

# Middleware Architecture Design Based on Aspects, the Open Implementation Metaphor and Modularity

H.-Arno Jacobsen  
University of Toronto  
Department of Electrical and Computer Engineering and  
Department of Computer Science  
jacobsen@eecg.toronto.edu \*

## 1 Introduction

A "middleware system" constitutes a set of services that aim at facilitating the development of distributed applications in heterogeneous environments. The primary objectives of a middleware are to foster application portability and distributed system interoperability. At least conceptually, the "middleware layer" comprises a layer below the application and above the operating system and network substrate. Common middleware platforms include CORBA, DCOM, and the Java suite of protocols.

The key design principal underlying middleware design has been to hide and encapsulate system details behind common abstractions and offer various dimensions of transparency to the application developer (e.g., with respect to object location, data access, and service implementation language). As systems code (or near systems code) standard middleware has not been designed with *extensibility*, *modularity*, *configuration*, *openness*, and *customization* in mind. It is, for instance, impossible to enhance a platform with techniques that address network awareness, mobile awareness, and dynamic adaptability, or to exploit application level semantic for system operation. Commonly, the argument against such a design is a postulated sacrifice of performance.

Some of these issues have been partially recognized by bodies defining middleware specifications, more notably the OMG, who has introduced a number of features addressing this dilemma. This includes, for instance, message and method level interceptors, hooks for custom marshalling (restricted to value types), open stub-to-orb interface (restricted to the Java language mapping), and pluggable transports (restricted to the real-time CORBA profile, although often implemented by standard ORBs). These enhancements leverage part of the problem, but leave much to be desired. Extending ORBs with language mappings, protocol mappings, object adaptors, general custom marshalling, smart

---

\* This work is supported by NSERC. Position Paper in Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster, UK, August 2001.

proxies, or techniques to achieve network awareness and dynamic adaptability to run-time conditions is still not a given (see [8, 5] for a more detailed discussion of some of these issues.)

Moreover, middleware systems are extended to address requirements of the most diverse application domains and new middleware services supporting application transcending requirements are constantly added. The application domains addressed by the CORBA platform alone, range from (distributed) business applications (standard CORBA), real-time systems (Real Time CORBA), and scientific and high performance computing (Data Parallel CORBA), to wireless and embedded systems (Minimum CORBA). The service sets extend from fault tolerance, security, and object group management to synchronous and asynchronous communication, to just name a few.

*Middleware platform families* are evolving to support these different application domain requirements within the same *platform line* (i.e., essentially product line). Different profiles (i.e., specialized subsets of the core model) are defined to address service sets and specific domain requirements. Similar efforts are emerging for Java-based middleware, and to some extent for DCOM-based systems. This proliferation as well as the aim to support a vast functional spectrum within one environment is leading to the co-existence of middleware platforms with many non-orthogonal features and considerable functional overlap.

To summarize, the key problem with current middleware systems is non-extensibility, proliferation with respect to supported features and application domains, and a too coarse-grained service model exhibiting functional redundancy.

While we cannot offer a full solution to these problems, we intend to outline our position towards a new architecture for middleware systems. We first discuss a number of principles to address shortcomings in current middleware architectures and then present potential solutions to the above outlined problems. The new middleware architecture will be based on three key concept, *modularity*, *aspect orientation*, and the *open implementation* metaphor. We also summarize related work that has addressed some of these issues.

## 2 Our Position: Towards an Improved Middleware Architecture Model

Our position is that, for the above outlined reasons, the design of a middleware platform should be re-thought from ground up and the hypothesis of whether or not an efficient modular platform cannot be developed, must be tested. We first discuss a set of principles a new middleware platform design should be based upon and then discuss how to achieve this with emerging development concepts.

### Middleware Design Principles

1. *Micro-kernel like architecture:*

Current middleware architectures favor one primary communication style – synchronous RPC – as basis and build other styles, as secondary add-on services next to it<sup>1</sup> (e.g., asynchronous communication, non-continuous operations, reliable / unreliable multicast, publish / subscribe, group communication etc.) A major drawback of this approach in current platform implementations is the overlap of the different models, the replication of core functional units that are part of all communication style, the subtle difference in the models (as they were incrementally developed over years), and the complexity of the final programming model supporting several non-orthogonal overlapping features.

To address these issues, a new middleware architecture should provide a model in which all communication styles are equally placed side-by-side and modularly broken down into functional units. Different communication styles are incrementally obtained from the composition of underlying functional units. In this model no one style is favored in the architecture over another one, functional redundancy is avoided, and consistency is maintained. We refer to this as a micro-kernel like design to emphasize that there should only be a very minimal core onto which different communication styles are to be configured.

2. *Open and modular middleware platform design:*

All platform functionalities, including the non-application exposed middleware service layers, should become accessible, open and modular building blocks. This refers to platform units such as communication sub-system, s-tubs, skeletons, ORB transport layer, interface repository, implementation repository, and all parts of the object adapter.

3. *All functional units in the platform should be accessible, customizable, and replaceable by custom implementations:*

A functional unit refers to one step in the execution chain of a service provided by the middleware (e.g., this could be one of the building blocks referred to in the item above). The exact extent of a step depends on the concrete platform design and the service offered. For a synchronous method invocation the functional units may be data structure packing, argument marshalling (unpacking and unmarshalling respectively), system-level RPC (possibly broken down in its constituent protocols), and call dispatching. These units should not be too coarse, to enable reuse, ease of customization, and low level access to their end-points (needed as join points for aspects). Different functional unit implementations, specialized for various environmental conditions and usage patterns, may be included with the platform to be selected at configuration time or provided as custom extensions. This unit breakdown is also required to enable very low-level aspects to be woven around unit end-points. Platform aspects such as, exception management, monitoring, authentication, encryption / decryption, and selective type support will need this modular break down as well.

---

<sup>1</sup> Not around or on top of it, i.e., these services are commonly stand alone implementations.

4. *Fine-grained configuration of platform:*

To date middleware systems are not configurable at all. For the most part, a one size fits all approach is taken. Under fine-grained configuration we understand the specialization of a platform instance for one particular application domain and, potentially, for one particular application in this domain. A higher level specification of the aspects the platform requires and the interfaces (e.g., required services, types, and exceptions) of the application guide the synthesis of the platform instance.

5. *Independence of accompanying platform tools:*

By platform tools we mean stub-/ skeleton-generator and meta-data repositories (IDL and implementation repositories). Current middleware platforms tightly integrate these tools into the platforms. This refrains from third party implementations of these components, use of standard technology, or the use of these components for purposes they were not immediately intended for. These tools should be separable from the platform, build on open interfaces, and follow an extensible design themselves.

### **Towards Realizing these Principles:**

We think that all of the above principles can be achieved by defining an open framework of a middleware platform, by using techniques from aspect-oriented programming for the generative configuration of middleware platform instances, and applying the open implementation metaphor to functional unit design to obtain customization at a very low level.

**The Role of an Open Middleware Platform Framework:** The framework lays out the middleware architecture, defines all platform interfaces, defines execution chains and access points (i.e., interfaces to individual steps in an execution chain), and defines how they inter-relate. This framework is the principal prerequisite for all the rest. It defines the level at which aspects can intervene and their granularity. It also impacts the possible design choices for all functional units.

**The Role of Aspects:** An aspect is a system feature that cross-cuts the implementation of the system and is manifest at multiple loci in the code [9]. Examples of aspects in the context of middleware are:

- exception management (raising, propagation, handling)

If an exception were to be defined as a system aspect and a middleware platform could be configured for a particular application (domain), the exception aspect could be configured in or configured out, depending on the application semantic and runtime environment.<sup>2</sup>

---

<sup>2</sup> When the Minimum CORBA (embedded system profile) standard was conceived a discussion of whether or not to include exception handling came up. Opponents

- synchronization management and concurrency control  
Synchronization constraints have often been used to illustrate aspect oriented programming techniques [9].
- interface definition language extensions  
Many extensions for interface definition languages have been proposed. This includes, for example, quality of service annotations, real-time constraints, assertions, pre and post conditions, and behavioral annotations. All of these constitute aspects.
- access control and security  
Access control (e.g., object-based, method-based, interface-based) and security (e.g., authentication and encryption / decryption) constitute examples of middleware aspects that may need to be blended in or out depending on the execution context.
- computing and network resource monitoring
- individual platform types  
A middleware platform supports a certain type model, ranging from basic types (e.g., int, float, char) to very sophisticated dynamically managed types (e.g., type ANY in the CORBA model). The processing of a type intervenes at different levels of the platform (e.g., in marshalling, in packaging, in transport etc.) In that sense a particular type constitutes an aspect of the middleware.

The above list comprises concrete aspects, there also exists a number of more abstract aspects, that will require a more refined decomposition. These include real-time, quality of service, and fault tolerance aspects.

Aspects could intervene at several stages in middleware platforms. For one, they could be used to extend the platform with certain features (cf. list above), but, more interestingly, aspects could be applied to configure a platform instance for a particular application domain and specific application.

The great benefit of this is that the proliferation of static platform profiles would disappear and, as new requirements (domain and feature) arise, simply more aspects need to be added. Clearly, this approach is entirely based on a modular open middleware platform framework that permits to define access points at which aspects can intervene.

**The Role of the Open Implementation Metaphor:** The open implementation metaphor reveals, at a function's interface, details about its implementation [10]. A client of the interface may influence the underlying implementation according to its usage pattern (this may be decided statically or dynamically depending on the sophistication of the open implementation based design of the function's implementation.) This concept can be applied to customize functional

---

argued that in the context of embedded systems exceptions were not needed and the primary design goal would be a small memory footprint. Similarly, other features of the platform where at scrutiny (e.g., various types of the CORBA type model, dynamic invocation support etc.)

units of the middleware platform. OI suggests several different levels of doing that, ranging to client provided implementation of the unit as final instance.

### 3 Related Work

Related work on the topics discussed in this position paper can be broadly classified into approaches that provide *customization* through static or dynamic policy selection, *reflection* to adapt middleware internals to changing runtime conditions, and *configuration* based on various forms of aspect definitions. Much of the discussed projects use several of these techniques. We briefly discuss some of them below.

Astley *et al.* [1] achieve middleware customization through techniques based on separation of communication styles from protocols and a framework for protocol composition. Further aspects that cross cut the system implementation are not explicitly addressed.

Several projects exploit reflective programming techniques to allow the middleware platform to adapt itself to changing runtime conditions. This includes projects such as openORB [2], openCORBA [12], and dynamicTAO [11]. Recent progress in this area has been summarized in a reflective middleware workshop<sup>3</sup>.

LegORB<sup>4</sup> [13] and Universally Interoperable Core (UIC)<sup>5</sup> are middleware platforms designed for hand-held devices, which allow for interoperability with standard platforms. Both offer static and dynamic configuration and aim to maintain a small memory footprint by only offering the functionality an application actually needs. Customizable functions range from the transport protocol to method dispatching and marshalling. Both platforms do not support the notion of aspects as code cross cutting concerns. Aspects in the sense of LegORB and UIC are functional units supporting application-level requirements.

Similarly, Jonathan<sup>6</sup> constitutes an open middleware framework that can be customized with respect to a large number of functions. Jonathan aims to embrace several standard middleware platforms and offer customization according to application needs. It can be configured to use IIOP or RMI.

The effectiveness of the open implementation metaphor for the design of a light-weight thread package is demonstrated by Haines [4]. The renewed design leads to a more efficient and portable package. While this approach is not primarily addressing middleware platform issues per se, it may also prove effective for the design of functional units within a platform.

Brodsky *et al.* [3] demonstrate very elegantly the use of aspect oriented programming for customization and extensibility of a distributed file system middleware with fault tolerance features.

Jacobsen and Krämer [8] show how to weave interface level specification of synchronization constraints into stubs and skeletons generated by standard IDL

<sup>3</sup> <http://www.comp.lancs.ac.uk/computing/rm2000/>

<sup>4</sup> <http://devius.cs.uiuc.edu/2k/LegORB/>

<sup>5</sup> <http://www.ubi-core.com/>

<sup>6</sup> <http://www.objectweb.org/jonathan/jonathanHomePage.htm>

compilers. This work is extended in Jacobsen and Krämer [6] to also account for other aspects defined at the interface level (e.g., QoS annotations, behavioral annotations, and pre and post conditions). A processing framework based on the extended markup language (XML) for this approach is presented in [7].

## References

1. M. Astley, D. C. Sturman, and G. A. Agha. Customizable middleware for modular software. *ACM Communications*, 44(5), May 2001.
2. G. S. Blair, G. Coulson, A. Andersen, M. Clarke, F. M. Costa, H. A. Duran, R. Moreira, N. Parlavantzas, and K. B. Saikoski. The design and implementation of OpenORB version 2. *IEEE Distributed Systems Online Journal*, 2(6), 2001.
3. Alex Brodsky, Dima Brodsky, Ida Chan, Yvonne Coady, Jody Pomkoski, and Gregor Kiczales. Aspect-oriented incremental customization of middleware services. Submitted.
4. Matthew Haines. An open implementation analysis and design for lightweight threads. In *OOPSLA*, pages 229 – 242, 1997.
5. H.-A. Jacobsen. Programming language interoperability in distributed computing environments. In Lea Kutvonen, Harmunt Knig, and Martti Tienari, editors, *Second IFIP Working Conference on Distributed Applications and Interoperable Systems II (DAIS)*, Helsinki, Finland, June 1999. Kluwer Academic Publisher.
6. H.-A. Jacobsen and B. Krämer. Design patterns for synchronization adaptors of CORBA objects. *Special issue of L'OBJECT Journal on "Object Orientation and Formal Methods"*, 2000. Hermes Publisher.
7. H.-A. Jacobsen and B. Krämer. Modeling interface definition language extensions. In *37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37)*, Sydney, Australia, 20-23 November 2000.
8. H.-A. Jacobsen and B. J. Krämer. A design pattern based approach to generating synchronization adaptors from annotated IDL. In *IEEE Automated Software Engineering Conference (ASE'98)*, pages 63–72. IEEE Computer Society, September 1998.
9. G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4), Dec 1996.
10. Gregor Kiczales, John Lamping, Christina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *international conference on Software engineering*, pages 481 – 490, 1997.
11. 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.
12. T. Ledoux. OpenCorba: A reflective open broker. *Lecture Notes in Computer Science*, 1616:197–??, 1999.
13. Manuel Roman, M. Dennis Mickunas, Fabio Kon, and Roy Campbell. LegORB and ubiquitous CORBA. Reflective Middleware Workshop. Held in conjunction with Middleware 2000. <http://www.comp.lancs.ac.uk/computing/rm2000/>, 7th-8th April 2000.