Next Generation Middleware: Requirements, Architecture, and Prototypes

Frank Eliassen[*], Anders Andersen[†], Gordon S. Blair[‡], Fabio Costa[‡], Geoff Coulson[‡],
Vera Goebel[*], Øivind Hansen[**], Tom Kristensen[*], Thomas Plagemann[*],
Hans Ole Rafaelsen[†], Katia B. Saikoski[‡], Weihai Yu[†]
[*]*Department of Informatics, University of Oslo , Norway*
[‡]*Distributed Multimedia Research Group, Computing Department, Lancaster University, UK*
[†]*Department of Computer Science, University of Tromsø, Norway*
[**]*Faculty of Engineering, Agder College, Grimstad, Norway*
frank@ifi.uio.no

Abstract

Middleware technologies such as CORBA or Java RMI have proved their suitability for "standard" client-server applications. However, challenges from existing and new types of applications, including support for multimedia, real-time requirements and mobility seems to indicate the need for defining a new architecture for open distributed systems. The new architecture should be designed from the beginning with flexibility and adaptability in mind. This can be achieved by defining an open engineering middleware platform that is run time configurable and allows inspection and adaptation of the underlying components.

This paper proposes a next generation middleware architecture that conforms to requirements as indicated above. This architecture is characterised by being open, and adaptable based on the principle of reflection. The paper also reports on some existing research prototypes with a focus towards their suitability as next generation middleware.

1. Introduction

The availability of multimedia personal computers and mobile computing devices, Internet services, mobile networks, and high-speed networks is drastically increasing the use (and need) of distributed multimedia applications for commercial, private, and other purposes. Such applications range from distance education applications for teaching and training to integrated command and control systems in naval defense vehicles.

Middleware technologies such as CORBA or Java RMI have proved their suitability for "standard" client-server applications. However, the middleware has to remain responsive to challenges new types of applications, including support for multimedia, real-time and mobility. It is well known in the research community that today's middleware solutions like CORBA implementations or Microsoft's DCOM are not suited for distributed multimedia systems. For example, there is no explicit representation of communication and no QoS support as required by multimedia applications. Furthermore, such platforms do not provide the required levels of adaptation and configurability that is needed to accommodate the diversity of modern distributed applications.

OMG is aware of these shortcomings, and a number of Request for Proposals and specifications have been issued to address themes such as "Minimal CORBA", "Real-time CORBA", etc. For example, both real-time CORBA and Minimal CORBA provide fixed solutions for given application domains, and interceptors provide a limited ad hoc approach to adaptation. In the same way, the Java technology is being adapted and extended in various ways to meet the needs of various domains.

However, in general there is no principled approach to these adaptations and evolutions. They are carried out in ad-hoc ways, yielding application programming difficulties and inordinate system complexity. Hence, today's middleware solutions still do not provide the level of system-level programmability required to adapt to application domains with differing or additional requirements.

Difficulties encountered when programming Java/CORBA platforms are well recognized, e.g. by the OMG and Sun. The current developments of the Enterprise Java Beans and the CORBA Components frameworks are intended in particular to alleviate the application-level programmability issues by hiding, within so-called component containers with more declarative interfaces, a large part of the complexity involved in dealing with transactional and persistent objects. Then again, these frameworks do not address the needs for adaptation and extension required in several application areas.

In conclusion, there seems to be a need for defining a new architecture for open distributed systems, designed

from the beginning with flexibility and adaptability in mind.

The aim of this paper is to propose a next generation middleware architecture that conforms to the above requirements. This work is the outcome of the CORBAng project, a collaboration between the Universities of Lancaster, Oslo and Tromsø. In the next section we outline the features we regard as essential for the next generation of middleware. Section 3 proposes a open architecture for middleware, while in section 4 we analyse some existing research prototypes with respect to their suitability as next generation middleware. The last section concludes with an outlook to future work.

2. Middleware manifesto

.

Next generation middleware should be run time configurable and allow inspection and adaptation of the underlying software. Our approach to achieve this is based on the marriage of reflection and components.

2.1 Reflective middleware

Reflection allows a program to access, reason about and alter its own interpretation. Access to the interpreter is provided through a meta-object protocol (MOP) which defines the services available at the meta-level. Initially, reflection was applied to the field of programming language design [1]. Later reflection has been applied to operating systems [2] and, more recently, distributed systems [3].

The primary motivation of a reflective language or system is to provide a principled (as opposed to ad hoc) means of achieving open engineering. For example, reflection can be used to *inspect* the internal behaviour of middleware, and to insert additional behaviour to monitor the middleware implementation. Reflection can also be used to *adapt* the internal behaviour of middleware, such as inserting a filter object to reduce the bandwidth requirements of a communication stream.

2.2 Components

A component is a unit for composition and independent deployment [4]. In our view components can be used for structuring of the middleware platform itself. Deployable components thus becomes the foundation for flexibility of next generation middleware. Component libraries supports reflective middleware platforms by providing primitive components from which more complex (composite) services can be built. Example components include a range of low level communication protocols such as IP and IP-multicast, end system components such as filters, dispatchers, buffers, and management components such as scheduling policies and QoS management components. These primitive components can be used to build composite components, including, for example, off-the-shelf bindings offering stream communications for audio and video.

2.3 Meta-space models

We further believe that component oriented programming should be applied both at the base and meta levels. More precisely, every component should have an associated meta-space supporting inspection and adaptation of the underlying infrastructure for the component [5]. In addition, this meta-space should be organised as a number of distinct meta-space models, each one dealing with an independent and orthogonal aspect of the meta-level. The benefit of this approach is to simplify the interface offered by the meta-space through separation of concerns between different system aspects.

According to this, a particular component of the platform can have independent meta-objects, each of them reifying a separate aspect of the component, without interfering with the other aspects. This principle reduces the complexity of the design and implementation of the meta-level for middleware platforms.

We believe the following four meta-models are sufficient to deal with the major aspects that arise in middleware platforms: encapsulation and composition (representing the structural aspects), and environment and resources (representing the behavioural aspects). The encapsulation meta-model deals with the properties of the interfaces of objects, such as their operations or flows. The composition meta-model represents the configurations of composite objects (such as bindings) in terms of object graphs. The environment meta-model reifies the processes involved in the execution of interactions on interfaces. Finally, the resources meta-model represents the resources (threads, memory, etc) used by the components of the platform. It is important to observe that the approach itself is open. This means that new meta-models can be defined, providing a way to represent the meta-level from a different perspective or to adopt a more appropriate granularity for meta-level manipulations.

2.4. Additional properties of reflective middleware

The previous sections have outlined a general architecture for the design of next generation middleware, supporting both structural and behavioural reflection, with the latter including the key element of providing access to resources and resource management. The architecture is based on a multi-model approach in order to provide a

separation of concerns between different systems aspects. We now complete this discussion by highlighting other key properties we seek.

As a group, we believe that next generation middleware should:

- provide seamless access to middleware, operating system and possibly network functionality (c.f. trends in OS design such as Exokernel [6], trends in network development [17], etc);
- promote language independence in that components generally (both within and above the traditional middleware boundary) should not be tied to any specific language, but instead should honour the constraints imposed by the component model;
- accommodate legacy systems such as existing operating systems, protocol services, middleware interfaces (e.g. CORBA), etc;
- address the difficult issues raised by the range of non-functional requirements that cross-cut system development (c.f. Aspect-oriented programming);
- provide a QoS management subsystem fulfilling the central role of sustaining the required non-functional properties (through introspection and adaptation), which uses the same architecture to ensure openness and extensibility;
- involve a minimal impact on performance of base level services, and should adopt an approach where the performance overhead is proportional to the level of reification (e.g. no access should imply little or no overhead).

Finally, and perhaps most importantly, it is crucial to address the issue of integrity of the underlying middleware platform given the level of openness and extensibility inherent in reflective technology. It is absolutely essential that techniques emerge which can constrain the level of change, e.g. by employing solutions emerging from the component community such as component frameworks.

3. Next generation middleware architecture

Tthe CORBAng architecture has been developed in response to the aspects discussed in the previous sections. The CORBAng component model is derived from the computational viewpoint of RM-ODP [7] where components may have multiple interfaces, and explicit bindings can be created between compatible interfaces. The focus of CORBAng is on binding frameworks – a component framework supporting communication.

3.1. Binding framework

The binding framework defines all the entities involved in the creation and manipulation of bindings, as well as the binding components and the way they are handled. Some principles are assumed for the binding framework, such as the support for multi-party and multimedia bindings, QoS negotiation/re-negotiation among the binding endpoints, and the use of policies. Policies are an important part of the framework since they provide a flexible way to control the behaviour of the entities involved in binding actions.

Crucially, the binding framework is based on the concept of *open binding*, as introduced in [8]. An open binding is a composite distributed object, used to connect multiple interfaces across a distributed environment. Open bindings therefore provide all the necessary functionality for supporting remote interaction. In addition, an open binding usually has multiple levels of composition, meaning that one binding can be composed of other, lower level bindings. This allows multiple levels of abstraction and the reuse of binding specifications.

The binding framework defines a number of entities involved in the binding actions. They can be divided as follows: active entities (binding factories and binding mutators), passive entities (binding templates and configurations) and more conceptual entities (binding protocols). Each entity has a particular role as presented below:

- *Binding factory (BF):* the entity responsible for the creation of bindings. The BF is designed as a distributed entity, i.e. as a set of replicated factories. Each replica behaves according to well-defined roles when co-operating to create a binding.
- *Binding mutator (BM):* the entity responsible for co-ordinating binding reconfigurations. Reconfiguration actions include adding, removing or replacing related components at different points of a binding, as well as changing the behaviour of existing components.
- *Binding template (BT):* which specifies the properties, constraints, and configuration guidelines for binding objects. Binding templates are used by BFs to define the structure of new bindings.
- *Configuration:* the set of components that constitute the internals of binding objects. A configuration is the product of the evaluation of a BT, and is represented by an *object graph* depicting the identities and types of the binding components, and their interconnections.
- *Binding protocol (BP):* is the protocol used by BFs to create new bindings. Since there are many issues related to BPs, more details about them will be explored later in this section.

In order to complete the binding framework, a component model as outlined above, is provided. The model prescribe how components are specified, how component specifications are stored for future use, and how components are created from such specifications.

3.2. Binding protocols

Binding protocols guide the communication and co-ordination among the BF replicas involved in a binding creation. By defining distinct binding protocols, we define different ways in which bindings may be created, and allow the creation of bindings with different characteristics (e.g., multi-party, with QoS guarantees). In addition, different binding protocols may be optimised for different binding templates. Therefore, prior to requesting the creation of a binding, the user may set up policies that configure the BFs so that they behave according to the desired binding protocol.

In general terms, all binding protocols have some common features, due to the way open bindings are structured. For instance, an open binding may have multiple levels of nesting, each corresponding to individual bindings. Each binding must be established using binding factories and binding protocols of the appropriate level. This implies that all binding protocols must rely on the execution of other lower level binding protocols.

Another commonality among different binding protocols refers to the way peer BF replicas communicate with each other. When a replica wants to pass information to a peer on the other side of the binding, it passes that information to the next upper level BF replica on its side. At the most external level of binding, the BF replicas transfer the binding information over the network. On the other sides of the binding, at each level, the respective information is delivered to the intended BF replicas. This way, we effectively reduce the communication overhead involved in multi-level binding establishment.

A third similarity among binding protocols is the way QoS negotiation is conducted. In general, the above principle is followed, where the overall QoS for the binding is negotiated among the BF replicas for the uppermost level of binding. The negotiation may take the form of a handshake protocol. This could be a two-way handshake for simple cases of point-to-point binding or a three-way handshake for a more complex negotiation (multi-party binding). Subsequently, the negotiated QoS parameters are propagated to the lower levels of binding (which may also involve some form of QoS translation).

3.3. Adaptation and reconfiguration through reflection

Adaptability is achieved through a comprehensive use of reflection and meta-object protocols, which allows any component of the platform to be reified in terms of meta-objects. Meta-objects can be used to inspect and manipulate the internal structure and behaviour of the reified component. In addition, each component may have four associated meta-objects, each one dealing with one of the meta models (c.f. section 2.3).

For each meta-model, a meta-object protocol is defined, providing a uniform way to inspect and manipulate the meta-space it represents. Each meta-object protocol can be seen as representing a certain “level” of adaptation. For example, the resource meta-model defines operations that can be used to adapt resources allocated to components. In the composition meta-model, operations are defined to get a view of the whole or part of the object graph, as well as to insert, remove or replace individual components. These operations form the building blocks for the complex reconfiguration actions performed by binding mutators. As another example, the encapsulation meta-model has operations to manipulate the functionality provided by individual components of the platform, such as to add (or remove) interactions to the interfaces of components. This can be used to adapt the behaviour of individual primitive components.

Quality of service (QoS) management can be supported as a set of components with given roles such as monitors for monitoring the behaviour of the system, and binding controllers encapsulating (user-defined) policies for reconfiguration and adaptation (using binding mutators to implement policy).

4. Related prototypes

In this section related research prototypes are analyzed with respect to their suitability for next generation middleware. We take a closer look at those prototypes that have been developed in our research groups, i.e., GOPI, Open-ORB, and MULTE-ORB, and two other prototypes we have experience of, i.e. TAO and Flexinet.

4.1. MULTE-ORB

MULTE-ORB is an adaptive multimedia ORB [9,10], that is based on a flexible protocol framework Da CaPo [11]. A flexible protocol system allows dynamic selection, configuration and reconfiguration of protocol modules to dynamically shape the functionality of a protocol to satisfy specific application requirements and/or adapt to changing service properties of the underlying network. .

The features supported by MULTE-ORB include simple QoS negotiation, and support for the procedural approach to configuration and reconfiguration (although it is possible to build declarative mechanisms embedded in policies). In addition, explicit bindings are supported whereby the application may choose fixed protocol graphs

The focus of the work on MULTE-ORB is on the compositional meta-model, and the binding framework, enabling (re-)configuration of the component graphs of remote bindings.

The recursive opening of protocol graphs in MULTE-ORB ends at the socket layer. The rest is "hidden" in the OS kernel or system processes. Another limitation is that the meta-spaces are not explicitly supported, although the compositional meta-space could be implemented as policies using basic mechanisms in MULTE-ORB today.

4.2. Open-ORB

The Open-ORB Python Prototype (OOPP) [12] implements a prototype of the architecture under development in the Open-ORB project [13]. Open-ORB is a reflective middleware based on components. It supports a binding framework providing explicit open bindings.

Reflection in CORBAng and Open-ORB is provided through the same set of distinct meta-space models. However, OOPP currently supports the encapsulation and composition meta-models only.

Quality of service (QoS) management has been introduced in OOPP with a set of objects with given roles, including monitors for monitoring the behaviour of the system, strategy selectors and activators for selecting and activating a strategy to maintain the QoS (if necessary).

In relation to the CORBAng architecture, remote binding objects in OOPP can not be opened beyond the socket layer. Furthermore, the environment and resources (behavioural aspects) meta-spaces are currently not supported,.

4.3. Flexinet

The *ANSA FlexiNet* platform [14] is a *Java* based toolkit for creating and (re)configuring ORB's. It allows programmers to tailor the platform for a particular application domain or deployment scenario. It provides a generic binding framework plus a set of basic engineering components to populate the framework. *FlexiNet* is focused at operational interaction (RPC).

The layers of the *FlexiNet* communication stack can be viewed as reflective objects that manipulate generic invocations in different ways before it is invoked on the destination object.

All configuration is done at binding time by binder components. There is no open interfaces (MOP) that gives access to the stacks as meta-level objects. Hence there is no explicit support for adaptation. Interfaces to the binder and generic call interface between layers are open. There is no implementation independent way to instantiate and configure binder and layer components, but this could be added.

In the *FlexiBind* architecture [15] we extend *FlexiNet* to support dynamic binding configuration by pluggable and replaceable *policies*. Bindings may be activated, passivated or reactivated (using different policies). Hence *FlexiBind* supports adaptable bindings without supporting a full MOP for the protocol stack configuration.

4.4 GOPI

GOPI (Generic Object Platform Infrastructure) [16] is a distributed object-based middleware platform that supports multimedia applications using stream interfaces, explicit bindings, third party-binding and QoS (quality of service) support.

Architecturally, GOPI is divided in the GOPI core and the API Personality (multiple personalities may be supported simultaneously). GOPI core is composed of modules that handle the communication and resource management, and personalities make the core functionality available to application programmers through some standard interface (e.g. a standard CORBA personality and an RM-ODP inspired personality are currently available).

The GOPI core *comm* module provides a framework for creating flexible protocol stacks; each application specific protocol (ASP) uses and provides a standard interface which includes QoS management.

The main objective of GOPI is to support multimedia applications by providing QoS management, flexible protocol composition and stream interfaces. Although some level of flexibility and adaptation is provided, this is not achieved by means of reflection and meta-object protocols as in CORBAng. Another feature of CORBAng that is not supported by GOPI is the component model. In GOPI, ASPs could be replaced by components with relative ease, but this is not necessarily true of the whole core.

4.5 TAO ORB

TAO [18] is a freely available and open source CORBA implementation with special efforts on high performance and predictable real-time QoS. TAO is built atop ACE [19], an object-oriented framework for concurrent communication software. ACE allows the deployment of multiple concurrency and memory management policies to meet specific application needs.

ACE supports the equivalent of a compositional meta model that automates most of system configuration and reconfiguration by dynamically linking services into applications at run-time. Although TAO is implemented on top of ACE, TAO does not support dynamic runtime reconfiguration and adaptation. The pluggable protocol mechanism in TAO allows the application to choose appropriate transport protocols at "application initiation time".

5. Conclusions and future work

This paper has proposed a next generation middleware architecture in response to requirements from existing and new types of applications, including support for multimedia, real-time requirements and mobility. This architecture is characterised by being open and adaptable based on the marriage of the two technologies of reflection and components. This provides a principled approach to open engineering as opposed to more ad hoc approaches taken, for example, by OMG to achieve similar goals for CORBA.

The paper also reported on some existing research prototypes with a focus towards their suitability as next generation middleware. None of the prototypes investigated fully conforms to our proposed architecture, although some are closer than other.

In our future work we plan to realise a middleware prototype that fully conforms to the desired properties of reflective middleware. In particular this new prototype will be reflective at "all levels" providing reflective access to middleware, operating system and possibly network functionality.

6. Acknowledgements

The CORBAng project is sponsored by the Norwegian Research Council, and British Council. Anders Andersen is sponsered by NORUT-IT, Norway. Fábio Costa is sponsored by CNPq, and UFG, Brazil. Katia Barbosa Saikoski is sponsored by CAPES, and PUCRS, Brazil

7. References

[1] G. Kiczales, J. des Rivieres, D. G. Borrow, "The Art of the Metaobject Protocol", *The MIT Press*, 1991.

[2] Y. Yokote, "The Apertos reflective operating system: The concept and its implementation", in *Proceedings of OOPSLA'92*, ACM Press, 1992, pp. 414-434

[3] J. McAffer, "Meta-level architecture support for distributed objects", in *Proceedings of Reflection'96*, (G. Kinzales ed.), San Francisco, 1996, pp. 39-62.

[4] C. Szyperski. Component Software, Beyond Object-Oriented Programming. *ACM Press/Addison-Wesley* 1998.

[5] G.S. Blair, F. Costa, G. Coulson, F. Delpiano, H. Duran, B. Dumant, F. Horn, N. Parlavantzas, and J-B. Stefani. "The Design of a Resource-Aware Reflective Middleware Architecture." In *Second International Conference on Reflection and Meta-level architectures (Reflection'99)*, St. Malo, France, July 1999.

[6] R. Engler Dawson, M. Frans Kaashoek, "Exterminate All Operating System Abstractions", in *Proceedings of 5th Workshop on Hot Topics in Operating Systems*, IEEE Society, 1995

[7] ISO/IEC, "Open distributed processing reference model, part 3: Architecture", *ITU-T rec. X.903 – ISO/IEC 10746-3*, ISO/IEC, 1995.

[8] T. Fitzpatrick, G.S. Blair, G. Coulson, N. Davies, and P. Robin. "Supporting adaptive multimedia applications through open bindings." In *Proceedings of International Conference on Configurable Distributed Systems (ICCDS '98)*, Annapolis, Maryland, USA, May 1998.

[9] Kristensen, T., Plagemann, T, "Extending the Object Request Broker COOL with Flexible QoS Support", Technical Report UniK – Center for Technology, University of Oslo, January 1999

[10] T. Plagemann, T., F. Eliassen, V. Goebel, T. Kristensen, H. O. Rafaelsen, "Adaptive QoS Aware Binding of Persistent Objects", in *Proceedings of International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, IEEE, September 1999

[11] T. Plagemann, A Framework for Dynamic Protocol Configuration", *Dissertation at Swiss Federal Institute of Technology*, Computer Engineering and Networks Laboratory, Zurich, Switzerland, Sept. 1994

[12] A. Andersen, G. S. Blair, G. Coulson, F. Eliassen, "A reflective component-based middleware in Python", *NORUT IT*, September 1999 (submitted to IPC8).

[13] G. S. Blair, G. Coulson, P. Robin, M. Papathomas, "An Architecture for Next Generation Middleware". *In Proceedings of Middleware'98*, Springer, September 1998.

[14] R. Hayton et. al., "FlexiNet Architecture Report", *ANSA Phase III report*, February 1999

[15] Ø. Hanssen, F. Eliassen, "A Framework for Policy Bindings", in *Proceedings of International Symposium on Distributed Objects and Applications (DOA'99)*, Edinburgh, Scotland, IEEE, September 1999.

[16] G. Coulson, "A Configurable Multimedia Middleware Platform", *IEEE Multimedia*, Vol. 6, No. 1, January-March 1999.

[17] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. Vicente, D. A. Villela, "Understanding Programmable Networks", *Computer Communication Review*, 29(2) April 1999.

[18] D. Schmidt, The ADAPTIVE Communication Environment: Object-Oriented Network Programming Components for Developing Client/Server Applications, 11th and 12th Sun Users Group Conference, December 1993 and June 1994.

[19] D. Schmidt, D., D. Levine, C. Cleeland, "Architectures and Patterns for High-performance, Real-time CORBA Object Request Brokers", *Advances in Computers*, Academic Press, Ed., Marvin Zelkowitz, to appear