes out at around a 10% difference between GOPI and Open ORB in terms of throughput of invocations. Again, 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, e.g. by removing unnecessary components at run-time or introducing more specialist components for given network conditions).

Note that tests were performed over the loopback interface on a Dell Precision 410 workstation specified with a 550Mhz Intel Pentium III processor and 256Mb RAM. The compiler used was Microsoft's cl.exe with flags /MD /W3 /GX /FD /O2.

4. EVALUATION OF OPEN ORB

4.1. Experiments in Dynamic Reconfiguration

The first experiments with Open ORB focused on the use of reflection to support dynamic reconfiguration. More specifically, we have experimented with adapting the structure of continuous services in response fluctuations in the underlying network quality of service. This has been achieved by the development of a QoS management subsystem for Open ORB offering both monitoring and controlling. In particular, QoS management is achieved by introducing management components into the component configuration
(accessed via the architecture meta-model). Different styles of management component are identified in our architecture (see table 1).

<table>
<thead>
<tr>
<th>Monitoring</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Collector</td>
<td>Observe behaviour of underlying functional components and generate relevant QoS events.</td>
</tr>
<tr>
<td>Monitor</td>
<td>Collect QoS events and report abnormal behaviour to interested parties.</td>
</tr>
<tr>
<td>Control</td>
<td></td>
</tr>
<tr>
<td>Strategy Selectors</td>
<td>Select an appropriate adaptation strategy (i.e. strategy activator) based on feedback from monitors.</td>
</tr>
<tr>
<td>Strategy Activators</td>
<td>Implement a particular strategy, e.g. by manipulating component graph.</td>
</tr>
</tbody>
</table>

Table 1. Styles of management component.

Event collectors and strategy activators are at the lowest level of this architecture, interfacing directly to 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, e.g. to decide 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. In our prototype system, the policies for monitoring and strategy selection are expressed as timed automata, which then map directly on to management components which then act as timed automata interpreters at run-time. They then interface to other components in the system using event notification, i.e. they register for events of interest, receive events, react to them and then emit events to interested parties (cf reactive objects [Manna92]). This use of timed automata also allows us to carry out formal analysis of the behaviour of the QoS management subsystem in isolation, and also when composed with a model of the rest of the system [Blair01].

Two examples have been implemented using this approach, i.e. a QoS-managed audio streaming application and a synchronisation protocol for stored video (both using the Python-based prototype). Experiments are also underway to extend this work to also allow the dynamic reconfiguration of resources and resource management policies, i.e. exploiting the resources meta-model as described above. 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 particularly heartening that QoS management can be introduced dynamically in our architecture, even if this has not been planned in advance. Furthermore, given the recursive nature of the architecture, it is also possible to monitor and adapt the management components, e.g. checking if a given policy is operating satisfactorily and, if not, changing the policy (effectively providing a level of meta-policy).

Further details of the QoS management architecture and associated experiments can be found in the literature [Blair00a]. A more general treatment of adaptation in Open ORB can also be found in [Blair00b].

4.2. Other Ongoing Experiments

Work is now continuing on a series of experiments to more fully evaluate the architecture, as discussed below:

1. Configurability. 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 in section 2.3 above). We are currently, however, 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) [Gallop00]. In both cases, the longer-term goals are also to support dynamic reconfiguration, broadening our experience in this area.

2. **Longer-term evolution.** We are convinced that the Open ORB architecture also provides interesting support for the longer-term evolution of software, i.e. as requirements change over time. For example, the architecture meta-model 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, enhancing the scalability of the platform, etc. 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 (see section 2.2.1). This approach enables us to evolve the architecture itself, e.g. by modifying the constraints over time.

The experiments described above will provide a relatively complete evaluation of the potential of reflective middleware, in general, and Open ORB, in particular.

5. **RELATED WORK**

As mentioned earlier, there is growing interest in the use of reflection in distributed systems. Pioneering work in this general area was carried out by McAffer [McAffer96], who developed the CodA reflective, distributed object-oriented platform. This platform features extensive support for behavioural reflection through the reification of a number of facets of communication (i.e. send, accept, queue, receive, protocol, execution and state). There is also now a growing corpus of work in the area of reflective middleware. For example, a series of experiments have been carried out at the University of Illinois at Urbana-Champaign. For example, they have experimented with the use of reflection to introduce more dynamic reconfigurability into the Tao middleware platform (dynamicTao) [Kon00]. This is achieved 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) [Roman01]. In associated work, they have also developed a task control model to support QoS management in their platforms [Li99]. In general, the approach at Illinois takes a fairly course grained view of reflection, by supporting the customisation 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 [Dowling00]. 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 [Dowling01]. 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 a further example of an open,
dynamically adaptable ORB that depends on a reflective language (NeoClasstalk) [Ledoux99]. Researchers at APM have developed an experimental middleware platform called FlexiNet [Hayton98], which 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. Other middleware platforms featuring aspects of reflection include QuO [Zinky01] and Tao [Wang01].

Our design has been influenced by a number of specific reflective languages and systems. As stated above, the concept of multi-models was derived from AL/1-D. The underlying models of AL/1-D are however quite different; the language supports six models, namely operation, resource, statistics, migration, distributed environment and system [Okamura92]. From this list, it can be seen that AL/1-D does however support a resources model. This resource model supports reification of scheduling and garbage collection of objects (but in a relatively limited way compared to our approach). In addition, our architecture meta-model is similar to architectural reflection as proposed by Cazzola et al [Cazzola99]. 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 behaviour (as a set of rules). In their work, they do not explicitly address the issue of integrity of architectures, but rather leave this to the designer of the behavioural rules. Similarly, they do not consider other forms of reflection in their work.

Our use of component graphs is inspired by researchers at JAIST in Japan [Hokimoto96]. In their system, adaptation is handled 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 VuSystem [Lindblad96] and Mash [McCanne97] advocate similar approaches. The same criticisms however also apply to their designs. Microsoft’s ActiveX software [Microsoft99] 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.

6. CONCLUSIONS

This paper has considered the design and implementation of Open ORB, an experimental reflective middleware architecture intended to provide more configurability, reconfigurability and support for longer-term evolution. The approach is based on a marriage of component technology and reflection. In particular, through reflection, the component-based approach pervades both the application level and indeed the implementation of the platform itself. Crucially, reflection allows this underlying structured to be accessed and indeed altered dynamically. In more detail, the platform has the following key features:

1. The use of an advanced component model with explicit support for multimedia programming, e.g. through continuous media interfaces and explicit binding;
2. The adoption of a multi-model approach to provide an important separation of concerns for programmers accessing the meta-level of the platform;
3. Support for structural reflection through introspection of component interfaces, and also more general access to the underlying software architecture of the system (in terms of component graphs and architectural constraints);
4. Support for behavioural reflection through the ability to attach interceptors to interfaces, as well as the more novel provision of a resources meta-model.

Two implementations of the platform have been developed, the first using a prototyping language to encourage experimentation with the overall architecture, and the second in C++
(together with a reflective extension to COM) to validate the performance of the approach. In addition, a series of applications have been developed focusing mainly on dynamic reconfigurability, but also now considering other capabilities of the platform. Crucially, this work has addressed two important concerns about reflective middleware platforms, i.e. that such technologies carry a penalty overhead, and that reflective middleware platforms carry risks in terms of the integrity of systems. Our experiences to date indicate that both these concerns can be overcome.

For the future, the most important work is now to reach international consensus firstly on the need for reflective middleware technology and secondly on the interfaces to be offered by such platforms. This is important to support the emergence of a next generation of middleware platforms that firstly are more configurable and reconfigurable, but which also can offer portability and interoperability for applications that choose to exploit such advanced features.

ACKNOWLEDGEMENTS

The research described in this paper is partly funded by France Telecom R&D (CNET Grant 96-1B-239). Particular thanks are due to 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 would like to 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 would also like to acknowledge the contributions of our partners on the CORBAng project (next generation CORBA) at UniK, and the Universities of Oslo and Tromsø (all in Norway). Particular thanks to Frank Eliassen, Vera Goebel, Oyvind Hanssen and Thomas Plagemann.

REFERENCES


[Roman01] Roman, M., Kon, F., Campbell, R.H., “Reflective Middleware: From the Desk to your Hand”, DS Online, This Special Issue, 2001.


# APPENDIX A: THE DESIGN OF THE 4 META-MODELS

## A.1 Structural Reflection

<table>
<thead>
<tr>
<th>Operation signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Architectural MOP</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Operations for introspection</strong></td>
<td></td>
</tr>
<tr>
<td>ObjectGraph get_obj_graph();</td>
<td>Returns the complete representation of the component graph that describes the structure of the base-level configuration.</td>
</tr>
<tr>
<td>IDSeq get_internal_components();</td>
<td>Returns a list with the identifiers of the components that constitute the base-level configuration.</td>
</tr>
<tr>
<td>BoundComponentSeq get_bound_components(in ID comp_id);</td>
<td>Returns a list with information (component id and interface names) of the all components bound to the one identified as the argument.</td>
</tr>
<tr>
<td>IDSeq get_internal_bindings();</td>
<td>Returns a list with the ids of all binding objects that are part of the base-level composition.</td>
</tr>
<tr>
<td>ArchStyle get_arch_style();</td>
<td>Returns the architecture style of a composite component.</td>
</tr>
<tr>
<td>RuleSeq get_style_rules();</td>
<td>Returns the sequence of rules of the composite architecture.</td>
</tr>
<tr>
<td>boolean check_rule_consistency(in Rule rule);</td>
<td>Checks if a given rule is conformant with the architecture style rules in use by an architecture.</td>
</tr>
<tr>
<td>SymbioSeq get_symbiotic_constraints();</td>
<td>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).</td>
</tr>
<tr>
<td><strong>Operations for reconfiguration</strong></td>
<td></td>
</tr>
<tr>
<td>void local_bind(in ID interf_id_1, in ID interf_id_2);</td>
<td>Establish a local binding between the two identified interfaces.</td>
</tr>
<tr>
<td>void break_local_bind(in ID interf_id_1, in ID interf_id_2);</td>
<td>Break the local binding between the two interfaces.</td>
</tr>
<tr>
<td>void insert_component(in OpenORB::RepositoryId new_comp_type, in Name new_comp_name, in InsertLocation location);</td>
<td>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).</td>
</tr>
<tr>
<td>void remove_component(in ID comp_id, in LBindSeq rebind_mapping);</td>
<td>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.</td>
</tr>
<tr>
<td>void replace_component(in ID old_comp_id, in Name new_comp_name, in OpenORB::RepositoryId new_comp_type);</td>
<td>Replace an existing component with a new component of the given type (the old component is deleted).</td>
</tr>
<tr>
<td>void expose_interf(in Name ext_interf_name, in ID interf_exposer_comp, in Name exposed_interf);</td>
<td>Map the interface of an internal component as a new interface of the composite component.</td>
</tr>
<tr>
<td>void init_arch_transaction();</td>
<td>Creates the boundary for a new set of modifications that will be introduced in the configuration graph (starts the transaction).</td>
</tr>
<tr>
<td>void commit_arch_transaction();</td>
<td>Completes the transaction.</td>
</tr>
<tr>
<td>void rollback_arch_transaction();</td>
<td>Rolls back the transaction.</td>
</tr>
<tr>
<td>void set_arch_style(in ArchStyle style);</td>
<td>Sets the architecture style which constraints the configuration.</td>
</tr>
<tr>
<td>void add_rule(in Rule rule);</td>
<td>Inserts a new rule in the style constraints.</td>
</tr>
<tr>
<td>void remove_rule(in string rule);</td>
<td>Removes a style rule from the style constraints.</td>
</tr>
<tr>
<td>void change_rule(in string rule, in Rule new_rule);</td>
<td>Removes the specified rule and adds the new rule.</td>
</tr>
</tbody>
</table>
void set_symbiotic_constraint(
    in NameSeq comps,
    in Prop property);

Sets a dependency constraint for a given architecture.

<table>
<thead>
<tr>
<th>Operation signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDSeq get_interfaces();</td>
<td>Return the identifiers of all the interfaces supported by a component.</td>
</tr>
</tbody>
</table>
| 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. |
| void call_operation(
    in ID interf_id,
    in Name op_name,
    in ArgValueSeq args);                                 | Enables a dynamic call to be made to an operation of the identified interface. (Does not work for stream and signal interfaces.) |

Notes

1. The terms name and id in the above table denote components and interfaces respectively. Names are non-qualified, meaning that they are only meaningful in the context where they are defined (e.g. an interface name in the context of the component that owns the interface), whereas ids are globally unique names given to such entities. In addition, when creating a new component (or interface) only its name is given; the id is generated automatically.
### A.2. Behavioral Reflection

#### Interception MOP

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

#### Resources MOP

<table>
<thead>
<tr>
<th>Operation signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AbstractResource NewResource( in Size size, in Policy mgntPolicy, in Param schedParam);</td>
<td>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.</td>
</tr>
<tr>
<td>int Suspend(in ResourceID abstResource_id);</td>
<td>Suspend an abstract processing resource.</td>
</tr>
<tr>
<td>int resume(in ResourceID abstResource_id);</td>
<td>Resume an abstract processing resource.</td>
</tr>
<tr>
<td>Resources getLLRs();</td>
<td>Get lower level resources.</td>
</tr>
<tr>
<td>int setLLRs(in Resources llrs);</td>
<td>Set lower level resources.</td>
</tr>
<tr>
<td>HigherLevelResources getHLRs();</td>
<td>Get higher level resources.</td>
</tr>
<tr>
<td>int setHigherLevelResources(in Resources hlrs);</td>
<td>Set higher level resources.</td>
</tr>
<tr>
<td>Manager getManager();</td>
<td>Get the manager of this resource.</td>
</tr>
<tr>
<td>int setManager(in Manager newMgr, in Param schedParam);</td>
<td>Set the manager of this resource.</td>
</tr>
<tr>
<td>Factory getFactory();</td>
<td>Get the factory of this resource.</td>
</tr>
<tr>
<td>int setFactory(in Factory fact);</td>
<td>Set the factory of this resource.</td>
</tr>
</tbody>
</table>
APPENDIX B: DETAILED COMPONENT ARCHITECTURE OF OPEN ORB

1) Multipointer receptacle of type IASCPolicy. Currently three implementations (ASCEDF, ASCDEF, ASCNATIVE).
2) Multipointer receptacle of type ITPProtocol. Currently four implementations (TPUDP, TPTCP, TPIPC, TPMIP).
3) Multipointer receptacle of type IASPProtocol. Currently five implementations (ASPLocal, ASPFrag, ASPShmem, ASPMin, ASPGIOP).
4) Multipointer receptacle of type IBufPolicy. Currently two implementations (BufBuddy and BufMalloc).
5) Multipointer receptacle of type IBindTemplate. Currently one implementation (BindGOPI).