Arctic Beans: Flexible and Open Enterprise Component Architectures

Anders Andersen, Gordon Blair[*], Vera Goebel[†],
Randi Karlsen, Tage Stabell-Kulø, and Weihai Yu
Department of Computer Science
University of Tromsø, Norway
{aa,gordon,vera,randi,tage,weihai}@cs.uit.no

Abstract

The Arctic Beans project is investigating the area of *"Configurable and Reconfigurable Enterprise Component Architectures"*. The general aim of this research is to develop more open and flexible middleware technologies, focusing on enterprise (or server-side) component architecture (such as Enterprise JavaBeans or the CORBA Component Model). The great benefit of such technologies is that they encourage re-use of components developed by third-party suppliers, and also manage distribution implicitly through the concept of a container. This works well for a significant number of application domains. However, there is increasing evidence that there are problems with this approach as the technology is applied in other areas (e.g. in mobile computing). The reason for this is that enterprise component architectures tend to enforce a certain style of distribution management in terms of, for example, the approach to persistence, security, and transactions, or the general model of interaction implied by the approach. We seek a more flexible technology that allows such elements to be configured at installation time, and re-configured at run-time.

1 Introduction

Middleware has emerged as an important architectural component in modern distributed systems. The role of middleware is to offer a high-level, platform-independent programming model to users, and to mask out problems of distribution. Examples of key middleware platforms include CORBA, DCOM, and the Java-based series of technologies (RMI, JINI, etc). Traditionally, such platforms have been deployed (with considerable success) in application domains such as banking and finance as a means of tackling problems of heterogeneity, and also supporting the integration of legacy systems. However, more recently, middleware technologies have been applied in a wider range of areas including safety critical or real-time systems, embedded systems, telecommunications, mobile and ubiquitous systems, the computational GRID, etc. Unfortunately, as this diversification proceeds, it is becoming ever more apparent that the middleware technologies mentioned above are not able to support such a diversity of application domains. The main reason for this is the black box philosophy adopted by existing platforms. In particular, existing

[*] Also at Distributed Multimedia Research Group, Lancaster University, UK.
[†] Also at Department of Informatics, University of Oslo, Norway.

middleware platforms offer a fixed service to their users, and it is not possible to view or alter the implementation of this service, i.e. they are closed systems. Inevitably, the platform architecture represents a compromise featuring, e.g. general- purpose protocols and associated management strategies. It is then not possible to configure platforms to meet the needs of more specific domains. Similarly, it is not possible to re-configure platforms at run-time as, for example, the underlying environmental conditions fluctuate. Equally, it is difficult to evolve such architectures in the longer-term to meet new application requirements.

This problem is particularly evident in emerging enterprise (or server-side) component architectures (such as Enterprise JavaBeans or the CORBA Component Model). Such architectures provide implicit support for distribution through the concept of a container. The main problem with this approach is that distribution management tends to be hidden from the developer, apart from through a series of configuration parameters. There is increasing evidence that this is not sufficient for modern distributed applications. The Arctic Beans project is tackling this problem, with the overall aim of developing a more open and flexible enterprise component architecture with intrinsic support for configuration, re-configuration and evolution. We expand on this analysis below.

2 Related work

2.1 Flexibility and middleware

As mentioned above, existing middleware is generally rather inflexible. Middleware designers are aware of this problem and have responded accordingly. For example, the OMG have introduced specifications including *Real-time CORBA* and *Minimal CORBA* [1]. These are however specific solutions for specific domains and are by no means generic. *Portable Interceptors* [2] enable the customisation of CORBA by allowing the interception of invocations/replies via pre- or post-processing, and also the interception of IOR creation. This is a useful but limited mechanism: it is not possible to add interceptors at arbitrary points in the ORB implementation, and there is no native support for structured composition or for the dynamic installation of interceptors. The other notable customisation feature is the use of *policy objects* [1], which allow control over particular internal services of the ORB, e.g. the POA, asynchronous messaging and security. However, once policies are installed, the set of potential policies are fixed, thus not fully supporting adaptation. Other standards and platforms offer similar degrees of flexibility, e.g. *custom marshalling* in COM [3], *interception* in COM+ [4] and *dynamic proxies* in Java [5]. However, they all suffer from similar problems as above. In general, such mechanisms can be viewed as ad-hoc and incomplete and insufficient to meet the considerable demands of next generation, distributed applications. Consequently, a number of researchers are experimenting with the more principled approach of reflective middleware. We examine this emerging area below.

2.2 Reflective middleware

Reflection has been deployed successfully in areas such as programming languages and operating systems, and is now increasingly being applied to middleware. The key to the approach is to offer a meta-interface, or meta-object protocol (MOP), supporting access to the underlying virtual machine. This MOP provides operations to inspect the internal

details of a platform (introspection). By exposing the underlying implementation, it is also possible to insert behaviour to monitor the implementation, e.g. quality of service monitors [6]. The MOP also typically provides operations to alter the underlying middleware (adaptation). Examples include changing the implementation of the underlying transport protocol to operate over a wireless link or inserting a filter to reduce the bandwidth requirements of a media stream.

More generally, middleware platforms typically offer two (complementary) styles of reflection, structural and behavioural reflection. *Structural reflection* is concerned with the underlying structure of objects, e.g. in terms of the set of interfaces supported (c.f. introspection Java [5]). More advanced options include support for adaptation of the behaviour of objects or architectural reflection [7]. In the latter approach, the MOP provides access to the architecture of the system, e.g. in terms of components and connectors. *Behavioural reflection* is concerned with activity in the underlying system, e.g. in terms of the arrival of invocations. Typical mechanisms provided include the use of interceptors (as found in CORBA) or dynamic proxies in Java [5]. Some research has also been carried out on providing access to underlying resources and associated resource management [8].

2.3 Component technologies

Recently, a range of component-based middleware technologies have emerged. According to Szyperski [9], a component is defined as "a unit of composition with contractually specified interfaces and explicit context dependencies only". In addition, he states that a component "can be deployed independently and is subject to third-party composition". A key part of this definition is the emphasis on composition; component technologies rely heavily on composition rather than inheritance for the construction of applications, thus avoiding the fragile base class problem (and the subsequent difficulties in terms of system evolution). To support third party composition, they also introduce more explicit contracts in terms of both provided and required interfaces. The overall aim is to reduce time to market for new services through the emphasis on programming by assembly rather than software development (c.f. manufacturing vs engineering).

In terms of middleware, most emphasis has been given to enterprise (or server-side) component technologies, including Enterprise Java Beans (EJB) and the CORBA Component Model (CCM). In such enterprise technologies, components execute within a container, which provides implicit support for distribution, in terms of support for transactions, security, persistence, location transparency and resource management. This offers an important separation of concerns in the development of business applications, i.e. the application programmer can focus on the development and potential re-use of components to provide the necessary business logic, and a more "distribution-aware" developer can then provide a container with the necessary non-functional properties. As well as support for distribution, containers also provide additional functionality including life-cycle management and component discovery.

Crucially, available enterprise component models and containers (servers) only provide limited and pre-selected set of ways to configure and customise the behaviour of the components and their environment (i.e. their containers). In addition, such customisation is done at deploy time (when the component is installed in its container). This makes the enterprise component model too static and not feasible for a wide range of applications with more flexible needs. For instance, most containers provide access control based on access control lists (ACL) with authenticated users where users are authenticated with

passwords. This model might not fit specific applications based on a different security model or a different security technology (e.g. authentication based on public and private key pairs).

Based on this analysis, we contend that enterprise component technologies suffer from precisely the same problems as encountered in the previous generation of middleware technologies, and require more sophisticated support for configurability, re-configurability and evolution. We further contend that they can benefit greatly from the application of reflection to their architectures. This is precisely the aim of the Arctic Beans project, i.e. to develop a more open and flexible enterprise component architecture through the application of reflection.

3 The Arctic Beans Project

3.1 Overall Aims

The primary aim of Arctic Beans is to provide a more *open and flexible* enterprise component technology, with intrinsic support for configurability, re-configurability and evolution. The Arctic Beans project focuses on the security and transaction services and how to adapt these services according to the application needs and its environment. The main application domain is that of mobile computing as this presents a number of major challenges for enterprise architectures. However, other non-functional properties and application domains are addressed to demonstrate the generality of the approach.

More specifically, Arctic Beans has the following objectives: (i) to design, implement and evaluate an *enterprise component* technology based on the concept of containers but with the capability to configure a container at installation time, and to monitor and reconfigure this container at run-time; (ii) to develop an associated *meta-architecture* for containers, featuring support for structural and behavioural reflection, in order to support the necessary level of openness as discussed above; (iii) to develop flexible models of *security and transactions* within the overall architecture with the goal of supporting a range of applications from large scale business applications to mobile applications; and (iv) to develop and extensively publicise a well-defined and complete *meta-object protocol* (MOP) for enterprise architectures with a view to influencing the next generation of standards and platforms.

3.2 Architecture

In the current phase, an initial version of the architecture is being defined and a corresponding prototype implemented. This architecture adopts a fairly classical approach to composing non-functional properties, i.e. the use of a range of *interceptors* [2] (c.f. composition filters [10]). This approach is not entirely novel, as a number of researchers have experimented with the use of interceptors for the implementation of non-functional concerns [11, 12], but this work will provide us with strong experience of using such architectures. The work requires the definition of a MOP providing access to this filter based structure. Currently, the reflective component technology developed at Lancaster [13] is being extended to experiment with this classical approach.

In the next phase the architecture will be extended in a number of key ways. Firstly, a more comprehensive MOP will be defined featuring both structural and behavioural reflection (incorporating a level of control over resource usage and management). Secondly,

a more sophisticated approach will be developed for the container architecture. In particular, we will also focus on the key issue of *interactions* between non-functional services. During adaptation, it is necessary to understand the possible interactions between the existing and new services. Some interactions may be unwanted and introduce errors into the system, e.g. caching interfering with the desire for privacy; others may be beneficial, e.g. being able to tentatively start a transaction while authentication is being carried out. To accommodate this, we will introduce an integration manager into our architecture to create a clear separation of concerns between non- functional services and their coordination (this separation does not exist with interception). This manager will have the task of coordinating the execution of non-functional services, preventing unwanted interactions and exploiting beneficial interactions. Investigations will also be carried out into the related concept of execution monitors recently developed at École des Mines de Nantes [14], and also of the general areas of coordination and software architecture [15] (w.r.t. input on separation of concerns) and feature interaction [16] (w.r.t. input on studies of interaction problems).

3.3 Security

When security is designed into a distributed system, public-key encryption is used more often than not. In the context of flexible distributed systems, three problems are paramount. Firstly, the use of public-key encryption makes digital signatures available, and thus ensures that access to the system (i.e. login) can be done in an orderly fashion. In particular, if the user has asserted that he will protect the secret part of the key (pair), digital signatures on login certificates can be used as evidence [17]. However, if security is built on the use of public keys, it is hard, if not impossible, to "downgrade" (parts of) the system to use other security technologies such as passwords for authentication. Secondly, it is prudent to design the system in such a manner that effective revocation is possible. The only reasonable way to implement revocation is by assuming (or creating) an on-line infrastructure where keys and certificates can be verified in real time (often called a public key infrastructure or PKI). When the system requires access to a PKI, off-line use of (parts of) the system becomes impossible. That is, even when the user has provided the necessary credentials, he might find himself being denied access. Thirdly, because a public key cannot be memorised, being identified by a public key implies that either a smartcard holding the key, or some other trusted computer, is at hand in order for the user to access the system. In particular, access to the system in an ad-hoc manner becomes impossible. These three issues clearly demonstrate that a distributed infrastructure cannot be built to rely on public-key encryption alone. Instead, the security infrastructure must be designed around a language, in which system owners, and users alike, can describe their demands on authentication, on authorisation, delegation, and so on [18, 19]. Currently the most suitable candidate for such a language seems to be SPKI [20].

In the Arctic Beans project we try to implement an infrastructure where security can be programmed, rather than being fixed, with applications left to adhere to the design chosen.

3.4 Transactions

The Arctic Beans project includes flexible distributed transaction processing mechanisms and integrate these in the core architecture. In our ongoing work these mechanisms will be

based on the principle of reflection and therefore a clearly defined meta-transaction model. More specifically, distributed transaction processing consists of three types of activities, namely concurrency control, recovery and overall transaction management. The primary focus of this work is transaction management in the distributed setting. The mechanisms are integrated in the core architecture and interact with other non-functional aspects such as security.

Overall transaction management, is concerned with controlling transaction executions over a number of participating servers so that overall transaction properties are guaranteed. Transaction managers often use the two-phase commit protocol to guarantee atomicity. This protocol represents a fixed service with an explicit policy that makes it a blocking protocol where every part of the transaction must be successfully executed in order to commit. Three-phase commit is an alternative protocol, that also represents a fixed service but with a slightly different policy. The policy of three-phase commit determines it to be a non-blocking protocol that also has the all or nothing execution property. Transaction managers are choosing one protocol with an explicit policy, and there are no possibilities for configurating or re-configurating the policy according to application or environmental needs. So, dynamically choosing between for instance a blocking and a non-blocking approach or dynamically introducing new execution properties are not possible.

The introduction of new application areas have caused the original ACID transaction properties to be questioned. Several research groups have provided mechanisms that relax these properties, and introduce for instance a semantic atomicity property. Much of this work is summarised in [21]. However, these new mechanisms also have the shortcoming of representing a fixed service with an explicit, but possibly more flexible policy. Even though the mechanisms may be more flexible and allow different properties, they do not allow configuration and re-configuration of the overall policy.

We are currently investigating how to implement an infrastructure where transaction processing can be programmed, rather then being fixed. We try to define a meta-transaction model with generic primitives for different transactional behaviour, and with support for configurable and re-configurable transaction execution policy. The policy together with the generic properties determine the mechanism controlling the transaction execution. The meta-transaction model is initially based on available research results, such as [22] and [23], that have been used to specify, reason about and implement different extended transaction models.

4 Conclusion

The Arctic Beans project aims at developing open and flexible enterprise component architecture supporting configuration, re-configuration and evolution. The goal is to provide an architecture that addresses the problems and limitations of closed middleware platforms offering fixed services to the users. The project focuses of three important aspects; to develop a meta-architecture for containers, and to develop flexible models for security and transactions. The initial work done on the Arctic Beans architecture is promising regarding flexible security and transaction mechanisms.

References

[1] Object Managment Group. The common object request broker: Architecture and

specification. Technical report, Object Managment Group, February 2001. (revision 2.4.2).

[2] Object Managment Group. Portable interceptors. TC Document orbos/99-12-02, Joint Revised Submission, 1999.

[3] Microsoft Corporation and Digital Equipment Corporation. The component object model specification. Technical report, Microsoft Corporation, oct 1995. (version 0.9).

[4] Mary Kirtland. Object-oriented software development made simple with COM+ runtime services. *Microsoft System Journal*, November 1997.

[5] Sun Microsystems. Java core reflection. Technical report, Sun Microsystems, Inc, 1998.

[6] Anders Andersen, Gordon S. Blair, and Frank Eliassen. A reflective component-based middleware with quality of service management. In *PROMS 2000, Protocols for Multimedia Systems*, Cracow, Poland, October 2000.

[7] W. Cazzola, W. Savigni, A. Sosio, and F. Tisato. Rule-based strategic reflection: Observing and modifying behaviour at the architectural level. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE'99)*, Cocoa Beach, Florida, USA, October 1999.

[8] Hector Duran-Limon and Gordon S. Blair. Specifying real-time behaviour in distributed software architectures. In *Proceedings of the 3rd Australasian Workshop on Software and System Architectures*, Sydney, Australia, November 2000.

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

[10] L. M. Bergamans and L. M. Askit. Composing multiple concerns using composition filters. *Communications of the ACM*, 43(10), October 2001. (accepted).

[11] E. Bruneton and M. Riveill. Reflective implementation on non-functional properties with the JavaPod component platform. In *Proceedings of the Workshop on Reflection and Meta-level Architectures (RMA'2000)*, Cannes, France, 2000.

[12] N. Diakov, H. Batteram, H. Zandbelt, and M. van Sinderen. Design and implementation of a framework for monitoring distributed component interactions. In *Proceedings of the 7th International Conference on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS 2000)*, volume 1905 of *Lecture Notes in Computer Science*, pages 227–240, Enschede, The Netherlands, October 2000. Springer-Verlag.

[13] R. M. Moreira, G. S. Blair, and E. Carrapatoso. A reflective component-based and architecture aware framework to manage architectural composition. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA'01)*, Rome, Italiy, September 2001.

[14] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of Reflection'01*, Kyoto, Japan, September 2001. (accepted).

[15] D. Garlan and M. Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Co., New Jersey, 1993.

[16] L. Blair, G. S. Blair, J. Pang, and C. Efstratiou. Feature interaction outside a telecom domain. In *Proceedings of the Workshop on Feature Interactions in Composed Systems, ECOOP'2001*, Budapest, June 2001.

[17] Michael Roe. *Cryptography and Evidence*. PhD thesis, Clare College, University of Cambridge, 1997.

[18] B. Lampson. Protection. In *Proceedings of the Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, 1971. Reprinted in Operating Systems Review, 8, 1, January 1974.

[19] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transaction of Computer Systems*, 10(4), November 1992.

[20] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, The Internet Society, 1999.

[21] K. Ramamritham and P. K. Chrysanthis. *Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Press, 1997.

[22] P. K. Chrysanthis. ACTA: A framework for specifying and reasoning about transaction structure and behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, 1990.

[23] R. Barga. *Reflective Framework for Implementing Extended Transactions*. PhD thesis, Oregon Graduate Institute, Portland, Oregon, 1998.