

Studies of Efficiency and Integrity in the OpenORB Reflective Middleware Architecture

Gordon S. Blair¹, Geoff Coulson², Michael Clarke² and Nikos Parlavantzas²

¹Department of Computer Science, University of Tromsø, N-9037 Tromsø, Norway (On leave from Lancaster University).

E-mail: gordon@cs.uit.no

²Distributed Multimedia Research Group, Department of Computing, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K.

E-mail: {geoff, mwc, parlavan}@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, DCOM and .NET are not flexible enough to meet the needs of emerging distributed applications, featuring for example access to multimedia services and also support for mobile users. In particular, they are not sufficiently configurable and they do not support reconfiguration or longer-term evolution of architectures. Recently, a number of reflective middleware platforms have emerged in an attempt to overcome such problems. Considerable progress has been made, particularly in terms of support for configuration. Major problems remain however, especially in terms of performance and integrity. This paper presents a study of OpenORB v2 with emphasis on these issues. In terms of performance, the design is based on a lightweight reflective component model, based on Microsoft's COM. It is shown that the resultant OpenORB implementation performs at least as well as commercial ORBs. In terms of integrity, the design also features the use of component frameworks offering a domain specific approach to reconfiguration management. Early experiences indicate that this is a highly promising approach for maintaining integrity of the underlying middleware platform. Ongoing research is investigating further extensions to the platform including support for group communications.

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 (e.g. object-oriented or component-based) to users, and to mask out problems of distribution. Examples of key middleware platforms include 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, mobile and ubiquitous systems, real-time systems, the computational GRID, etc. Unfortunately, as this diversification proceeds, it is becoming ever more apparent that the middleware technologies mentioned above are not able to support such a diversity of application domains.

The main reason for this is the *black-box* philosophy adopted by 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, i.e. they are *closed systems*. Inevitably, the platform architecture represents a compromise featuring, e.g. general-purpose protocols and associated management strategies. It is then not possible to *configure* platforms to meet the needs of more specific target domains. Similarly, it is not possible to *reconfigure* platforms at run-time as, for example, the underlying environmental conditions fluctuate. Equally, it is difficult to *evolve* such architectures in the longer-term to meet new application requirements.

Middleware designers are aware of this problem and have responded with a number of initiatives. Focusing on CORBA for example, the OMG have introduced a series of platform specifications including *Real-time CORBA* [OMG01] and *Minimal CORBA* [OMG01]. These are however specific solutions to specific domains and are not a general solution to this problem. In addition, the *Portable Interceptors* specification [OMG99] enables the customisation of CORBA platforms by allowing the interception of invocations or replies via pre- or post-processing. Portable interceptors also enable the interception of IOR (Interoperable Object Reference) creation. This is a useful but limited mechanism. For example, it is not possible to add interceptors at arbitrary points in the ORB implementation. Similarly, there is no native support for structured composition or for the dynamic installation of interceptors. It could be argued that the interceptor programmer can enhance the system with such functionality, but this would inevitably result in a proprietary solution. The other notable

customisation feature in CORBA is the use of *policy objects* [OMG01], which allow a level of control over particular internal services of the ORB. Such policies can be used for example to customise aspects such as the POA (Portable Object Adapter), asynchronous messaging, security and real-time functionality. However, once the policies are installed (e.g. at the time an object reference is created), the set of potential policies are fixed and cannot be changed, thus not fully supporting dynamic adaptation. Other middleware standards and platforms also offer similar degrees of flexibility, e.g. *custom marshalling* in COM [Microsoft00a], *interception* in COM+ [Microsoft00b] and *dynamic proxies* in Java [Sun00]. These, however, all suffer from similar problems to those described above. In general, such mechanisms can be viewed as *ad-hoc* and *incomplete* in terms of support for openness and adaptation. They are certainly insufficient to meet the considerable demands of next generation distributed applications.

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 (among 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 platform (e.g. protocols and management structures being deployed) and to make changes at run-time. This paper presents the design and implementation of *OpenORB*, a reflective middleware platform developed at Lancaster University. More specifically, the paper focuses on *OpenORB 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 [Blair98, Costa98, Costa00, Andersen00]).

The paper is structured as follows. Section 2 presents an analysis of the state of the art in reflective middleware, highlighting the problems relating to performance and integrity. Section 3 then presents the approach adopted in *OpenORB* with regard to these two issues. In particular, we describe a lightweight component model, *OpenCOM*, and also our use of *component frameworks* to manage integrity. Following this, section 4 discusses the implementation of *OpenORB* with emphasis on the role of such frameworks. Section 5 offers an initial performance evaluation of *OpenORB*, with some overall conclusions drawn in section 6.

2. Reflective Middleware: An Analysis of the State of the Art

2.1. Motivation

The main motivation for this research is to provide a *principled* (as opposed to *ad hoc*) means of achieving openness. For example, reflection can be used to inspect the internal behaviour of a platform (*introspection*). By exposing the underlying implementation, it becomes straightforward to insert additional behaviour to monitor the implementation, e.g. performance monitors, quality of service monitors, or accounting systems. Reflection can also be used to alter the internal behaviour of the underlying middleware (*adaptation*). Examples include replacing or changing the implementation of the underlying transport protocol to operate more optimally over a wireless link, introducing an additional level of distribution transparency in a running computation (e.g. migration transparency), or inserting a filter to reduce the bandwidth requirements of a media stream.

There are also strong arguments that middleware is the *most appropriate locus* to offer such capabilities. Offering such functionality in the underlying operating system is dangerous and can compromise the overall integrity of the system. Considerable advantages have been gained from supporting reflection in programming languages but, with this approach, the benefits are obviously restricted to that particular language. It is also clearly inappropriate to leave support for adaptation to the application, as application writers would not have available the necessary levels of openness to implement their desired policies. In reflective middleware, the role of the middleware platform is to offer a language and operating system independent framework for managing adaptation on behalf of the application, thus extending the benefits of portability and interoperability to adaptive applications and systems.

2.2. Reflection and Middleware

2.2.1. Styles of Reflection

Middleware platforms offer two (complementary) styles of reflection, i.e. structural and behavioural reflection:

- *Structural reflection* is concerned with the underlying structure of objects or components, e.g. in terms of interfaces supported (c.f. introspection features found in Java [Sun00]). More advanced features may also be offered such as the ability to adapt the structure of an object, e.g. to add new behaviour at run-time. Similarly, some systems provide *architectural reflection*, whereby the software architecture of the system, e.g. in terms of components and connectors, can be reified and altered [Blair01, Cazzola99]. This can be

applied to the very structure of the middleware platform itself, allowing the customisation of the architecture to current environmental conditions.

- *Behavioural reflection* is concerned with activity in the underlying system, e.g. in terms of the arrival and dispatching of invocations. Typical mechanisms provided include the use of interceptors that support the reification of the process of invocation and the subsequent insertion of pre- or post- actions (as mentioned above). Other systems provide similar capabilities through dynamic proxies [Sun00]. Finally, some research has been carried out on providing access to underlying resources and associated resource management, e.g. through the reification of a set of logical tasks and enabling the customisation of resource allocation and management policies [Duran-Limon00a, Duran-Limon00b].

In our own research, we have also combined both forms of reflection into an overall architecture [Blair98, Blair01] (c.f. the work on AL-1/D [Okamura92]).

2.2.2. Examples of Reflective Middleware Platforms

As mentioned above, a significant number of experimental platforms have now emerged including:

- *mChARM* from the University of Genova which focuses on the use of architectural reflection in terms of topological reflection, involving the manipulation of structure (in terms of components and connectors) and strategical reflection involving the manipulation of behaviour [Cazzola99];
- *DynamicTao* from the University of Illinois at Urbana-Champaign, an extension to TAO offering *configurators* that maintain dependencies between components and provide a set of hooks for the attachment or detachment of components dynamically [Kon00];
- *FlexiNet* from APM Ltd in Cambridge (now Citrix) which exploits reflection in Java to enable the programmer to tailor the underlying communications infrastructure by inserting/ removing protocol layers [Hayton98];
- Experiments at Trinity College Dublin on the use of the reflective language *Iguana* to develop more open and extensible middleware platforms, including implementations of minimum CORBA [Dowling00, Dowling01];
- *LegORB (now UIC)*, also from the University of Illinois at Urbana-Champaign, applying similar ideas as in *DynamicTao* (above) but for the customisation of platforms for mobile computing and, more generally, what they refer to as *active spaces* [Kon00];
- *OpenCorba*, developed by researchers at the Ecole des Mines de Nantes, which is an open, dynamically adaptable ORB that depends on a reflective language (NeoClasstalk), especially in terms of exploiting class-based reflection as offered by this language [Ledoux99];
- *OOPP* from the University of Tromsø, which is closely based on OpenORB but focusing on the area of quality of service management in reflective middleware platforms [Andersen00].

Other middleware platforms featuring aspects of reflection include Jonathan [Dumant98], QuO [Zinky01] and TAO [Schmidt99] (including recent work on extending the latter with an open implementation of the CORBA Component Model [Wang01]). Finally, a number of researchers have carried out more specific and indeed complementary experiments on the use of reflection in key areas such as transactions [Barga98] and fault-tolerance [Fabre98, Killijian00].

2.3. Overall Analysis

Overall, middleware plays an increasingly central role in the design of modern computer systems and will, we believe, continue to enjoy this prominence in the future. There is however a demonstrable need for more openness and flexibility in middleware. We argue strongly that reflective middleware is the right technology to meet these demands. Indeed, there is growing evidence that such platforms are more configurable, reconfigurable, and also offer support for software evolution generally [Blair01]. As an example, a minimal configuration of LegORB, targeted at palm devices, has been created with a footprint of 16kbytes [Roman01].

While great progress has been made in the reflective middleware community, many outstanding issues remain. Firstly, it is not clear if reflective middleware technologies can achieve acceptable *performance* when compared to more traditional platforms. Secondly, the issue of *integrity* is not yet resolved. This is crucial if such technologies are to be fully deployed. The main goal of OpenORB v2 is to address these limitations. More specifically, we have the following key objectives:

- In the worst case, performance should be on a par with that of conventional middleware platforms, and in the best case (e.g. in the case of cut-down configurations) it should be significantly *better*;
- While permitting maximal reconfigurability, it should be possible to control and constrain the scope of reconfigurations so that damaging changes are discouraged and/ or disallowed.

We report on the results of this work in the rest of this paper.

3. Overall Approach

3.1. Overview

The OpenORB architecture builds on two complementary technologies; namely *components* and *reflection*. More specifically, in OpenORB we provide a component model [Szyperski98] not just at the application level, but also for the construction of the middleware platform itself. Thus, an instance of OpenORB is a particular configuration of components, which can be selected at build-time and reconfigured at run-time (full details of the component model, including its intrinsic support for multimedia, can be found in the literature [Blair01]). Access to the underlying platform, and by implication the associated component structure, is provided through reflection. In particular, every application-level component offers a meta-interface providing access to an underlying meta-space that is in effect the support environment for this component (c.f. 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 render this implementable, meta-components are instantiated on demand; unless accessed, they exist in theory but not in practice. In OpenORB, meta-space is partitioned into distinct meta-models offering both structural and behavioural reflection (again, further details can be found in the literature [Blair01]).

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 its intrinsic support for rapid prototyping and also the underlying reflective capabilities of the language. Nevertheless, because of 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 OpenORB using C++. In more detail, the approach adopted is to define a base reflective component model, *OpenCOM*, as an extension to Microsoft's COM architecture, and then to use this to implement a component-based middleware platform. In addition, to address the issue of integrity, we rely heavily on the concept of *component frameworks*. We look at these two key underlying technologies below.

3.2. OpenCOM

As mentioned above, OpenCOM is closely based on Microsoft's COM but enhanced with richer reflective facilities. OpenCOM relies only on the core of COM, i.e. i) the basic binary-level interoperability standard (the *vtable* data structure), ii) Microsoft's Interface Definition Language (IDL), iii) COM's globally unique identifiers (GUIDs), and iv) the *IUnknown* interface (for interface discovery and reference counting); it avoids dependencies on other features of COM such as distribution (via DCOM), persistence, security and transactions. Crucially, we 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, the type and location of which is 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 can be 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. OpenCOM also deploys a standard *run-time* that is available in every OpenCOM address space. This run-time manages a repository of available component types for lifecycle management, and also maintains an overall system graph in support of the *IMetaArchitecture* interface (see below).

Crucially, OpenCOM also provides a number of meta-interfaces providing low-level support for introspection and adaptation:

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.

The overall architecture of OpenCOM is summarised in figure 1 below.

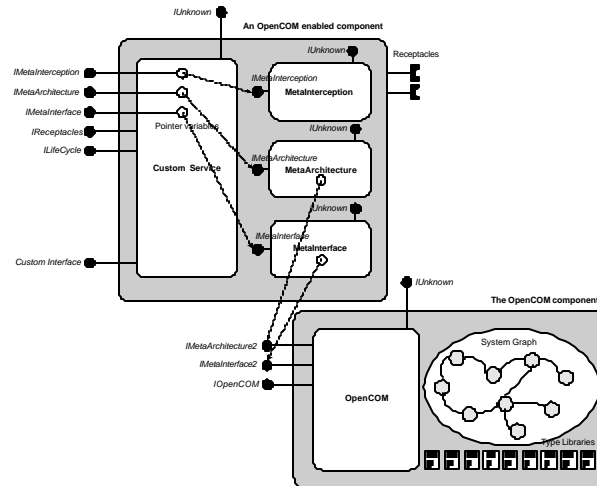


Fig. 1. The Architecture of OpenCOM

Further details on all aspects of OpenCOM can be found in [Coulson01c].

3.3. Component Frameworks

The second key technology underpinning OpenORB is an instantiation of the concept of *component frameworks* (CFs). This term was originally defined by Szyperski [Szyperski98] to refer to “collections of rules and interfaces (contracts) that govern the interaction of a set of components plugged into them”. They are targeted at a specific domain and embody rules and interfaces that make sense in that domain.

In OpenORB, we specialise the generic CF notion in a number of key ways. Most importantly, CFs in OpenORB are not just a design concept. Rather, they are reified as run-time software entities (packages of components) that support and police components plugged into the CF to ensure that they conform to its rules and contracts. CFs therefore are given the important role of maintaining the overall *integrity* of that part of the component architecture through *reconfiguration management*. This has the advantage that CFs can exploit domain-specific knowledge and built-in constraints to enforce the desired level of integrity across reconfiguration operations (in terms of both functional and non-functional concerns), and also perform domain specific trade-offs between flexibility and consistency. In more general terms, CFs aim to establish architectural properties and invariants by constraining the design space of inserted (i.e. plug-in) components.

The fundamental issues in reconfiguration management are: i) to control and constrain the scope of reconfiguration operations, ii) to separate concerns between reconfiguration operations and core middleware functionality, and iii) to maintain integrity in the face of dynamic change. To address the first of these issues, we employ *nesting* as a means of providing hierarchical scoping and structure. For example, the top-level structure of OpenORB is itself a CF (see below). We then address the second and third issues by applying a *manager/managed* pattern within the resultant hierarchical scopes. In the *manager/managed* pattern, *managers* collect events and issue management operations (i.e. implement management *policies*). Conversely, *managed entities* accept management operations and issue event notifications (i.e. implement management *mechanisms*).

More specifically, CFs take the manager role: they monitor events emitted by their plug-ins, maintain meta information representing the current configuration, and effect changes on this configuration. These changes may involve the invocation of management operations on plug-ins, the setting of attributes, or modification of the plug-in configuration (i.e., adding/ deleting/ connecting/ disconnecting plug-ins using OpenCOM primitives). At the same time, CFs are responsible for exposing themselves as *managed* entities with respect to higher-level CFs or other manager components (e.g., application components). Thus, each CF provides a meta-interface offering adaptation operations and also possibly generates events.

4. The Design and Implementation of OpenORB v2

4.1. Overview

The implementation of OpenORB v2 is structured as a top-level CF that is then composed of three layers of further CFs. The top level CF enforces the three-layer structure by ensuring that each component/CF only has access to interfaces offered by components/CFs in the same or lower layers. Furthermore, it imposes policies concerning dynamic changes in layer composition. The second level CFs address more focused sub-domains of middleware functionality (e.g., binding establishment and thread management) and enforce appropriate sub-domain specific policies.

This hierarchical structure opens up two distinct dimensions of flexibility. Firstly, the top level CF can be configured by selecting the set of CFs that will initially populate the layers (together with their inter-connections and their associated policies). This configuration defines the *middleware architecture* as published to developers who want to use, configure or extend the platform. Different architectures can be defined for different platforms or application domains. Secondly, a particular instance of the middleware architecture is dynamically configurable in terms of introducing new CFs (as long as they conform to the policies of the top-level CF) and by customizing or extending the existing second-level CFs (both statically and dynamically).

The current OpenORB architecture consists of 6 CFs and is seen in figure 2. However, we should stress again that this is only one possible configuration, and that many other architectures can equally well be created.

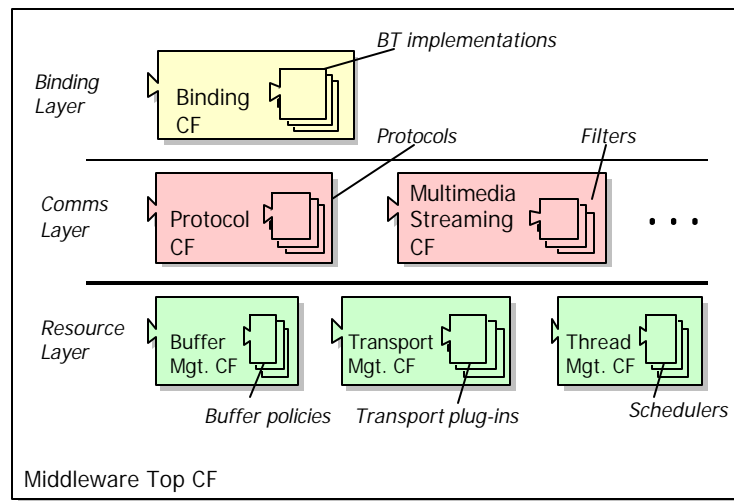


Fig. 2. Top level architecture of OpenORB

The *resources layer* currently contains *buffer*, *transport*, and *thread management* CFs that respectively manage buffer allocation policies, transport protocols and thread schedulers. Next, the *communication layer* contains *protocol* and *multimedia streaming* CFs. The former accepts plug-in *protocol* components and the latter accepts *filter* components. Finally, the *binding layer* contains the *binding CF* that accepts *binding type implementations* (e.g., remote object invocation, streaming connections or publish/subscribe etc.). This is a crucial part of the architecture because it determines the programming model offered to middleware users.

The following sections present the architecture in more detail. The resources and communications layers are covered in section 4.2, and the binding layer is discussed in detail in section 4.3.

4.2. The Resources and Communications Layers

Our current implementation of these two layers consists of 50,000 lines of C++ (including the OpenCOM runtime) divided into about thirty components and five CFs. The bulk of the code is derived from GOPI, a CORBA compliant, multimedia capable, middleware platform that we have developed previously [Coulson99].

Each CF in the resource and communications layers follows a similar pattern: it defines an abstract interface and manages different implementations of this interface, which are plugged in as separate components and are selectable at run-time. In addition, the CFs all offer meta-interfaces for domain specific dynamic reconfiguration. For example, there are operations to ensure that managed components can be dynamically loaded and unloaded without disruption to currently executing applications.

The *thread management* CF multiplexes user-level threads over kernel threads (referred to as *virtual processors*), and supports the dynamic selection of *scheduler* components, each of which manages its own threads and dedicated virtual processors [Coulson01a]. The thread CF's dynamic reconfiguration interface enables the dynamic loading/ unloading of these schedulers. Currently there are three scheduler implementations: a simple priority based policy, an earliest deadline first policy, and a 'native' policy (which maps every user level thread to a single kernel thread).

Also in the resources layer, the *buffer management* CF defines an abstract interface that enables developers to write their own tailored buffer allocation policies using a common buffer abstraction. Currently there are two buffer implementations, a *malloc()* based implementation that maps directly down to OS level memory management routines, and a more efficient buddy scheme [Knuth73]. Similarly, the *transport* CF defines an abstract interface that enables developers to add transport protocols. Currently, we support TCP, UDP, multicast IP, and IPC (i.e. pipes on UNIX and memory mapped files on Windows). In addition, the transport CF supports another type of plug-in that supports the definition of alternative *message detection strategies* for incoming messages. The default strategy only looks for new messages when all threads in the address space are currently blocked. Other strategies rely on various combinations of server threads, thread pools and signal driven I/O notification. Further details are available in [Coulson99] and [Coulson01b].

Finally, in the communications layer, the *protocol* and *multimedia streaming* CFs define a plug-in environment for stacked communications protocols and composable media processing filters. Currently, four protocol implementations exist: a CORBA GIOP v1.2 object request protocol implementation, a protocol that implements simple fragmentation services over any underlying transport, a protocol that efficiently passes data between end-points in the same address space, and a protocol that employs shared memory for communication and the IPC transport for synchronisation. These CFs also maintain information about the current protocol/ filter configurations (organized as a graph of instances) and offer meta-interfaces with specialised operations to reconfigure the graph in such a way that rules constraining permissible configurations are obeyed.

4.3 The Binding Framework

The binding layer is arguably the most interesting feature of this architecture. In contrast to most existing middleware platforms, OpenORB supports an extensible set of binding types including remote method invocation, publish/ subscribe, message queuing and media streaming. By capturing diverse forms of interaction as middle are-provided binding types (*BTs*), the binding CF significantly simplifies application development and promotes the reuse of interaction mechanisms over multiple applications. *BTs* effectively realise software architecture *connectors*, thus bridging the gap to software architecture research.

The binding CF specifies two contracts (as shown in figure 3): i) the *binding API*, which defines the view of *BTs* seen by binding users, and ii) the *BT contract*, which governs the collaboration between *BT* implementations and the binding CF itself.

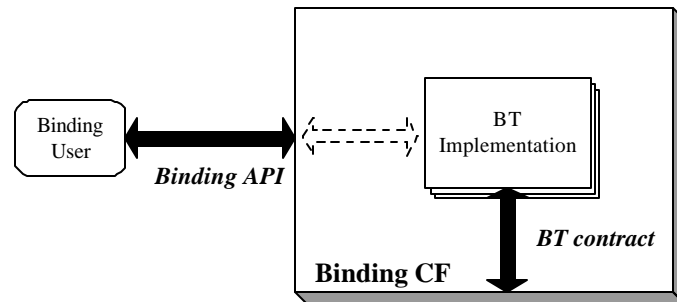


Fig. 3. The Binding CF Contracts

The binding API is based on a small number of generic entities and interactions designed to capture commonalities across various *BTs*. It does not attempt to specify a uniform interface for all *BTs* (this is clearly unfeasible) but rather it defines generic COM interfaces and rules and guidelines that provide consistency for binding users and guidance for *BT* implementers.

Briefly, the API model assumes that bindings are established between *participants* and the responsibility for binding establishment is assigned to *binders*. Binders take as input a number of objects representing participants together with related attributes, verify that the supplied participant objects conform to roles specified in the associated *BT*, establish the binding and return (if nothing goes wrong) a *binding control* object. Participants that are remote with respect to a binder's location are represented by *rep* objects. The process of creating a *rep* falls into two stages as follows. First, a *generator* is used at the participant's (remote) site to generate both an

iref and an associated communications infrastructure. Second, the *iref* is transferred to the binder's site (by some means or other) where it is passed to a *resolver* that is responsible for creating a corresponding *rep*.

The BT contract defines interactions that enable the binding CF to expose BT implementations to binding users, manage their lifecycle and provide them with access to both other BTs and services in the lower platform layers (communications and resources). The set of available BTs can be configured statically and also changed at run-time by dynamic loading when an *iref* of a specific type arrives. The reconfiguration of established bindings is achieved by building on both binding control objects provided by other BTs and meta-interfaces offered by the lower layers and the component model. Further details of the binding CF can be found in a forthcoming paper [Coulson01c].

5. Performance Evaluation

Our expectation is that OpenORB should perform as well as existing ORBs while simultaneously providing dynamic reconfigurability through componentisation, reflection and CFs. To evaluate this expectation we compared the performance of OpenORB with two other ORBs: GOPI v1.2 and Orbacus 3.3.4. As stated, GOPI provides much of the source code for OpenORB v2 but is written in C and implemented in a single library. A direct performance comparison should therefore yield insight into the overhead of our component model. Orbacus is well known as one of the fastest and most mature CORBA-compliant commercial ORBs available.

The OpenORB configuration used was that shown in figure 2 above with a binding type that implements remote method invocation on top of the CORBA GIOP protocol (and without locks on receptacles). Our tests compared raw method invocations per second over loopback on a Dell Precision 410 workstation equipped with 256Mb RAM and an Intel Pentium III processor rated at 550Mhz. The operating system was Windows 2000 and the compiler was Microsoft's cl.exe version 12.00.8804 with flags /MD /W3 /GX /FD /O2. An IDL interface was employed that supported a single operation that took as its argument an array of octets of varying size. The implementation of this method at the server side was null.

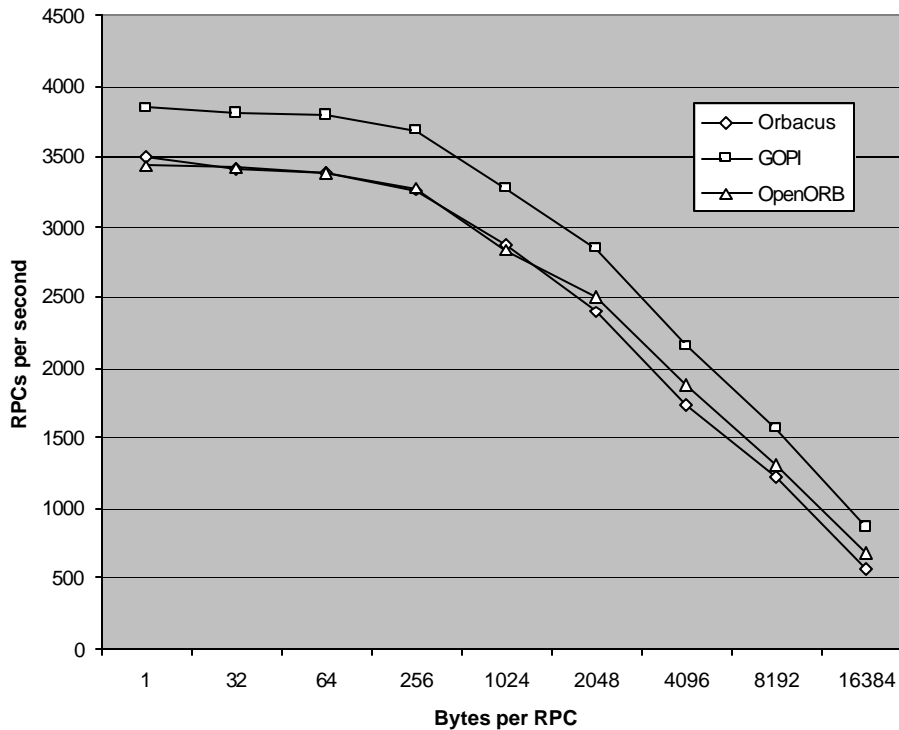


Fig. 4. Performance of OpenORB versus GOPI and Orbacus

The results of timing a large number of round-trip invocations using this setup are shown in figure 3. It can be seen that OpenORB performs about the same as Orbacus, with GOPI running around 10% faster. As might be expected, there is a diminishing difference between all three systems as packet size increases; this is presumably

because the overhead of data copying begins to outweigh the cost of call processing. The relative overhead of OpenORB compared to GOPI can be attributed to COM's use of indirection (through the *vtable*) and also to the former's use of receptacles. The OpenORB configuration involved 67 receptacle-interface connections on the data path per RPC (32 on the client-side and 35 on the server side). In the GOPI case, a method call is implemented as an immediate register load followed by a call through that register. In the OpenORB case, on the other hand, an interface pointer must first be extracted from the target receptacle (involving a memory access to locate the receptacle in the current component and a call to the overloaded dereference operator followed by a memory access within the receptacle), and then a C++ virtual method call on the interface pointer (involving a memory access to get the current component's appropriate interface *vtable* pointer, a register indirect operation to acquire a pointer to that interface's table of function pointers, an addition to work out the method's offset within the interface, a push of "this" and finally, the actual method call). Despite this additional work, it can be seen that the performance of OpenORB is entirely comparable to the non-componentised ORBs.

6. Conclusions

This paper has discussed the design and implementation of OpenORB, an experimental reflective middleware architecture intended to provide a high degree of 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 structure to be accessed and indeed altered dynamically.

More specifically, the paper has described an implementation of the OpenORB architecture using C++ together with a lightweight and reflective component model based on COM (OpenCOM). In addition, the implementation features the use of domain specific component frameworks supporting reconfiguration management and integrity. Our experiences from this work are summarized below:

1. We believe that the combination of a reflective component model and the CF-based structuring principle represents a highly promising basis for the construction of configurable and reconfigurable ORBs. While the reflective component model provides a powerful basis for maximal flexibility and reconfigurability, on its own it is too expressive, and its unconstrained use can easily lead to chaos. The presence of CF-based structuring tempers this expressiveness by imposing domain specific constraints on the reconfiguration process.
2. We also believe that the three-layer structure of our top-level CF represents a generic and future-proof basis for the expansion, extension and evolution of OpenORB v2. We are encouraged in this belief by the fact that we have been able to easily accommodate several components and CFs that were not initially envisaged. For example, we have recently accommodated thread pool and operating system abstraction components in the resources layer and plan the introduction of a group communications CF in the communications layer. As for the binding CF, we have not yet encountered a binding style that it could not accommodate, although we have less implementation experience in this area than in the other layers.
3. Our implementation efforts have also validated other aspects of our design. For example, we have discovered that the component model supports the construction of ORB functionality that is at least as efficient as conventional object-based ORBs (see section 5). Furthermore, we have confirmed that the component model scales well in terms of its explicit enumeration of per-component dependencies. This is primarily due to the use of CFs which reduce dependencies by forbidding connections between plug-in components and components outside the CF. In our current implementation, the maximum number of dependencies in any single component is just seven and the average figure is just four. This leaves considerable scope for further reducing the granularity of componentisation that, if carried out with care, should correspondingly increase the ORB's potential for reconfigurability.

Currently, our efforts are focusing on the further development of the OpenORB environment. For example, we are adding a plug-in component type to the protocol CF that will enable the configuration of a range of demultiplexing strategies (that have previously been implemented in the GOPI platform [Coulson01a] although with limited scope for run-time reconfiguration). We are also adding support for group communications via an additional communications layer CF as mentioned above [Saikoski00].

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.

Thanks also to the other members of the reflection team at Lancaster who have contributed greatly to the ideas presented in this paper, in particular Lynne Blair, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira and Katia Saikoski.

Finally, we would 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 Anders Andersen, Frank Eliassen, Vera Goebel, Oyvind Hassen and Thomas Plagemann.

References

- [Andersen00] Andersen, A., Eliassen, F., Blair, G.S., "A Reflective Component-Based Middleware with Quality of Service Management", Proceedings of PROMS'2000 (Protocols for Multimedia Systems), Cracow, Poland, 2000.
- [Barga98] Barga, R., "A Reflective Framework for Implementing Extended Transactions", PhD Dissertation, Oregon Graduate Institute, Portland, Oregon, 1998.
- [Blair98] Blair, G.S., Coulson, G., Robin, P., Papathomas, M., "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, 1998.
- [Blair01] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzias, N., Saikoski, K., "The Design and Implementation of OpenORB v2", To appear in in IEEE DS Online, Special Issue on Reflective Middleware, 2001.
- [Cazzola99] Cazzola, W., Savigni, W., Sosio, A., Tisato, F., "Rule-based Strategic Reflection: Observing and Modifying Behaviour at the Architectural Level, Proc. 14th IEEE International Conference on Automated Software Engineering (ASE'99) Cocoa Beach, Florida, USA, October 1999.
- [Costa98] Costa, F., Blair, G.S., Coulson, G., "Experiments with Reflective Middleware", Proceedings of the ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems, ECOOP'98 Workshop Reader, Springer-Verlag, 1998.
- [Costa00] Costa, F., Duran-Limon, H., Parlavantzias, N., Saikoski, K., Blair, G.S., Coulson, G., "The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications", In Reflection and Software Engineering, Cazzola, W., Stroud, R. and Tisato, F. (Eds), Springer-Verlag, LNCS Vol. 1826, pp 79-98, 2000.
- [Coulson99] Coulson, G., "A Configurable Multimedia Middleware Platform", IEEE Multimedia, Vol 6, pp 62-76, No 1, January - March 1999.
- [Coulson01a] Coulson, G., and Mounian, O., "A Quality of Service Configurable Concurrency Framework for Object Based Middleware, Lancaster University Internal Report, submitted for publication, 2001.
(Available from http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/papers/ASC_paper_submitted.pdf)
- [Coulson01b] Coulson, G., and Baichoo, S., "Implementing the CORBA GIOP in a High-Performance Object Request Broker Environment", to appear in Distributed Computing Journal, 2001.
- [Coulson01c] Coulson, G., Blair, G.S., Clarke, M., Parlavantzias, N., "The Design of a Reconfigurable and Efficient Middleware Platform", In Preparation, 2001.
- [Dowling00] Dowling, J., Schäfer, T., Cahill, V., Haraszti, P., Redmond, P., "Using Reflection to Support Dynamic Adaptation of System Software: A Case Study Driven Evaluation", In Reflection and Software Engineering, Cazzola, W., Stroud, R. and Tisato, F. (Eds), Springer-Verlag, LNCS Vol. 1826, 2000.
- [Dowling01] Dowling, J., Cahill, V., "Building a Dynamically Reconfigurable minimumCORBA Platform with Components, Connectors and Language-Level Support", Proceedings of the Workshop on Reflective Middleware (RM'2000), IBM Palisades, New York, USA, April 2000.
- [Dumant98] Dumant, B., Horn, F., Dang-Tran, F. and Stefani, J.-B., "Jonathan: an Open Distributed Processing Environment in Java", Proc. Middleware '98, The Lake District, England, November 1998.
- [Duran-Limon00a] Duran-Limon, H., Blair, G.S., "The Importance of Resource Management in Engineering Distributed Objects", Proc. 2nd International Workshop on Engineering Distributed Objects (EDO'2000), California, USA, November 2000.
- [Duran-Limon00b] Duran-Limon, H., Blair, G.S., "Specifying Real-time Behaviour in Distributed Software Architectures", Proc. 3rd Australasian Workshop on Software and System Architectures, Sydney, Australia, November 2000.
- [Fabre98] Fabre, J.-C., Pérennou, T., "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach", IEEE Trans. on Computers, Special Issue on Dependability of Computing Systems, vol.47, no.1, pp. 78-95, 1998.
- [Hayton98] Hayton, R., Herbert, A., Donaldson, D., "FlexiNet: A Flexible Component-oriented Middleware System", Proc. 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications, Sintra, Portugal, September 1998.
- [Killijian00] Killijian, M.-O., Fabre, J.-C., "Implementing a Reflective Fault-Tolerant CORBA System", Proc. of the 19th Symposium on Reliable Distributed Systems (SRDS2000), Nurnberg, Germany, pp. 154-163, Oct. 2000.
- [Knuth73] Knuth, D.E., "The Art of Computer Programming, Volume 1: Fundamental Algorithms", Second Edition, Reading, Massachusetts, USA, Addison Wesley, 1973.

- [Kon00] Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L.C., Campbell, R.H., "Monitoring, Security and Dynamic Configuration with the dynamicTAO Reflective ORB", Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), IBM Palisades, New York, April 2000.
- [Ledoux99] ledoux, T., "OpenCorba: A Reflective Open Broker", Proc. Reflection'99, Saint-Malo, France, Springer-Verlag, LNCS, Vol. 1616, 1999.
- [Microsoft00a] Microsoft, "COM: Delivering on the Promises of Component Technology", URL: <http://www.microsoft.com/com/default.asp>, 2000.
- [Microsoft00b] Microsoft, "COM Technologies - COM+", URL: <http://www.microsoft.com/com/tech/complus.asp>, 2000.
- [Okamura92] Okamura, H., Ishikawa, Y., Tokoro, M., "AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework", Proceedings of the Workshop on New Models for Software Architecture, November 1992.
- [OMG99] OMG, "Portable Interceptors - Joint Revised Submission", Object Management Group, TC Document orbos/99-12-02, 1999.
- [OMG01] OMG, "Common Object Request Broker Architecture and Specification - Revision 2.4.2, Object Management Group, 2001.
- [Roman01] Roman, M., Kon, F., Campbell, R.H., "Reflective Middleware: From the Desk to your Hand", To appear in IEEE DS Online, Special Issue on Reflective Middleware, 2001.
- [Saikoski,00] Saikoski, K. B. and Coulson G., "Configurable and Reconfigurable Group Services in a Component Based Middleware Environment", Proc. International SRDS (Symposium on Reliable Distributed Systems) Workshop on Dependable and Group Communication (DSMGC 2000), October 2000.
- [Schmidt99] Schmidt, D.C., and Cleeland, C., "Applying Patterns to Develop Extensible ORB Middleware", IEEE Communications Magazine Special Issue on Design Patterns, April, 1999.
- [Sun00] Sun Microsystems, "Java Reflection", URL: <http://java.sun.com/j2se/1.3/docs/guide/reflection/index.html>, 2000.
- [Szyperski98] Szyperski, C., "Component Software: Beyond Object-Oriented Programming", Addison-Wesley, 1998.
- [Wang01] Wang, N., Parameswaran, K., Schmidt, D.C., Kircher, M., "Towards a Reflective Middleware Framework for QoS Enabled CORBA Component Model Applications", DS Online, This Special Issue, 2001.
- [Zinky01] Zinky, J., Shapiro, R., Loyall, J., Anderson, K., Schantz, R., Pal, P., "The Use of Reflectivity in the QuO 3.0 Framework", To appear in IEEE DS Online, Special Issue on Reflective Middleware, 2001.