Dynamically Evolvable Distributed Systems

Raju Pandey Scott Malabarba Tim Stapko
Brant Hashii
Department of Computer Sciences
University of California, Davis CA 95616
{pandey, malabarb, stapko, hashii}@cs.ucdavis.edu

1 Introduction

Software systems must change over time. Changing business practices, the relentless advance of technology, and the demands of end users drive this evolution. The functionality required of applications inevitably changes in response to these factors. Consequently, in order to remain viable, applications must evolve to meet new requirements. Software component evolution is a major focus of effort in software engineering [6, 13].

The vast majority of commercial software is written in a few imperative languages, such as C++ or Java [1]. For these languages, software evolution is generally a slow, static process. Since any update requires stopping a program and overwriting all or part of it, incremental updates are often impractical, and major updates problematic. For a large class of critical applications, such as business transaction systems, telephone switching systems and emergency response systems, the interruption poses an unacceptable loss of availability. What is needed, then, is more support for applications that evolve *during execution*. Dynamic evolution provides a number of benefits in addition to easing upgrades to critical software.

The Evolvable Systems Project (ESP) is developing techniques for safely and securely evolve distributed Java applications. Our current research involves (i) design and implementation of a type-safe model for dynamically modifying Java programs on a single host, (ii) application of the model for developing an adaptive security model, and (iii) extension of the single host model to a distributed environment. We briefly describe them below.

2 Type safe dynamic evolution of Java Programs

We have developed a model [8] for dynamic evolution of Java programs, which preserves the syntax and semantics of Java. Doing so ensures compatibility with existing code, and provides greater ease of use as developers do not need to learn new language constructs. This constraint requires that we preserve the type safety characteristics of a program throughout its execution. Type safety encourages the development of safer, more disciplined code. In a dynamic system, type safety can restrict wild, unsound changes, alleviating the dangers inherent in changing code. Further, many of Java's security mechanisms, e.g., separation of user and system name spaces and protection of private data, depend on the type-safety properties of Java programs. Therefore, we impose the restriction that all changes in a program preserve the type safety properties of the program.

In order to provide a convenient, backward-compatible interface, and to support changes in any Java class, we extended the Java class loader [7]. This new, dynamic class loader allows a program to define a class multiple times. The dynamic class loader implements changes in a class and any resulting changes in its instances in an executing program.

Java is increasingly being used to support distributed programming through code mobility [14]. Although appealing in terms of system design and extensibility [2], systems that support mobility are vulnerable to malicious mobile code. The Java programming environment provides several security mechanisms [3, 4] for protecting hosts from malicious applets. Support for dynamic evolution, however, raises additional security issues, as malicious applets may use the dynamic class mechanism to modify the classes that enforce specific security policies of a host. Therefore, the dynamic class loader implements a security model that ensures that Java programs can dynamically modify only those resources to which they are authorized. We enforce this policy using name space separation and resource access control.

We implemented support for dynamic classes by modifying Sun's Java virtual machine (JDK 1.2). Dynamic classes can be implemented in several ways: by changing the language, through library-based support, or by modifying the virtual machine. As stated above, we did not wish to change the language. Library-based support proved to be too awkward and inefficient for our requirements. Thus, we chose to directly modify the virtual machine. We performed several experiments to measure the performance characteristics of our implementation. The experiments show that dynamic classes add about 6-10% of overhead to Sun's JVM. Further, the cost of updating classes is moderate.

3 Dynamically adaptive security

We have developed [5] a security infrastructure that supports dynamic policies. The infrastructure uses a declarative policy language to specify access constraints. It enforces these constraints by performing binary editing on programs and resources [12]. In addition, the infrastructure provides a runtime meta-interface by representing access control policies as first class objects. The user can inspect, add, delete, and modify security policies at runtime. This mechanism supports dynamic security environments that adapt to unanticipated operating condition changes and system evolution. For example, the meta-interface is useful in large distributed systems, where the local policies in individual clusters must be discovered in order to construct and enforce global policies, and to verify consistency among the different local policies.

We have implemented the infrastructure by generating binary code for each security policy on the fly, and integrating this code directly into the protected resource. We support dynamic evolution of security policies by using dynamic classes, which allow the system to generate *new* interposition code and add it to previously instrumented classes.

4 Dynamic evolution of distributed applications

We are extending our single host dynamic evolution model for distributed applications. Clearly, These programming paradigms are not mutually independent. Distributed systems that support dynamic evolution are rare. The difficulty is clear: the problems involved in dynamic evolution are aggravated by network latency, packet loss, and the discontinuity between the namespace models implemented by runtime systems on single hosts and those used to link applications over a network.

In addition, our current implementation is based on a Java virtual machine that runs on a single host, in a single process. The techniques we use to discover dependencies between classes and update objects in memory simply do not scale to multiple VMs. It is much more difficult to discover and handle dependencies between code and objects on a network; the same is true for different versions of a class.

A distributed programming model must specify the semantics of naming and moving components between hosts. Most models accomplish this by adding a secondary programming layer: programmers code to a specification, such as an IDL, that is external to the language they work in. They define the semantics in terms of abstractions, such as services or resources, that are the units of distribution. Programmers access them by interfaces; the implementation, defined in terms of language constructs such as objects or hardware access, is hidden. This technique is well motivated. Often, it is precisely the abstraction of "services" that programmers are concerned with. The secondary programming layer hides language and implementation details. For applications that require cooperation between multiple language platforms, or where the programmer wishes to hide class and object details, this approach is optimal.

Unfortunately, the extra programming layer reduces transparency in distributed programming. Language-level constructs cannot serve as units of distribution, applications require extra code and design work to be properly distributed, and the programmer must learn additional mechanisms. Dynamic evolution models that act on abstract components, as in dynamic component architectures, may integrate well with abstract distributed programming models – *if* designed to do so. However, if more fundamental constructs, such as classes, can change, the advantage is lost.

Consider, for instance, a simple code distribution mechanism. A server contains the most current versions of a group of classes, which may change at any time. A number of clients use these classes in perpetually running applications. Upon any class change, the server notifies its clients, which immediately download and activate the new version. In this case, the simplest specification for client behavior is "get class C from host H". Any secondary namespace system is unnecessary complexity for the programmer, who is concerned with actual Java class names and not abstractions. The discontinuity between language-level and secondary namespace models becomes an impediment when dealing with dynamic evolution at the language level. We have extended Java's class namespace model to permit binding across hosts, or even different namespaces on the same host; Generally, Java's class loader binds class names to their implementations automatically: upon encountering a symbolic reference to a class, it attempts to load the class file and create a class object. The symbolic reference is then resolved to point to the newly created class object. User-defined class loaders can fine-tune this process, for instance by loading class files from the network instead of the local disk. Our model allows users to explicitly bind class names in one namespace to class definitions previously loaded in another namespace. Further, the model allows class loader hierarchies to span a network. The mechanism is secure: users can restrict inter-namespace binding as desired, and all communication is authenticated.

We accomplish this using a variant of publish-and-subscribe [11]. In actuality, each host has its own copy of the class definition. Hosts can publish classes they define, and subscribe to classes published by other hosts. Any update of a published class is automatically pushed to subscribers. Hence, the definitions on publisher and subscriber are always identical; the class name in the subscriber's namespace is effectively bound to the definition in the publisher's namespace. We use Java Remote Method Invocation (RMI) [10] for the communications infrastructure. The programming model presented to the user has several additional characteristics. Class changes in any given namespace are safe, due to the dynamic class mechanism; users can change classes freely. Further, these changes can propagate over the network as desired, and affect both active and inactive classes. The model supports simple distributed class namespaces, such as the client/server system described above, as well as more complex hierarchical, branching structures. Extensions to Java's standard model are slight; the API is very small and easy to use. Finally, using Java public key verification [9] with signed objects, we guarantee the integrity of class data passed over the network.

Our implementation as it currently stands has certain drawbacks. The workgroups that namespaces are organized into are static, due to the RMI API and the difficulty of ensuring integrity in initial public key transfer. More importantly, our distributed namespaces apply only to classes; mobile objects are not explicitly supported. Nothing prevents the programmer from using them, but incoming objects are not checked for class version conflicts. Further, the user is not entirely insulated from class dependency issues. Thus, the user bears some responsibility for preventing conflicts.

We are currently developing techniques for addressing many of these issues.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[2] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In Jan Vitek and Christian Tschudin, editors, *Mobile Object Systems. Towards the Programmable Internet. Second International Workshop, MOS '96*, number 1222 in Lecture Notes in Computer Science, pages 25–47, Linz, Austria, July 1997. Springer-Verlag. Also available at http://www.research.ibm.com/massdist/mobag.ps.

[3] J.S. Fritzinger and M. Mueller. Java Security. JavaSoft White Paper, 1996. http://www.javasoft.com/security/whitepaper.ps.

[4] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.

[5] B. Hashii, S. Malabarba, R. Pandey, and M. Bishop. Extensible security policies for mobile java programs. In *The proceedings of the 9th International World Wide Web Conference*, pages 77–94, Amsterdam, May 2000. Elsevier.

[6] Robert Laddaga and James Veitch. Dynamic object technology. *Communications of the ACM*, 40(5):36–38, March 1997.

[7] S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. Draft. JavaSoft, Sun Microsystems, April 1998.

[8] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *In the Proceedings of the European Conference on Object-Oriented Programming.*, 2000.

[9] Sun Microsystems. *Java Security Articles*. Sun Microsystems. http://developer.java.sun.com/developer/technicalArticles/Security/index.htm

[10] Sun Microsystems. Java remote method invocation – distributed computing for Java. Technical report, Sun Microsystems, Inc., November 1999. http://java.sun.com/marketing/collateral/javarmi.html.

[11] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus – an architecture for extensible distributed systems. *ACM Operating Systems Review*, 27(5):58–68, December 1993.

[12] R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In *13th Conference on Object-Oriented Programming. ECOOP'99*, Lecture Notes in Computer Science. Springer-Verlag, June 1999.

[13] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for a dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[14] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, September 1997.