

# Towards a Reflective Component-based Middleware Architecture

Nikos Parlavantzas, Geoff Coulson, Mike Clarke and Gordon Blair

Distributed Multimedia Research Group,  
Computing Department, Lancaster University,  
Bailrigg, Lancaster LA1 4YR, UK.  
{parlavan, geoff, mwc, gordon}@comp.lancs.ac.uk

## **Abstract**

*Current middleware is inflexible and monolithic and fails to address emerging needs for adaptation to changing requirements. As a solution, we propose that middleware be built as a reflective and component-based system. Our architecture is based on component frameworks and component framework-specific meta-interfaces and leads to extensible, composable, and dynamically flexible middleware systems. Moreover, it addresses the critical problem of ensuring integrity in the face of dynamic adaptation. A pilot implementation, based on an efficient, lightweight component model, is currently underway.*

## **1. Introduction**

It is now well established that middleware platforms must accommodate a wide variety of requirements imposed by both applications and underlying environments. Moreover, they must be able to absorb both design-time and run-time changes in these requirements. Unfortunately, the current generation of middleware is, to a large extent, monolithic and inflexible and, thus, fails to address these needs. There have been some efforts to introduce reconfigurability, but these are typically piecemeal, ad-hoc, and usually involve selection between a fixed number of options. In our opinion, a more systematic and principled solution is needed.

To this end, we have previously carried out research on a *reflective* middleware architecture [2,3,4]. The main principles of this architecture are the association of a meta-space per object (as opposed to, for example, per-class meta-interfaces) and the decomposition of the meta-space in multiple, orthogonal meta-models. The research has resulted in a pilot implementation of the architecture in Python which has focused primarily on achieving as great a degree of flexibility as possible. The work described in this paper represents an evolution of our previous work, based on *component technology*, which is additionally focused on efficient implementation and on ensuring integrity in the face of dynamic adaptation.

Component technology has recently emerged as a promising approach for building highly adaptable software systems. This adaptability is a result of the capability to change configurations by adding, removing and replacing their constituent components (importantly, components come in a binary form and thus can be dynamically deployed within an address space). Additional benefits of component technology include reusability, dynamic extensibility, understandability, and reduced development costs [19].

Currently, component technology is applied only at the application level on top of middleware infrastructures, which hide distribution and other non-functional concerns from component developers. However, in order to address the need for adaptability, we believe that middleware *itself* should be built according to a component-based architecture. Furthermore, we believe that the resulting component-based middleware should be reflective to help facilitate and manage run-time changes in component configurations. In other words, it should incorporate structures representing aspects of itself and offer meta-interfaces for inspecting and adapting these reified aspects. Unfortunately, designing these meta-interfaces is not easy. We have found that the ‘obvious’ solution of exposing the whole middleware implementation as a graph of component instances and allowing arbitrary manipulations is not acceptable for reasons of robustness. Maintaining integrity is a critical issue arising in the construction of all reflective systems. Our proposed approach to addressing this issue is through the notion of *component frameworks*. This paper discusses our proposed architecture for middleware which employs both reflection and component frameworks.

This paper is structured as follows. Section 2 outlines the basic principles underlying our architecture. Section 3 then describes our component model in some detail. Following this, Section 4 outlines the component-based middleware architecture itself, and Section 5 discusses our implementation work to date. Finally, Section 6 discusses some related work, and Section 7 presents concluding remarks.

## 2. Principles

Our proposed reflective component-based architecture for middleware is based on the following principles:

**Use of Component Frameworks** A component framework (CF) is a collection of rules and contracts that govern the interaction of a set of components [19] (we say that these components extend the CF). CFs typically address a specific and focused problem domain (e.g., implementing communications protocols as components), and thus many CFs may need to be integrated in a component system. The primary motivation for CFs is to provide built-in architectural properties and invariants by constraining the design space of extending components. Moreover, CFs simplify component development and assembly, enable lightweight components and increase the understandability and maintainability of systems.

While CFs are, by standard definition, design-level entities, in our architecture we *explicitly represent* CFs as components that enforce some of these rules by implementing common mechanisms and regulating interactions. We refer to these as *component framework representatives* (CFRs). Our approach is then to decompose the middleware architecture into an extensible set of specialized and focused domains of concerns, such as buffer management and binding establishment, each based on a CF/ CFR.

**Use of CF-specific meta-interfaces** In our architecture, CFs are associated with meta-interfaces through which the implementation of CF-based subsystems (i.e., configurations of component instances obeying the CF rules) can be exposed in a controlled and principled way. The meta-interfaces are typically implemented by CFRs, which maintain information about the current configuration and apply it to perform inspection and adaptation.

Because meta-interfaces are designed as an integral part of the CF, they can easily embody domain-specific knowledge, and can enforce a desired level of (domain-specific) consistency and integrity. The degree of flexibility afforded by the meta-interface has to be balanced against concerns for efficiency, assured consistency/ integrity, and understandability. By using CF-specific meta-interfaces, we can make a different trade-off for each different domain.

Additionally, the designer can choose the particular architectural style and meta-interface style that is appropriate for each domain of concern. For instance, a CF for building communication protocols may benefit from an event-based architectural style with data sharing (like Coyote [1]), whereas a CF for multimedia streaming may employ a pipes-and-filters style (cf. Microsoft DirectShow, Open bindings [7]). In the first case, a possible meta-interface may enable the rebinding of

events to different event handlers, while in the second case it may enable the reconfiguration of the filter graph.

**Use of uniform component model** In our architecture, application *and* middleware components/ CFs all use the same component model. This is beneficial in terms of interoperability (e.g., a protocol component could make use of an application level spreadsheet component!), portability, and development costs, since programmers only have to familiarize themselves with a single programming model.

In addition, this principle removes the hard and fast distinction between applications and middleware. This, in turn, dissolves the traditional barrier between application programmers and system programmers, enabling the former to adopt the role of the latter when the need arises.

## 3. The OpenCOM Component Model

Our *OpenCOM* component model is closely based on Microsoft's COM [13] but enhances the latter with richer reflective facilities. OpenCOM relies only on the core of COM (i.e., the basic language-independent binary-level standard which enables components to be dynamically combined within a single address space); it avoids dependency on features of COM such as distribution (DCOM), persistence, security, and transactions. Note that our approach enjoys the benefit of interoperability with the COM world of components. Moreover, the binary-level nature of interconnections promises considerably less performance overhead than alternative component models such as JavaBeans.

A limitation of COM from our perspective is that COM components are often implicitly coupled to their environment. This makes it difficult to track dependencies and thus adapt component configurations with safety and integrity. In our model, therefore, components *explicitly* specify not only their *provided* interfaces but also their *required* interfaces. This is achieved by declaring an *IReceptacles* interface through which required interface references can be passed, so that connections can be established. OpenCOM thus promotes a connection-oriented programming model in which components are instantiated and bound by a *third party*.

OpenCOM improves on the basic reflective facilities of COM by supporting *introspection*, whereby each component has an *IMetaEncapsulation* interface providing meta-information about the interface types. This meta-information is used to support *dynamic invocation* of arbitrary interfaces (like Java's core reflection). OpenCOM also supports *interception* at specified interfaces. In particular, components implementing *IMetaEnvironment* enable dynamic attachment/detachment of interceptors that insert wrapping behaviour around method invocations.

## 4. A Component-Based Architecture for Middleware

Our proposed middleware architecture is structured as an extensible set of CFs based on OpenCOM. If we consider a basic communications middleware platform (e.g., without transactions support), the set of CFs described below are minimally required (our requirements are influenced by the GOPI platform discussed below). These CFs are organized in three layers, wherein components are only aware of interfaces/CFs defined at layers below themselves. Note that, courtesy of the underlying OpenCOM infrastructure, components are loaded only when they are actually needed and unload themselves when they are not used (exploiting COM's reference counting mechanism).

### 4.1 Binding Layer

This layer contains a **binding CF**, which defines a set of interfaces and associated rules/ semantics that govern the collaboration between to-be-bound components, *binder* components, and the CFR itself. Binder components, which extend the binding CF, are used to establish remote access paths (bindings) between to-be-bound components; their specific responsibilities include marshalling/unmarshalling interface references to/from appropriate binary representations and producing the required infrastructure (such as proxies, stubs and protocols). Third party components are given control over the process through CFR-specific meta-interfaces.

The binding CF can be extended with various binder components (dynamically downloadable) providing different binding types (e.g., request-reply invocations, event emission, stream bindings, group bindings). At run-time, binders create a component instance that locally represents each binding and offers a standard, minimal interface for adaptation. Additional interfaces can be provided by specific binders.

### 4.2 Communication Layer

This layer contains components/CFs that are used by binders to establish required communication paths. Minimally, it contains the **protocol CF**, which defines an architecture for dynamically composing and reconfiguring protocol stacks using lightweight protocols. This is extended with *protocol* components that are organized in a graph and offer various value-added services on top of OS-provided transports. The protocol CFR provides to the upper layer a generic base interface for communications services. It also maintains information about current protocol configurations and offers a meta-interface for performing dynamic adaptations. Note that the CF embodies rules that constrain permissible configurations. In particular, it

performs sanity checks on the topology of protocol configurations based on CF-defined attributes attached to protocols. It also provides specialised operations to add/ remove/ replace protocols in a local stack in such a way that connection state is preserved across the change.

The communication layer may contain additional CFs depending on the needs of the binding layer. For example, we provide a binder that establishes and manages multimedia streams and requires a **multimedia streaming CF** in the communication layer. This CF is extended with *filter* components (such as compressors, buffers, and renderers) and defines a series of protocols for the exchange of media samples, control and QoS information, error detection, etc. (cf. Microsoft DirectShow). The multimedia streaming CFR has a meta-interface that permits modifications of the filter graph (constrained by rules from the above mentioned protocols) and controls the state of processing.

### 4.3 Resource Layer

This layer contains a collection of components/CFs that provide a uniform API for using and controlling low-level resources. Resource control is essential for supporting applications and bindings with QoS requirements [3]. Components from the higher layer CFs (e.g., filters or protocols) are explicitly associated with the resource layer CFRs that serve their resource needs. This achieves a separation of the resource management aspect and facilitates component reuse.

The resource layer minimally contains the **buffer management CF** and the **transport management CF**. The associated CFRs provide base interfaces to buffer allocation and OS-level transport services respectively. The buffer management CF is extended with *buffer policy* components. The transport management CF is extended with *transport plug-in* components that encapsulate the network dependent aspects of transport protocols (like OCI for Orbacus [15]). It is associated with a meta-interface for adding and selecting between transport plug-ins at run-time. We also provide a **thread management CF** that handles user-level threads. It is extended with application-specific *scheduler* components each of which is associated with a particular scheduling policy (such as priority-based, or rate-monotonic). The thread management CFR meta-interface enables dynamic installation of scheduler components and migration of existing threads between schedulers.

## 5. Implementation

Our pilot implementation of the above architecture, referred to as *OpenORB*, is based on an existing middleware platform called GOPI [5]. GOPI provides a rich set of middleware services, including stream interfaces, third-party binding and QoS support, but is

implemented as a traditional procedural system. Specifically, GOPI is structured as a set of layered modules, comprising: **iref** (supporting communication endpoints called *irefs*, and a binding protocol/ QoS negotiation framework), **asp** (a framework that is extended with user level ‘application-specific protocols’ or *asps*), **tp** (a framework extended with transport protocols), **buf** (providing buffer management), and **threads** (a user-level real-time thread package extended with ‘application-scheduler contexts’, or *ascs*, which are essentially pluggable schedulers).

As well as reimplementing and repackaging GOPI’s *asp*, *tp*, *buf* and *threads* modules as OpenCOM CFRs, the OpenORB implementation adds numerous additional degrees of flexibility. For example, whereas GOPI offers essentially one binder/ binding protocol which supports a fixed set of binding types (signal, request/reply and stream), OpenORB offers multiple binder components as part of the binding CF. Furthermore, whereas the GOPI *asp* module provides only a simple static scheme of protocol composition, the OpenORB protocol CF is being extended to support dynamic composition and adaptation. A number of coarse-grained protocol components have already been developed for this CF including GIOP, components for shared memory communication, and networked (RTP-based) audio/ video protocols. We intend to investigate the provision of finer grained protocols in the future.

## 6. Related Work

FlexiNet [9], FlexiBind [8] and Jonathan [6] are Java ORBs that, like OpenORB, employ an extensible binding framework. TAO [16] and Quarterware [17] are other flexible ORBs. However, all these platforms are built in terms of OO frameworks rather than CFs. Our component-based approach has important advantages over OO frameworks; in particular, CFs are not bound to a specific programming language, and there is no implementation inheritance between components and the framework. As a result, components and CFs can be distributed in binary form, be independently developed, be combined at run-time, and evolve independently from each other.

OpenCorba is an open, dynamically adaptable ORB but depends on a reflective language (NeoClasstalk) [12]. COMERA [20] enables customisation of the remoting infrastructure of COM (i.e., DCOM), but the components are coarse-grained; we apply more aggressive componentisation guided by a set of CFs. DynamicTAO and LegORB are reflective ORBs that rely on a set of *configurators* which maintain dependencies among components and provide a set of hooks at which components can be attached or detached dynamically [11]. The adaptation approach seems to favour replacing

shared, platform-wide components (e.g., the scheduling strategy or the IOP protocol); it is not clear how finer-grained adaptation can be performed (e.g., changing the scheduling parameters of a thread of a specific binding), and the interface that triggers adaptation is generic and potentially unsafe. In [10], a component-based ORB architecture is presented that supports run-time reconfiguration on a per remote method invocation basis. The configuration is driven by declarative, application-specific policies. The presented meta-interface is thus very usable, but its power is restricted to selecting between alternative implementations of a given component type. Declarative meta-interfaces for specific CFs are, of course, also possible in our architecture.

COM+[13], Enterprise JavaBeans [18] and CORBA Components [14] all support similar container-based models for building distributed applications. The significance of these architectures lies in that they achieve a separation of concerns between the functional aspects of the application and the non-functional aspects that are managed by the container (distribution, concurrency, transactions, etc.). The drawback is that the configurability of these aspects is severely limited. The implementation of the container services is, essentially, hidden and out of the control of the application developer. Following our approach, we can engineer containers as CFs with associated meta-interfaces that open up the container’s internal machinery and expose the components responsible for handling those non-functional aspects.

## 7. Final Considerations

In this paper, we have presented an approach to the design of flexible middleware platforms that relies upon component technology and reflection. In particular, we proposed that both middleware and applications be built from components following the same lightweight, efficient component model. This is based on a popular commercial standard, COM, which we have enhanced with richer reflective facilities. Our OpenCOM-based middleware architecture is decomposed into multiple CFs/ CFRs, each associated with a specific aspect of middleware functionality. CFRs are first-class components and implement meta-interfaces. The architecture has a number of benefits: extensibility via independently developed components, a uniform programming model for applications and middleware, and the opportunity to use diverse architectural and meta-interface styles within the platform. Importantly, we exploit the fact that CFs enforce framework-wide rules and policies in order to constrain the dynamic adaptation permitted by meta-interfaces.

An implementation of the architecture, based on the GOPI platform, is underway. At the current stage of our

work, we are convinced that the component/ CF paradigm is a powerful and promising way of engineering adaptable and reflective middleware platforms. However, the design of appropriate CFs and meta-interfaces is a challenging task, and further work is needed before we can confidently propose sets of generally applicable guidelines/styles for CF development.

## Acknowledgements

The research described in this paper is funded by CNET, France Telecom (Grant 96-1B-239). We would also like to thank a number of people from Lancaster's Reflective Middleware Group who have contributed to the work described in this paper, including Anders Andersen, Lynne Blair, Fabio Costa, Hector Duran, Tom Fitzpatrick, Lee Johnston and Katia Saikoski.

## References

- [1] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu, "Coyote: A System for Constructing Fine-Grained Configurable Communication Services", ACM Transactions on Computer Systems, vol. 16, no. 4, November 1998, pp. 321-366
- [2] G.S. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware", Proc. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), Springer, 1998.
- [3] G.S. Blair, F. Costa, G. Coulson, H. Duran, N. Parlavantzas, F. Delpiano, B. Dumant, F. Horn, J.B. Stefani, "The Design of a Resource-Aware Reflective Middleware Architecture", Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), St-Malo, France, Springer-Verlag, LNCS, Vol. 1616, pp115-134, 1999.
- [4] F. Costa, H. Duran, N. Parlavantzas, K. Saikoski, G.S. Blair and G. Coulson. "The Role of Reflective Middleware in Supporting the Engineering of Dynamic Applications". In Walter Cazzola, Robert J. Stroud and Francesco Tisato, editors, Reflection and Software Engineering, Lecture Notes in Computer Science 1826. Springer-Verlag, 2000
- [5] G. Coulson, "A Distributed Object Platform Infrastructure for Multimedia Applications", Computer Communications, Vol. 21, No. 9, pp 802-818, July 1998.
- [6] B. Dumant, F. Dang Tran, F. Horn, and J.B. Stefani, "Jonathan: an open distributed processing environment in Java", Middleware'98, The Lake District, U.K., September 1998.
- [7] T. Fitzpatrick, G.S. Blair, G. Coulson, N. Davies and P. Robin, "Supporting Adaptive Multimedia Applications through Open Bindings", Proceedings of the 4th International Conference on Configurable Distributed Systems (ICDS'98), Annapolis, Maryland, U.S.A., 1998.
- [8] Ø. Hanssen, F. Eliassen, "A Framework for Policy Bindings", Proc. DOA'99, Edinburgh September 1999, IEEE Press
- [9] R. Hayton, A. Herbert, D. Donaldson, "Flexinet: a flexible, component oriented middleware system", Proceedings of the 8th ACM SIGOPS European Workshop: Support for Composing Distributed Applications, Sintra, Portugal, 7-10 September 1998
- [10] B.N. Joergensen, E. Truyen, F. Matthijs, W. Joosen. "Customization of Object Request Brokers by Application Specific Policies". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
- [11] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L.C. Magalhães, and R.H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB". IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000). New York. April 3-7, 2000.
- [12] T. Ledoux, "OpenCorba: a Reflective Open Broker," Reflection'99, Saint-Malo, France, Springer-Verlag, LNCS, Vol. 1616, 1999.
- [13] Microsoft, COM Home Page, <http://www.microsoft.com/com/default.asp> Last updated: December 29, 1999
- [14] Object Management Group, CORBA Components Final Submission, OMG Document orbos/99-02-05
- [15] Object Oriented Concepts, "ORBacus User Manual – Version 3.3.1", [www.ooc.com/ob](http://www.ooc.com/ob), 2000
- [16] D.C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware", IEEE Communications Magazine Special Issue on Design Patterns, April, 1999.
- [17] A. Singhai, A. Sane, and R. Campbell "Quarterware for Middleware", 18th IEEE International Conference on Distributed Computing Systems (ICDCS 1998). Amsterdam, The Netherlands. May 1998.
- [18] Sun Microsystems, Enterprise JavaBeans Specification Version 1.1, <http://java.sun.com/products/ejb/index.html>
- [19] C. Szyperski, "Component Software. Beyond Object-Oriented Programming", Addison Wesley, ISBN: 0-201-17888-5, 1997.
- [20] Y. M. Wang and Woei-Jyh Lee, "COMERA: COM Extensible Remoting Architecture," in Proceedings of COOTS, April 1998.