Evolution in Distributed Heterogeneous Systems

Premkumar Devanbu and Eric Wohlstadter,
Dept. Of Computer Science,
University of California, Davis, CA, 95616
devanbu@cs.ucdavis.edu

October 31, 2001

Abstract

Distributed, heterogeneous systems are becoming very common, as globalized organizations integrate applications running on different platforms, possibly written in different languages. Component-interoperability standards such as CORBA are critical enablers of this trend. Certain non-functional requirements for such features as security, quality of service, flexible administration are specially critical to distributed heterogeneous systems. Unfortunately, such requirements are often formulated late, since they depend upon a particular installation, and/or change rapidly with business and political climate. Distributed, heterogeneous systems are particularly difficult to evolve, since the elements are written in different languages, and the operational environment is heterogenous and distributed. Adding "non-functional" features late in the game is specially hard; the required modifications are scattered through the implementations of the different components. Their design and implementation is also obscured by code delocalization, as well as by complexities arising from co-ordination and synchronization considerations. We would like to address this problem with solutions that are animated by practical software engineering goals: *type safety* of scattered changes, and their interactions; *explicit design models*, with tracability to code; *inter-operability* with legacy components and binary COTS components; and *opportunistic* optimization, leveraging any available optimizations existing in the compilers that are in use.

1 Background

Many software applications are both distributed and heterogeneous. *Distribution* arises from business and human imperatives (such as globalization) and has been facilitated by the rise of reliable, high-speed networks, and by standards-based middleware such as CORBA [11]. Modern systems are also heterogeneous: they comprise sub-systems built on many different platforms. *Heterogeneity* arises for a variety of reasons: physical constraints (weight, cost, battery capacity, size), market conditions (e.g., certain components available only on certain platforms) available technical skills (Java, C++, Perl, etc), and history (legacy systems). Distributed heterogeneous ($\mathcal{DH}$) systems are now very common, and are likely to endure.

Like all software systems, $\mathcal{DH}$ systems often come under pressure to evolve during their life-times, as requirements change. Unfortunately, $\mathcal{DH}$ systems are inherently hard to evolve, for a variety of reasons: they employ different languages and hardware/software platforms; installation procedures may be quite complex; synchronization, concurrency, and failures present difficult programming challenges; and source code may not be available.

Of special interest here are *non-functional* requirements, such as security (including access control), quality-of-service, monitoring (for intrusion detection), and administrative features. Such requirements can often only be finalized quite late, perhaps even after deployment; they give rise to *late evolutionary pressure* on software systems. Unfortunately, these requirements changes can rarely be addressed by isolated changes to a few closely related modules. Experience amply indicates [1, 3, 4] that the implementation of features such as security are scattered through a variety of different modules. Features that require this type of de-localized implementation are called “cross-cutting concerns” in the literature. This phenomenon makes non-functional features specially difficult to add or evolve, even in non-distributed systems implemented in a single language. The problem become much worse for $\mathcal{DH}$ systems.

In this research, we propose to support the evolution of $\mathcal{DH}$ systems by enhancing middleware. Our research is animated by several software engineering principles, such as: type safety; the need to evolve COTS and legacy components available only in binary form; inter-operability of changes and components written in different languages; and the opportunity to leverage any optimizations available in the compilers. Our scope encompasses changes to interface definition languages to support architecture-level modeling of cross-cutting concerns, enhancements to the IDL compilers to provide additional type-checking for the implementations of cross-cutting concerns, changes to run-time environment, and finally, changes to configuration management to support static and dynamic integration of cross-cutting implementation.

2 An Example

Consider a $\mathcal{DH}$ medical application (figure 1), with a set of clients making use of three groups of servers (shown as groups of circles): clinics, pharmacies, and insurers. The servers in a group could be running on different platforms (doctor’s offices might use different types of computers), but each provides the same function (e.g., same CORBA IDL interface). In Figure 2 (A), the original service is shown. The components in this architecture communicate using $\mathcal{DH}$ middleware. We now desire to add security into this system, implementing a policy that has two critical elements

1. Each client must be authenticated by an authentication server (perhaps by a password scheme, or a public-key scheme; the details are not critical).
2. Each client must deal *with exactly one* server from each category. Thus, each client must deal with just one doctor, one pharmacy, and one insurer. This could be useful to discourage some types of fraud, drug abuse, etc.

Since our central goal is to facilitate this type of modification, we consider in detail the implementation changes required. Figure 2 (B) schematically indicates the high-level architectural modifications that might be required. A new authentication server has been added, to validate users. Each group of servers also has a security “wall” surrounding it, which provides the access control. At a a lower level, this modification requires changes to every component, and also to the interactions between components. The client now has to authenticate itself to the authentication server, which presumably provides some of identity token. This token must now added to all client-service requests. All members of each group of services must now co-ordinate among themselves to make

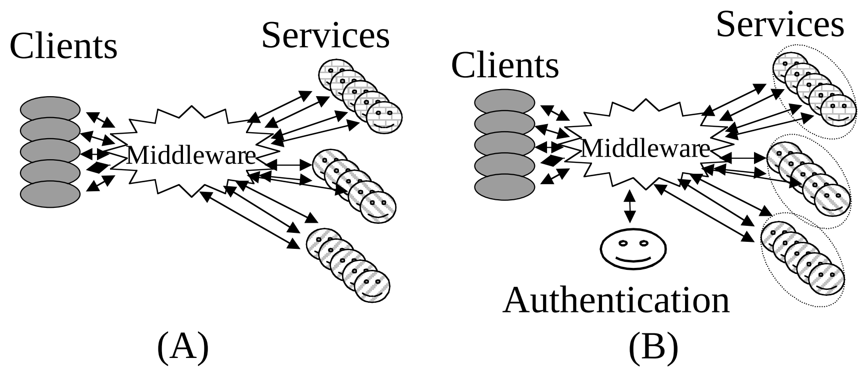

Figure 1: The figure (A) on the left represents a $\mathcal{DH}$ system comprising several clients using services shown in 3 groups. Figure (B), is the same system, with security added. Clients use an authentication server to identify themselves to servers, who now have access control figuratively surrounding them. We consider a security policy where clients can access only one server in each group. This might prevent, for example, a patient getting multiple prescriptions of a controlled drug from several different doctors. The implementation of such a security feature transcends the existing component boundaries of the system on the left.

sure that a client with a particular identity does not interact with more than one specific member of a group. Since malicious clients may try to induce race conditions by simultaneously contacting several members of a group, group members need to synchronize with each other before they "commit" to serving a client.

Making this type of change to the different elements presents several challenges. The changes are clearly "cross-cutting", and scattered; understanding these distributed changes as a whole is a conceptual challenge. Programs may be based on different platforms, possibly written in different languages. In some cases, source code for some elements may not be available. Thus, it may not be feasible to change the interface of existing components. Changes may have to be "loosely coupled" in the form of wrappers or proxies. Changes might have to be made by different organizations. However, the changes must be made in a consistent way, to ensure correct interaction. Changes must be all deployed in the different elements. For comprehensibility, and to support further evolution, the impact of the changes should be clearly modeled and traceable all the way from architecture to implementation. Finally, changes should be composable, and there should be clear support for explicitly dealing with feature interactions.

Our goal address these challenges, and support the evolution of $\mathcal{DH}$ systems. Specifically, we propose to advance the state of middleware (including IDLs, IDL compilers, and run-time environments). Our designs are animated by the following goals:

Safety Modifications to different components should be co-ordinated; in particular, information exchange between modifications should provide all the type safety guarantees available to components using the middleware (e.g., with CORBA IDL).

Inter-operability Some component modifications can be made in a language different from the component implementation language; some modifications can also be applied to COTS components available only in binary form.

Opportunism However, if source code is available, modifications should be amenable to optimizations supported by the available compilers.

Modeling & Traceability There should be a high-level architectural model (e.g., CORBA IDL or MCA or equivalent) of the scattered, cross-cutting implementation changes; this high-level design model should be traceable to the code modification.

Composability Modifications should be composable, and there should be support for composition in the infra-structure. There should be explicit ways of dealing with feature interactions [12].

We have developed practical designs that address some of these goals. We hope to develop these designs further, implement them, and evaluate them.

3 Related Work

The challenges of evolving large systems to adapting to changing non-functional requirements has been recognized by many researchers, and various approaches have been proposed.

Language-specific mechanisms have been in the vanguard. Aspect-oriented programming, or AOP, (see a recent issue of *CACM* [1]) introduces language design principles that have produced enhancements to Java [6] and also to C [4]. AOP supports evolution via *cross-cuts*, which are sets of events (method calls, exception raises, etc) that are to be intercepted, and *advice* that is to be executed when these events are activated. The insertion of advice is accomplished through static code transformation (evocatively called "weaving"). Cross-cuts and advice are integrated into a static scoping device called an "aspect" that allows AOP programmers to conceptualize and integrate otherwise scattered changes to a system. Both the advice and the cross-cuts are language-specific mechanisms. A different mechanism, with strong roots in the concept of monads [15] from abstract algebra has been used as a way of dealing with software evolution in lazy functional languages. Monads basically provide a way for programmers to "over-ride" fundamental mechanisms of expression evaluation and value propagation, and thus change the behavior of a program written in monadic style; changes are encapsulated into the context of the operations on a monad. While monadic evolution doesn't allow nearly as much flexibility in selecting groups of program execution events as parts of cross-cuts, the mathematical foundations of monads allow for more compositionality (through combinators and explicit "lifting" [12] to handle feature-interaction), generality (though higher-order programming), and static checking (through built-in, powerful higher-order type systems). Our goal, however, is to seek *language-independent* mechanisms that allow adaptations in $\mathcal{DH}$ systems built out of components in different languages, and/or COTS components only available in binary form. Moreover, non-functional adaptations in $\mathcal{DH}$ systems are likely to cross process boundaries; we need mechanisms to support the propagation of typing environments across the boundaries to allow static typing mechanisms different compilers to work together to ensure that the adaptations, taken together, are type-safe.

Research in Component-based technologies, and industry, have also produced adaptation mechanisms. Wrappers are established means of adapting legacy systems for inter-operability [17], and also of adding security features to individual components [5, 14]. While wrappers work on the implementation side, smart proxies [16] work on the client side. While individual components can be adapted by wrappers or smart proxies, these mechanisms do not easily support evolutionary changes to a $\mathcal{DH}$ that involve co-ordinated modifications to several separate components. QUO [9] allows co-ordinated modifications to all elements of a $\mathcal{DH}$. QUO is mainly intended to address quality of service issues, and is based on a domain-specific language. Other mechanisms involve modifying the middleware infra-structure. Approaches include interceptors [10, 16], filters [13], and specially constructed ORBs [7]. Some use language-specific reflection mechanisms [8]. With these, one can add instrumentation to *all* invocations arriving at an ORB to perform such generic actions as logging. They also use reflective techniques that treat invocations as objects and analyze them. We seek mechanisms that can be more closely tailored to specific components; in addition, we would like to allow mechanisms that allow static type checking, using existing compilers. Thus, we would prefer to avoid reflection to communicate between adaptations. While reflection is sometimes very useful, we believe that the evolution of $\mathcal{DH}$ systems often demands a more rigorous static typing infra-structure to ensure agreement between the different adaptations (which may not always be written by the same people). Component containers are another approach to evolvable systems, but so far these approaches, just those with Enterprise Java Beans (EJB) and Aspect EJB [2] are also language-specific, and rely heavily upon reflection.

In conclusion, we believe that the goals of our research programme as outlined at the end of the previous section, and illustrated in the example, are novel, and worthy of pursuit. We briefly describe in the section our approach to this problem.

4 Approach

One manifestation of our approach is shown in figure 2. We show an existing pair of darkly shaded components, which used to communicate over an interface that was devoid of security-related parameters. These communicate

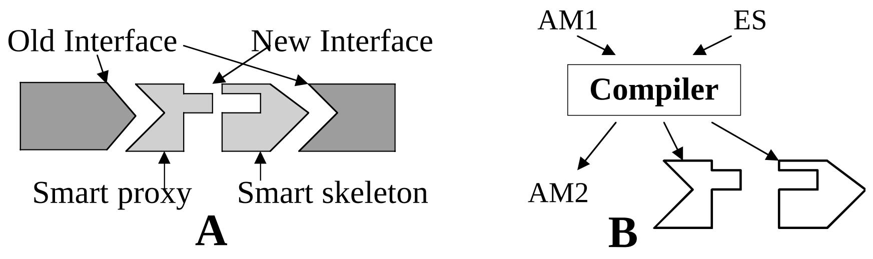

Figure 2: On the left (A) we show a realization of an adaptation mechanism. The two darker elements communicate over an interface that doesn't include security information. An adaptation introduces a lightly shaded smart proxy and a smart skeleton, which communicate over a new interface, which forms the common typing environment. These elements communicate with the existing, darkly shaded elements. On the right (B), we show the tool infra structure. The old architectural model (AM1) is annotated with an evolution specification (ES). A compiler processes these two together to produce a new architectural specification, as well as the boiler plate for the smart skeleton and proxy, which are implemented by programmers, possibly in different languages.

now through a newly introduced pair of lightly shaded components, whose communication interface is enhanced with security parameters. These are referred to as "smart proxy" and "smart skeleton", by analogy with the CORBA OMG architecture. The "fitted" shapes of the interface are meant to be evocative of type safety, unlike, for example, approaches based on reflection. On the right hand side, we show the compilation infra-structure.

We assume that there is an existing architectural model (AM1) (e.g., IDLs, OMG's architectural models, or another ADL specification). Our approach calls for an evolution specification (ES) to be written, which describes which components are to be modified, and/or which architectural interactions are to be intercepted. The design of the evolution specification language will draw from both aspect-oriented programming languages [4, 6], as well as the "lifters" of Prehofer [12], which are based on monad composition. This ES, along with the existing AM1 is processed by a compiler to produce a new architectural specification AM2, along with boiler plates for the smart proxies and smart skeletons, which must be implemented by the programmers; the boiler-plate provides the typing environment. If there are "lifters" specified in the evolution specification, the programmer is given boiler plates for the lifters which must be implemented. This provides explicit handling of feature-interactions and composition.

There is considerable flexibility and opportunism in how these smart proxies and skeletons can be implemented. If the existing dark components are written in different languages, or perhaps if they are available only in binary then, the smart-proxy and smart-skeleton elements behave in a manner similar to the existing stubs and skeletons in the middleware—this allows additional functionality to be interposed at some performance cost, but without requiring any changes to legacy components. We build upon the interceptors and smart proxies of [16], but without relying upon reflection on the server side.

In addition, this approach provides another advantage. If the source code for the legacy components is available, and the language that is in use supports evolution (through monads, aspects, continuations or other means), then the compiler shown in figure 2 B will generate boilerplate that simplifies the task of writing these adaptations. In this case, any optimization benefits available from the language infrastructures will not be lost.

5 Conclusion

We have illustrated the difficulties of evolving $\mathcal{DH}$ systems with an example, and proposed a multi-tiered, eclectic approach to solving this problem. Our research is animated by some important software engineering goals: *type safety*, *language interoperability*, *opportunistic* use of optimization, *modeling and traceability* to implementation.

The use “lifting” at the architectural level, and a compiler which generates boiler plate for lifters, provides support for *compositionality.*

References

[1] *Communications of the ACM, Special Issue on Aspect-Oriented Programming*, October 2001.

[2] Jung Pil Choi. Aspect-Oriented Programming with Enterprise Javabeans. In *Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC’00)*, 2000.

[3] Yvonne Coady, Alex Brodsky, Dima Brodsky, Jody Pomkoski, Stephan Gudmundson, Joon Suan Ong, and Gregor Kiczales. Can AOP support extensibility in client-server architectures? In *Proceedings, ECOOP Aspect-Oriented Programming Workshop*, 2001.

[4] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *ACM SIGSOFT FSE*, 2001.

[5] Tancmothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, 1999.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj, 2001.

[7] 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.

[8] T. Ledoux. OpenCorba: A reflective open broker. *Lecture Notes in Computer Science*, 1616, 1999.

[9] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. Quo aspect languages and their runtime integration. In *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Systems for Scalable Components*, 1998.

[10] Priya Narasimhan, Louise E. Moser, and P. M. Melliar-Smith. Using interceptors to enhance CORBA. *Computer*, 32(7):62–68, 1999.

[11] OMG. The common object request broker architecture (CORBA) `http://www.omg.org/`, 1995.

[12] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP’97—Object-Oriented Programming, 11th European Conference*, volume 1241, pages 419–443, Jyväskylä, Finland, 9–13 1997. Springer.

[13] Jon Siegel. *CORBA 3 Fundamentals and Programming.* Wiley Press, 2000.

[14] T. S. Souder and S. Mancoridis. A tool for securely integrating legacy systems into a distributed environment. In *Working Conference on Reverse Engineering (WCRE)*, Atlanta, GA, October 1999.

[15] P. Wadler. The Essence of Functional Programming. In *Symposium on principles of Programming Languages*, 1992.

[16] N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of meta-programming mechanisms for object request broker middleware, 2000.

[17] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. Generating Wrappers for command-line legacy systems—the Cal Aggie Wrap O’Matic Project. In *International Conference on Software Engineering*, May 2001.