Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell

The Case for Reflective Middleware

It's flexible and reconfigurable yet simple for programmers to use, notably for building dynamic distributed applications operating on the Net.

Recent advances in distributed, mobile, and ubiquitous systems demand new computing environments characterized by a high degree of dynamism. Variations in resource availability, network connectivity, and hardware and software platforms influence the performance of the related user applications. The expected growth of ubiquitous computing over the next five years will further alter the nature of the computational infrastructure, bringing a plethora of small devices and requiring customized protocols and policies to fulfill users' evolving quality of service (QoS) requirements.

During the past 10 years, software developers created middleware technology to facilitate development of software systems, most notably distributed and Internet-based, to support activities as diverse as scientific computation, information discovery and dissemination, and e-commerce. Middleware mediates interaction between the application and the operating system (hence its name). Related technologies, including the Object Management Group's CORBA, Sun Microsystems' 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 construction of complex distributed systems. CORBA, or Common Object Request Broker Architecture, and Java also hide the differences in the underlying software and hardware platforms, increasing portability and facilitating maintenance, as new versions of operating systems are released.

Despite aiding development of distributed applications, conventional middleware technology lacks support for the dynamic aspects of the new computational infrastructure. Next-generation applications require middleware that can be adapted to changes in the environment and customized to fit into devices, from PDAs and sensors to powerful desktops and multicomputers [1, 5]. Here, we reflect on how two independent research projects might influence the evolution of next-generation middleware. The reflective middleware model is a principled and efficient way of dealing with highly dynamic environments yet supports development of flexible and adaptive systems and applications.

TERRY MIURA

Hides the Details

Middleware hides the details of the underlying layers and operating-system-specific interfaces. Programmers of distributed applications can write code that looks similar to code written for centralized applications; the middleware extends method invocation by adding networking, marshalling, method dispatching, and scheduling. Applications written for middleware are easily ported, and programmers need not worry about the internals of the operating system or of the middleware itself.

On the other hand, some applications can benefit from exploiting the underlying layers in the physical environment, as well as in the computational environment. For example, a multimedia streaming or videoconferencing application can dramatically improve QoS by selecting a network transport protocol that suits the underlying network infrastructure (whether wireless LAN, wired LAN, or long-distance Internet), as well as the available bandwidth. It may also benefit from being aware of its physical context, detecting the presence of, say, a wall display and reconfiguring the application to show the video in the larger display. As another example, an e-commerce Web site might improve its response time by examining information about resource utilization, then dynamically changing the location of its system components, creating replicas of its most requested services or changing the middleware's request scheduling policies. As a third example, an effective calendar application for ubiquitous computing might detect the kind of hardware platform on which it is executing, possibly a PDA, wristwatch, desktop PC, or wall display, enabling it to provide a graphical interface optimized for that platform.

Most applications clearly benefit from middleware that hides the details of the underlying layers; but other applications can significantly improve their performance through interaction with the dynamic state of the underlying layers, tuning the middleware implementation to their requirements [1]. A desirable middleware model provides transparency to the applications that want it and translucency and fine-grain control to the applications that need it.

Reflective middleware model. In the reflective model, middleware is implemented as a collection of components that can be configured and reconfigured by the application. The middleware interface remains unchanged and may support applications developed for traditional middleware. In addition,

Basic Reflection Terminology

The foundations of reflective computing systems were originally laid out in [12] in the context of programming languages. A system is reflective when it is able to manipulate and reason about itself the same way it does about its application domain. As a result of such introspective processing, a reflective system is able 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, including operating systems and distributed systems. This appeal followed the assumption that the same underlying principles seamlessly apply to these other areas, too. The assumption has also fueled work on reflective middleware involving a few fundamental concepts:

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 of 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 dealing with application concerns and a meta-level dealing with reflective computation.

Meta-objects and the meta-object protocol (MOP). In object-oriented reflective systems, the entities populating the meta-level are called meta-objects. The interaction protocol supported by meta-objects provides reflective capabilities and is known as the meta-object protocol.

Structural reflection. This ability of a language (or system) provides a complete reification of the program currently executing in terms of, say, the language's methods and state. The programmer can inspect or change the functionality of the program, as well as the way it models the domain.

Behavioral reflection. This ability of a language (or system) provides a complete representation of the language's own semantics in terms of the internal aspects of its runtime environment. Programmers can inspect or change the way the underlying environment processes a program with regard for, say, nonfunctional properties and resource management. C system and application code may inspect the internal configuration of the middleware and, if needed, reconfigure it to adapt to changes in the environment through meta-interfaces. In this manner, it is possible to select networking protocols, security policies, encoding algorithms, and various other mechanisms to optimize system performance for specific and often unpredictable 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 [5]. The same reflection techniques used in, say, programming languages also apply to middleware (see the sidebar "Basic Reflection Terminology"). Self-representation refers to an explicit representation of the internal structure of the middleware implementation that the middleware maintains and manipulates. In this limited sense, 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 middleware constructed as a monolithic black box, reflective middleware is organized as a group of collaborating components. This principle permits the configuration of very small middleware engines that interoperate with conventional middleware. Such middleware implementations often include all the functions applications might need; however, in most cases a particular application may use only a small subset of this functionality. The current difficulties deploying standard middleware technologies to the small devices used in ubiquitous computing do not apply to component-based middleware. While conventional CORBA object request brokers (ORBs) and Java virtual machines require several megabytes of memory, component-based reflective ORBs have memory footprints as small as 6KB [10].

In addition to these properties, a reflective architecture must also support the dynamic customization of component behavior and fine-grain resource management through meta-interfaces. Two different implementations—DynamicTAO and Open ORB—of reflective middleware systems developed at the University of Illinois and at Lancaster University, respectively, each addresses these issues in its own ways.

DynamicTAO. DynamicTAO [6] is an extension of the C++ TAO ORB (see www.cs.wustl.edu/schmidt/TAO.html), enabling on-the-fly reconfiguration of the ORB internal engine and of applications running on top of the ORB. In DynamicTAO, ComponentConfigurators represent the dependence relationships between ORB components and between ORBs and application components. A ComponentConfigurator is a C++ object that stores these dependencies as lists of references pointing to other component configurators, thus creating a directed dependence graph of ORB and application components (see Figure 1).

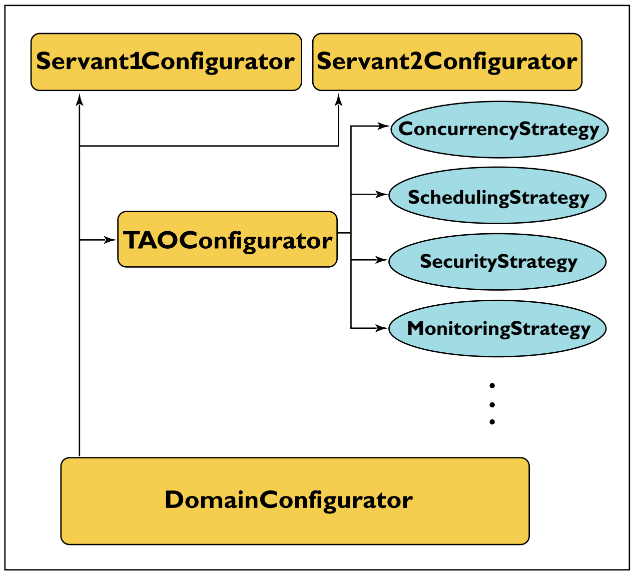

Figure 1. DynamicTAO component configurators.

Whenever a request for the replacement of 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 extend the ComponentConfigurator class through inheritance to provide customized implementations dealing with different kinds of components. Middleware developers use this feature to write 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 controlling concurrency, security, and monitoring.

DynamicTAO exports a meta-interface for loading and unloading modules into the system runtime, and for inspecting and changing the ORB configuration state. The meta-interface is available to developers for debugging and testing purposes, to system administrators for maintenance purposes, and to other software components that can inspect and reconfigure the internals of the ORB based on information collected from other sources, including resource utilization monitors. In addition, to support the reconfiguration 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 into the network; the agents travel from ORB to ORB, inspecting and reconfiguring each one according to the instructions the administrator has programmed into the agents [6].

To allow dynamic interposition of application- or enterprise-specific code into the remote method invocation path, DynamicTAO has provided support for interceptors since its earliest release in 1999. More recently, the OMG defined a standard for portable interceptors that is now part of TAO (see www.omg.org). Developers can install portable interceptors at the client or server sides and at the message or request levels. They can use interceptors to support cryptography, compression, access control, monitoring, and auditing.

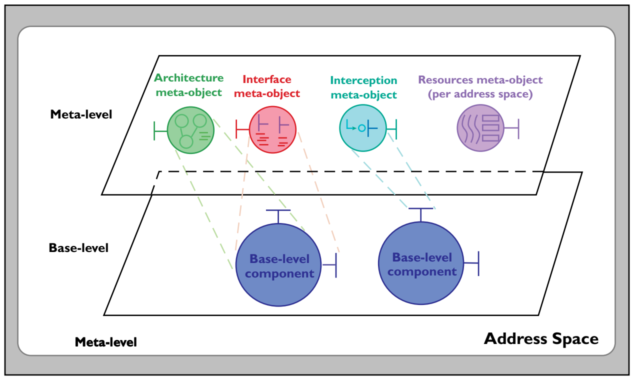

Figure 2. Structure of the meta-space in Open ORB.

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

The mechanisms for reification, inspection, and reconfiguration of the ORB internal engine added by DynamicTAO to the conventional TAO implementation make DynamicTAO a reflective ORB.

Other Reflective Middleware Implementations

A number of researchers have developed middleware architectures that apply the concepts of reflection and meta-level architectures, as in the following examples:

OpenCORBA. This reflective implementation of CORBA is written in NeoClasstalk, a Smalltalk-like reflective language based on meta-classes [7]. 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. This early reflective middleware platform supports multiple middleware standards, including CORBA, Java RMI, and MPI [11]. It uses a component framework in which the various ORB mechanisms are realized in terms of components. A reflective interface allows programmers to plug customized versions of these components into the framework.

multi-Channel Reification Model (mChaRM). This experimental reflective middleware platform uses the communication reification approach to enable explicit control over multi-party communication [4]. The architecture is centered on channels as the main meta-level abstraction and permits the interception of method calls to inspect and adapt the channels' structures and behavior.

Besides the object-oriented approach of most reflective middleware projects, researchers have also considered the aspect-oriented programming (AOP) paradigm to structure middleware meta-level architectures. AOP extends the basic notion of the separation of concerns in reflective systems, or base-level vs. meta-level, to a finer level of granularity; multiple crosscutting concerns, or aspects, (at both base-level and meta-level) can be implemented separately yet can also 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. Other approaches for developing aspect-oriented systems have employed such mechanisms as composition filters [2] and fragmented components that realize aspects in terms of first-class runtime entities. These new research initiatives open the possibility of aspect-oriented reflective middleware; the design of the meta-level benefits from the greater separation of a system's various aspects into different parts, typical of the AOP paradigm. C

Open ORB. The Open ORB project aims to design highly configurable and dynamically reconfigurable middleware platforms to support applications with dynamic requirements, including those involving distributed multimedia and mobility [3]. Components with well-defined interfaces implement the elements of middleware functionality. Customized instances of the Open ORB platform can then be configured by assembling the appropriate components, following a component model that allows hierarchical composition and distribution. The Open ORB architecture preserves components as identifiable entities at runtime, facilitating runtime reconfiguration as it eases identification of the parts of the platform that need to change.

Dynamic reconfigurability is achieved through the extensive use of reflection, with a clear separation between the base-level and the meta-level. While the base-level consists of components implementing the usual middleware services, the meta-level comprises reflective facilities to expose these implementations to the programmer, enabling inspection and adaptation. The structure of the meta-level follows the same component model used to define the base-level, so reflection can 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 individual base-level components. Each base-level component may have its own private set of meta-level components, collectively referred to as the component's meta-space.

A DESIRABLE MIDDLEWARE MODEL PROVIDES TRANSPARENCY TO THE APPLICATIONS THAT WANT IT AND TRANSLUCENCY AND FINE-GRAIN CONTROL TO THE APPLICATIONS THAT NEED IT.

To tackle the complexity of the meta-level architecture and provide for manageable yet comprehensive reflective interfaces, the meta-space of a component is defined according to a multi-model reflection framework [9]. The meta-space is partitioned into distinct meta-space models that offer different views of the platform implementation and can be independently reified. Open ORB defines four meta-space models grouped according to the distinction between structural and behavioral reflection (see Figure 2).

The *Interfaces* and *Architecture* meta-space models support structural reflection. The former is concerned with the external representation of a component in terms of the set of provided and required interfaces. The associated meta-object protocol, or MOP, offers facilities to enumerate and search the elements of interface definitions, allowing, say, the dynamic discovery of the services provided by a particular component. The latter is in turn concerned with the internal implementation of components in terms of their 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 defining the rules needed to validate component assemblies. The associated MOP helps inspect and adapt the software architecture to, say, add, remove, or replace components and change constraints, thus enabling dynamic adaptation.

The *Interception* meta-space model supports behavioral reflection. The corresponding MOP enables manipulation of nonfunctional properties in the form of interceptors performing pre- and post-processing of the interactions emitted from and received at an interface. In addition to this behavioral reflection, the *Resources* meta-space model offers structured access to the underlying platform's resources and resource management. The MOP allows inspection and reconfiguration of the resources (such as storage and processing) allocated to particular activities in the system; it might do so by adding or removing resources or changing the parameters and algorithms needed for resource management. With the *Resources* meta-space model, resource allocation and properties can evolve to match an application's QoS requirements.

Over the past several years, the Open ORB research group has implemented prototypes of this architecture, focusing on performance, management of meta-information, and resource management [3]. In each case, the researchers tested the suitability of the architecture for distributed multimedia applications.

Similar motivations, similar solutions. Open ORB and DynamicTAO were developed independently on opposite sides of the Atlantic Ocean by people with different backgrounds using different technologies. Nevertheless, their motivations were the same, and both projects have yielded similar solutions based on reflective architectures (see the sidebar "Other Reflective Middleware Implementations").

Open ORB and DynamicTAO illustrate two opposite approaches to developing 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. Its developers reused tens of thousands of lines of functional code and concentrated on adding reflective features to make the system more flexible, dynamic, and customizable. Conversely, Open ORB development started from scratch, focusing on a novel middleware architecture whereby all the elements are consistent with the principles of reflection.

During the past few years, existing implementations of traditional middleware have incorporated some of the innovations produced through research in reflective middleware. For example, CORBA now includes a standard for portable interceptors. And the Orbix2000 commercial ORB allows the specification for such policies as security, transactions, and communication, while supporting the dynamic loading of new components, or plug-ins (see www.iona.com). Despite the usefulness of these features, support for customization and dynamic adaptation in mainstream middleware systems does not cover all aspects of a platform's life cycle. This limitation is due mostly to the inherent black-box nature of the technologies, limiting the extent 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 yields openness. Finally, reflection permits the manipulation and adaptation of the different aspects of a platform in ways not anticipated during its design.

The middleware community today should be discussing the architecture of next-generation middleware technologies. Reaching an international consensus in this area and working toward standards for reflective middleware would be extremely beneficial for researchers, software developers, system administrators, and, ultimately, users. C

References

1. Astley, M., Sturman, D., and Agha, G. Customizable middleware for modular distributed software. *Commun. ACM 44,* 5 (May 2001), 99–107.
2. Bergmans, L. and Aksit, M. Aspects and crosscutting in layered middleware systems. In *Proceedings of the IFIP/ACM (Middleware2000) Workshop on Reflective Middleware* (Palisades, NY, Apr. 7–8, 2000), 23–25.
3. Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., and Saikoski, K. The design and implementation of Open ORB 2. *IEEE Distrib. Syst. Online 2,* 6 (Sept. 2001); see computer.org/dsonline.
4. Cazzola, W. *Communication-Oriented Reflection: A Way to Open Up the RMI Mechanism.* Ph.D. thesis, Universita degli Studi di Milano, Italy, Nov. 2000.
5. Coulson, G. What is reflective middleware? *IEEE Distrib. Syst. Online 2,* 8 (Dec. 2001); see computer.org/dsonline/middleware/ RMarticle1.htm.
6. Kon, F., Roman, M., Liu, P., Mao, J., Yamane, T., Magalhaes, L., and Campbell, R. 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 (Middleware2000)* (Palisades, NY, Apr. 3–7). Springer-Verlag, Heidelberg, Germany, 2000, 121–143.
7. Ledoux, T. OpenCORBA: A reflective open broker. In *Proceedings of Reflection'99* (St. Malo, France, July 19–21). Springer-Verlag, Heidelberg, Germany, 1999, 197–214.
8. Nahrstedt, K., Chu, H., and Narayan, S. QoS-aware resource management for distributed multimedia applications. In the special issue on multimedia networking, *J. High-Speed Network. 7,* 3 (Dec. 1998), 227–255.
9. Okamura, H., Ishikawa, Y., and Tokoro, M. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on New Models for Software Architecture* (Tokyo, Nov. 4–7, 1992), 36–47.
10. Roman, M., Mickunas, D., Kon, F., and Campbell, R. LegORB and Ubiquitous CORBA. In *Proceedings of the IFIP/ACM (Middleware2000) Workshop on Reflective Middleware* (Palisades, NY, Apr. 7–8, 2000), 1–2.
11. Singhai, A., Sane, A., and Campbell, R. Quarterware for middleware. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS'98)* (Amsterdam, The Netherlands, May 26–29). IEEE Press, Los Alamitos, CA, 1998, 192–201.
12. Smith, B. *Reflection and Semantics in a Procedural Language.* Ph.D. thesis, MIT Laboratory for Computer Science, Cambridge, MA, 1982.

Fabio Kon (kon@ime.usp.br) is an assistant professor of computer science at the University of São Paulo, Brazil.
Fabio 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, U.K.
Roy H. Campbell (roy@cs.uiuc.edu) is a professor of computer science at the University of Illinois, Urbana-Champaign.

This work is supported by National Research Council (CNPq) (Brazil) proc. 680037/99-3, Sao Paol State Agency for REsearch Funding (FAPESP) (Brazil) proc. 01/03861-0, National Science Foundation (U.S.) proc. 99-70139, and Engineering and Physical Sciences Research Council (EPSRC) (U.K.) proc. GR/M04242.