The Case for Reflective Middleware

Building Middleware that is Flexible, Reconfigurable, and yet Simple to Use.

Fabio Kon
Department of Computer Science
University of São Paulo
kon@ime.usp.br

Fabio Costa
Department of Computer Science
Federal University of Goiás
fmc@inf.ufg.br

Gordon Blair
Department of Computing
Lancaster University
gordon@comp.lancs.ac.uk

Roy H. Campbell
Department of Computer Science
Univ. of Illinois at Urbana-Champaign
roy@cs.uiuc.edu

Recent advances in distributed, mobile, and ubiquitous systems are creating new computing environments that are characterized by a high degree of dynamism. Variations in resource availability, network connectivity, and hardware and software platforms impact greatly the performance of user applications. The expected growth of ubiquitous computing will further change the nature of the computational infrastructure, bringing a plethora of small devices and requiring customized protocols and policies in order to fulfill the user's evolving quality of service requirements.

In the past ten years, software developers witnessed the creation of various middleware technologies whose goal is to facilitate the development of software systems. Middleware resides between the operating system and the application (thus its name), mediating the interactions between them. Technologies such as OMG's CORBA 3, Sun's Java-based J2EE, and Microsoft's .NET hide from the programmer the complicated details of network communication, remote method invocation, naming, and service instantiation, easing the construction of complex distributed systems. CORBA and Java also hide the differences among the underlying software and hardware platforms, increasing portability and facilitating maintenance as new versions of operating systems are released.

While conventional middleware technology aids the development of distributed applications for the new computing environments, it does not provide appropriate support for dealing with the dynamic aspects of the new computational infrastructure. Next generation applications require a middleware that can be adapted to changes in the environment and customized to fit into devices ranging from PDAs and sensors to powerful desktops and multicomputers [1, 2]. This article draws on the experience of two independent research projects on next generation middleware. We argue that the *reflective middleware* model is a principled and efficient way to deal with highly dynamic environments, supporting the development of flexible and adaptive systems and applications.

Why Reflective Middleware?

A major advantage of using middleware to develop software is that it hides the details of the underlying layers and operating system specific interfaces. Developers of distributed applications can write code that looks very similar to code for centralized applications; the middleware takes care of networking, marshalling, method dispatching, scheduling, etc. The code that runs on top of the middleware is easily portable and the programmer need not to worry about the internals of the operating system and of the middleware.

On the other hand, some applications can benefit greatly from knowing what is happening inside the underlying layers, in the computational environment, and in the physical environment. For example, a multimedia streaming or videoconferencing application can obtain dramatic improvements in its quality of service by selecting a network transport protocol that suits the underlying network infrastructure (e.g., wireless LAN, wired LAN, or long distance Internet) and the available bandwidth. It may also benefit from being aware of its physical context, detecting the presence of a wall display and reconfiguring the application to show the video in the larger display. An e-commerce web site can improve its response time by examining information about resource utilization and changing dynamically the location of its system components, creating replicas of its most requested services, or changing the middleware's request scheduling policies. A calendar application for ubiquitous computing can be more effective if its code can detect in what kind of hardware platform it is executing (e.g., PDA, wrist watch, desktop, or wall display) so that it can provide a graphical interface that is optimized for that platform.

In other words, most of the applications benefit from middleware that can hide the details of the underlying layers, but some applications can get very significant performance improvements by examining the dynamic state of the underlying layers and tuning the middleware implementation to its needs [2]. Therefore, what we need is a model of middleware that provides transparency to the applications that want it and translucency and fine-grain control to the applications that need it.

The Reflective Middleware Model

In the reflective model, the middleware is implemented as a collection of components that can be configured at application startup time. The middleware interface is unchanged and can be used by applications developed for traditional middleware. In addition, system and application code may also use *meta-interfaces* to inspect the internal configuration of the middleware and, if needed, reconfigure it to adapt to changes in the environment. In that manner, it is possible to select networking protocols, security policies, encoding algorithms, and various other mechanisms to optimize system performance for different contexts and situations.

In general terms, reflective middleware refers to the use of a *causally connected self-representation* to support the inspection and adaptation of the middleware system [1]. Thus, the same reflection techniques used in traditional areas, such as programming languages, apply to middleware as well (see Sidebars). By *self-representation*, we mean an explicit representation of the internal structure of the middleware implementation that is maintained and can be manipulated by it; one can also say that the middleware is *self-aware*. The self-representation is *causally connected* if changes in the representation lead to changes in the middleware implementation itself and, conversely,

changes in the middleware implementation lead to changes in the representation.

Unlike traditional middleware that is constructed as a monolithic black box, reflective middleware is organized as a group of collaborating components. This organization permits the configuration of very small middleware engines that are able to interoperate with traditional middleware. Conventional middleware implementations include all the functionality that any application may ever need; however, most of the times, applications use only a small subset of this functionality. The current difficulties in deploying standard middleware technologies to the small devices used in ubiquitous computing do not apply to component-based middleware. While conventional CORBA ORBs and Java virtual machines require several megabytes of memory, component-based reflective ORBs can have a memory footprint as little as 6KB [3].

In addition to the characteristics mentioned above, a reflective architecture must also provide support for customizing component behavior dynamically and for fine-grain resource management through system meta-interfaces (see the *Basic Reflection Terminology Sidebar*).

Case Studies

We now describe two different implementations of reflective middleware systems developed at the University of Illinois and at Lancaster University and explain how each implementation addresses the issues discussed above.

dynamicTAO

DynamicTAO [4] is an extension of the C++ TAO ORB [5], enabling on-the-fly reconfiguration of the ORB internal engine and of applications running on top of it. In *dynamicTAO*, *ComponentConfigurator*s represent the dependence relationships between ORB components and between ORB and application components. A *ComponentConfigurator* is a C++ object that stores the dependencies as lists of references, pointing to other component configurators, creating a directed dependence graph of ORB and application components as shown in Figure 1.

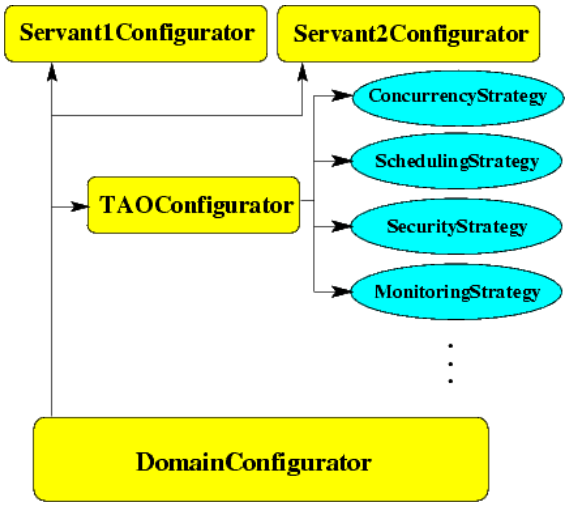

Figure 1: *dynamicTAO* component configurators

Whenever a request for replacing a component C arrives, the middleware examines the dynamic dependencies between C and other middleware and application components using the *ComponentConfigurator* object associated with C. Programmers can extend the *ComponentConfigurator* class by inheritance to provide customized implementations dealing with different kinds of components. Middleware developers use this feature to write the code that takes the proper actions to guarantee the consistency of the ORB internal structure in the presence of dynamic reconfigurations. *DynamicTAO* supports safe dynamic reconfiguration of the middleware components that control concurrency, security, and monitoring.

DynamicTAO exports a meta-interface for loading and unloading modules into the system runtime, and for inspecting and chang-

ing the ORB configuration state. This meta-interface can be accessed by developers for debugging and testing purposes, by system administrators for maintenance purposes, or by other software components that can inspect and reconfigure the internals of the ORB based on information collected from other sources, such as resource utilization monitors [6]. In addition, to support the reconfiguration of a large collection of distributed ORBs, *dynamicTAO* exports a similar meta-interface for mobile agents. In this case, system administrators use a graphical interface to build mobile agents and inject them in the network; the agents travel from ORB to ORB, inspecting and reconfiguring them according to the instructions the administrator programmed [4].

To allow dynamic interposition of application- or enterprise-specific code into the remote method invocation path, *dynamicTAO* provided support for interceptors since its first releases. More recently, the OMG defined a standard for portable interceptors [7], which is now part of TAO. Developers can install portable interceptors at the client and server sides and at the message or request levels. This facility can be used for supporting cryptography, compression, access control, monitoring, auditing, etc.

DynamicTAO delegates resource management to components that are not part of the basic middleware engine but that can be dynamically loaded into it. It employs the Dynamic Soft Real-Time Scheduler (DSRT) [8], which runs as a user-level process in conventional operating systems like Solaris, Linux, and Windows. DSRT uses the system's low-level real-time API to provide QoS guarantees to applications with soft real-time requirements. It performs QoS-aware admission control, resource negotiation, reservation, and real-time scheduling [6].

The mechanisms for reification, inspection and reconfiguration of the ORB internal engine that *dynamicTAO* adds to the conventional TAO implementation make it a reflective ORB.

Open ORB

The Open ORB project [9] aims at the design of highly configurable and dynamically reconfigurable middleware platforms to support applications with dynamic requirements, such as those involving distributed multimedia and mobility.

Components with well-defined interfaces implement the several elements of middleware functionality. Customized instances of the Open ORB platform can then be configured by assembling the appropriate components together, following a component model that allows for hierarchic composition and distribution. The Open ORB architecture preserves components as identifiable entities at runtime, which in turn facilitates runtime reconfiguration, as it eases identification of the parts of the platform that need to change.

Dynamic reconfigurability is achieved with the extensive use of reflection, with a clear separation between base- and meta-level. While the base-level consists of components that implement the usual middleware services, the meta-level comprises reflective facilities to expose such implementation, enabling inspection and adaptation. The structure of the meta-level follows the same component model used to define the base-level, which means that reflection can also be applied to inspect and adapt the meta-level itself. Meta-level components comprise a causally connected self-representation of the platform and are associated with the base-level components on an individual basis. Each base-level component may have its own private set of meta-level components, which are collectively referred to as the component's *meta-space*.

To tackle the complexity of the meta-level

architecture and to provide for manageable, yet comprehensive reflective interfaces, the meta-space of a component is defined according to a *multi-model reflection framework* [10]. The meta-space is partitioned into distinct *meta-space models*, which offer different views of the platform implementation and can be independently reified. Open ORB currently defines four meta-space models, which are grouped according to the distinction between structural and behavioral reflection, and are illustrated in Figure 2.

The *Interfaces* and *Architecture* meta-space models support structural reflection. The former model is concerned with the external representation of a component, in terms of the set of provided and required interfaces. The associated meta-object protocol (MOP) offers facilities to enumerate and search the elements of interface definitions, allowing, for instance, the dynamic discovery of the services a component provides. The *Architecture* meta-space model in turn is concerned with the internal implementation of components, in terms of its *software architecture*. The self-representation consists of two parts: a *component graph*, representing the interconnections between the components in a component assembly, and a set of *architectural constraints*, which define the rules to validate component assemblies. The associated MOP provides the ability to *inspect and adapt* the software architecture (e.g., to add, remove or replace components, and to inspect and change the constraints), enabling dynamic adaptation.

The *Interception* meta-space model supports behavioral reflection. The corresponding MOP enables the manipulation of non-functional properties, in the form of interceptors that perform pre- and post-processing of the interactions emitted and received at an interface. In addition to this form of behavioral reflection, the *Resources* meta-space model offers structured access to the underlying platform's resources and resource management [11]. The associated MOP allows the inspection and reconfiguration of the resources, such as storage and processing, allocated to particular activities in the system (e.g., by adding or removing resources, or changing the parameters and algorithms for resource management). In this way, resource allocation and properties can evolve to match the quality of service requirements of applications.

The Open ORB research group implemented prototypes of the architecture focusing on performance, management of meta-information, and resource management [9]. In each case, the researchers carried out tests that demonstrated the suitability of the architecture for the support of distributed multimedia applications.

Comparison

Open ORB and *dynamicTAO* were developed independently at different sides of the Atlantic by people with different backgrounds using different technologies. Nevertheless, their motivations were the same and both projects led to similar solutions based on reflective architectures.

These projects illustrate two opposite approaches for the development of reflective systems and, more specifically, reflective middleware. The development of *dynamicTAO* started with TAO, a complete implementation of a CORBA ORB that was modular but static. The *dynamicTAO* developers re-used tens of thousands of lines of code that were already functional and concentrated on adding reflective features to make the system more flexible, dynamic, and customizable. Conversely, the development of Open ORB started from scratch, its designers had the opportunity to plan its architecture from the earliest stages. Therefore, while *dynamicTAO* focused on code reuse and on leveraging existing systems, Open

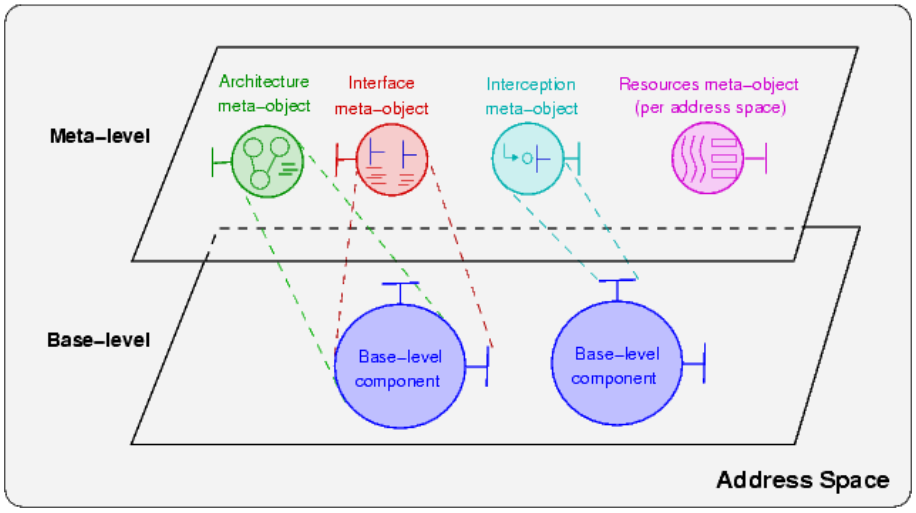

Figure 2: Structure of the meta-space in Open ORB

ORB focused on a novel middleware architecture where all the elements are consistent with the principles of reflection.

Conclusions and Future Directions

In the past two years, existing implementations of traditional middleware have been incorporating some of the contributions offered by research in reflective middleware. CORBA has now a standard for portable interceptors [7]. Orbix2000 allows the specification of different policies and supports dynamic loading of new components called plug-ins [12]. Despite the usefulness of these features, the degree of support for customization and dynamic adaptation is only partial, not covering all aspects of the design and the different phases of a platform's life cycle. This is mostly due to the inherent black-box nature of these technologies, which limits the extent to which elements of the design can be opened and exposed to the programmer. Reflection, on the other hand, offers a truly generic solution to the problem with a principled approach to middleware design that naturally renders itself to openness. Finally, the use of reflection permits the manipulation and adaptation of the different aspects of a platform in ways that were not anticipated during its design.

We believe that it is now time for the middleware community to get together to discuss the architecture of the next generation middleware technologies. Reaching an international consensus in this area and working for the emergence of standards for reflective middleware would be extremely beneficial.

References

[1] Geoff Coulson. What is Reflective Middleware? *IEEE Distributed Systems Online*, 2(8), 2001. `http://computer.org/dsonline/middleware/RMarticle1.htm`.

[2] Mark Astley, Daniel C. Sturman, and Gul Agha. Customizable Middleware for Modular Distributed Software. *Communications of the ACM*, 44(5):99–107, May 2001.

[3] Manuel Román, Dennis Mickunas, Fabio Kon, and Roy H. Campbell. LegORB and Ubiquitous CORBA. In

Proceedings of the IFIP/ACM Middleware'2000 Workshop on Reflective Middleware, pages 1–2, Palisades, NY, April 2000.

[4] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[5] Douglas C. Schmidt and Chris Cleeland. Applying Patterns to Develop Extensible ORB Middleware. *IEEE Communications Magazine Special Issue on Design Patterns*, 37(4):54–63, May 1999.

[6] Fabio Kon, Roy H. Campbell, M. Dennis Mickunas, Klara Nahrstedt, and Francisco J. Ballesteros. 2K: A Distributed Operating System for Dynamic Heterogeneous Environments. In *Proceedings of the 9th IEEE International Symposium on High Performance Distributed Computing (HPDC'9)*, pages 201–208, Pittsburgh, August 2000.

[7] OMG. The object management group. Home page: `http://www.omg.org`, 2002.

[8] Klara Nahrstedt, Hao hua Chu, and Srinivas Narayan. QoS-aware Resource Management for Distributed Multimedia Applications. *Journal of High-Speed Networking, Special Issue on Multimedia Networking*, 7:227–255, 1998.

[9] Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fábio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The Design and Implementation of Open ORB 2. *IEEE Distributed Systems Online*, 2(6), 2001.

[10] Hidehaki Okamura, Y. Ishikawa, and Mario Tokoro. AL-1/D: A Distributed Programming System with Multi-Model Reflection Framework. In *Proceedings of the International Workshop on New Models for Software Architecture (IMSA'92)*, pages 36–47, Tokyo, November 1992.

[11] Hector Duran-Limon and Gordon S. Blair. The Importance of Resource Management in Engineering Distributed Objects. In *Proceedings of the 2nd. International Workshop on Engineering Distributed Objects*, number 1999 in LNCS, pages 44–60, Davis, CA, November 2000. Springer-Verlag.

[12] IONA Technologies. *Orbix 2000*, 2000. White paper available at `http://www.iona.com`.

Fabio Kon (kon@ime.usp.br) is an assistant professor of Computer Science at the University of São Paulo, Brazil.

Fábio Costa (fmc@inf.ufg.br) is an assistant professor of Computer Science at the University of Goiás, Brazil.

Gordon Blair (gordon@comp.lancs.ac.uk) is a professor of Computer Science at the Lancaster University, UK.

Roy H. Campbell (roy@cs.uiuc.edu) is a professor of Computer Science at the University of Illinois at Urbana-Champaign.

This work is supported by CNPq-Brazil proc. 680037/99-3, FAPESP-Brazil proc. 01/03861-0, NSF-USA proc. 99-70139, and EPSRC-UK proc. GR/M04242.

SIDEBAR1: Basic Reflection Terminology

The foundations of reflective computing systems were originally laid out by Smith [1] and Maes [2] in the context of programming languages. In short, a system is reflective when it is able to manipulate and reason about itself in the same way as it does about its application domain. As a result of such introspective processing, a reflective system provides the ability to inspect and change itself during the course of its execution.

From its original application in programming languages, reflection gained wider acceptance in other areas, such as operating systems [3] and distributed systems [4]. This was based on the assumption that the same underlying principles seamlessly apply to these areas. Such assumption has also fueled the work on reflective middleware, as seen in the main body of this article. We now describe a few fundamental concepts of reflective systems.

- **reification:** the action of exposing the internal representation of a system in terms of programming entities that can be manipulated at runtime. The opposite process, *absorption*, consists in effecting the changes made to reified entities into the system, thus realizing the *causal connection* link.

- **meta-level architectures:** a reflective system has a meta-level architecture when it is explicitly structured in terms of a *base-level*, which deals with application concerns, and a *meta-level*, which deals with reflective computation.

- **meta-object and meta-object protocol (MOP):** in object-oriented reflective systems, the entities that populate the meta-level are called meta-objects. The interaction protocol the meta-objects support provides the reflective capabilities and is known as the meta-object protocol (MOP).

- **structural reflection:** the ability of a language (or system) to provide a complete reification of the program currently executing, for instance, in terms of its methods and state. This enables the programmer to inspect or change the functionality of the program and the way it models the domain.

- **behavioral reflection:** the ability of a language (or system) to provide a complete representation of its own semantics, in terms of internal aspects of its runtime environment. This enables the programmer to inspect or change the way the underlying environment processes the program, for example, with regard to non-functional properties and resource management.

References

[1] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, MIT Laboratory of Computer Science, 1982.

[2] Patie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, pages 147–155, Orlando, FL, October 1987.

[3] Y. Yokote. The Apertos Reflective Operating System: The Concept and its Implementation. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'92)*, pages 414–434, Vancouver, October 1992.

[4] Jeff McAffer. Meta-Level Architecture Support for Distributed Objects. In *Proceedings of Reflection'96*, pages 39–62, San Francisco, CA, 1996.

SIDEBAR2: Other Reflective Middleware Implementations

A number of researchers have developed middleware architectures that apply the concepts of reflection and meta-level architectures. Some examples are discussed below.

FlexiNet is a configurable ORB architecture aiming at environments of mobile Java objects [1]. FlexiNet defines an open distributed binding framework, structuring it in terms of modules in a protocol stack. It supports adaptation through a meta-object protocol that allows for reconfiguration of such modules to alter the properties of the binding.

OpenCORBA is a reflective implementation of CORBA in NeoClasstalk, a Smalltalk-like reflective language based on the concept of meta-classes [2]. The reflective features of OpenCORBA are based on the idea of modifying the behavior of a CORBA service by replacing the meta-class of the class defining that service.

Quarterware is a reflective middleware platform supporting multiple middleware standards such as CORBA, Java RMI, and MPI [3]. It uses a component framework for middleware, where the various ORB mechanisms are realized in terms of components. A reflective interface allows the programmer to plug customized versions of these components into the framework.

mChaRM is a reflective middleware platform that uses the communication reification approach to enable explicit control over multi-party communications [4]. The architecture is centered on *channels* as the main meta-level abstraction, which permits the interception of method calls to inspect and adapt their structure and behavior.

Besides the object-oriented approach of most reflective middleware projects, researchers have considered the use of the aspect-oriented programming (AOP) paradigm to structure middleware meta-level architectures. AOP extends the basic notion of separation of concerns in reflective systems (i.e., base- vs. meta-level) to a finer level of granularity, where multiple crosscutting concerns or *aspects* (at both base- and meta-level) can be implemented separately and yet be integrated into a cohesive system. In mainstream AOP research, aspects are not preserved at runtime as identifiable entities, thus hindering their use for dynamic adaptation. Nevertheless, other approaches for the realization of aspect-oriented systems have employed mechanisms such as composition filters [5] and fragmented components [6], which realize aspects in terms of first-class runtime entities. This opens the possibility for aspect-oriented reflective middleware, in which the design of the meta-level benefits from the greater separation of concerns that is typical of the AOP paradigm.

References

[1] Richard Hayton. FlexiNet Open ORB Framework. Technical Report 2047.01.00, APM Ltd., Cambridge, UK, October 1997.

[2] Thomas Ledoux. OpenCORBA: A Reflective Open Broker. In *Proceedings of Reflection'99*, number 1616 in LNCS, pages 197–214, St. Malo, France, July 1999. Springer-Verlar.

[3] Ashish Singhai, Ahmond Sane, and Roy H. Campbell. Quarterware for Middleware. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98)*, pages 192–201, Amsterdam, May 1998.

[4] Walter Cazzola and Mario Ancona. mChaRM: A Reflective Middleware for Communication-based Reflection. Technical Report DISI-TR-00-09, DISI, Universita degli Studi di Milano, Milan, May 2000.

[5] Lodewijk Bergmans and Mehmet Aksit. Aspects and Crosscutting in Layered Middleware Systems. In *Proceedings of the IFIP/ACM Middleware'2000 Workshop on Reflective Middleware (RM'2000)*, New York, April 2000.

[6] Franz Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. AspectIX - An Aspect-Oriented and CORBA-compliant ORB Architecture. Technical Report TR-I4-98-08, IMMD IV, Univesity of Erlangen, Nurnberg, Germany, September 1998.