The Glasshouse - A Reflective Container for Mobile Code

Ian S. Welch
Department of Computing
University of Newcastle upon Tyne
Email: i.s.welch@ncl.ac.uk

June 11, 2001

1 Introduction

Mobile code has become mainstream, Java's applets provide a familiar example. Other examples, where the code may be loaded from a local store but may have been provided by a third party, are Enterprise Java Beans [1] and Microsoft's COM [2]. Here, components execute within a container that provides services to the component. This container and component programming model is very similar to mobile code architectures where mobile code is executed within some form of execution platform.

In this position paper, I describe how the component and container model is related to behavioural reflection and outline a proposal to build a reflective container for components called the *Glasshouse* that allows the use of intrusion-tolerant services, and supports some intrusion-tolerance for components.

2 Containers and Components

Figure 1 shows a component and container framework. Components can represent mobile code or third party code that has been loaded from a local file store. In some instantiations of the framework, the dynamically loaded code comes from local file system and in others the dynamically loaded code can come from a remote location. The common idea is that code is loaded into a container where it executes. The component can then take advantage of local services provided by the container (for example, transactional support or security).

The components do not have to explicitly invoke container services. A common way to achieve transparency is to generate proxies for the components. References to the components are then passed to clients instead of the real reference. This means that the proxy can now intercede in all calls from clients to the components and non-functional requirements can be enforced at this point. In principle, this allows a component to be deployed in different servers without any need to modify its source code. However, it is important to note that only external client access control is mediated by the proxy. Internal references to the components will allow sub-components or other components executing in the same container to access them directly and bypass control of the component or server. This could be a real problem where components may be attempting

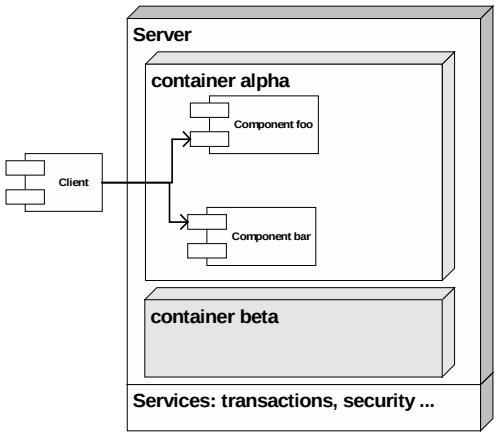

Figure 1: Container and Components

to observe or interfere with each other's execution. For example, a company may wish to use two components from competitors who are mutually distrustful.

3 Relationship to Behavioural Reflection

The use of proxies to implement the addition of non-functional properties can be seen as a limited application of behavioural reflection [3]. Behavioural reflection allows a system to dynamically change its behaviour at runtime. Metaobject protocols [4] is an object-oriented approach is implementing behavioural reflection. Here, a metaobject class is associated with an object class. This metaobject class can be used to override the default behaviour of the object execution model. The interface of the metaobject class is the metaobject protocol. This protocol narrowly defines the possible changes to the object execution model. A common way to implement metaobject classes is to generate proxies for the object class that intercept every method invocation sent to the object class. Once interception has taken place then control can be switched to the associated metaobject class and the method execution delegated by the metaobject class to the object class. As the metaobject class has control over the execution of the method, it can manipulate the semantics of the method execution. This is effectively how component behaviour is adapted by containers. However, they combine the metaobject class and the proxy. The metaobject class is then used to invoke server services according to the declarative specification associated with the component.

Although reflective, the approach taken by current container and component models are typically inflexible. They do not allow easy customisation of the metaobject. Configuration information associated with components is used to generate the metaobjects and invoke known services. A programmer cannot extend the default metaobject, and implement new services or vary the way in which current services are used.

The main drawback of using proxies to implement behavioural reflection is that they can be bypassed. This could allow illegal access to a component, if not from external clients then from other local code. One approach to lessening the possibility that the meta level can be bypassed is to not use a proxy at all! Instead of using indirection

and delegation, the approach is to modify the object class implementation. A highly flexible approach is to add hooks by rewriting the compiled object class. The reflective Java implementation *Kava* [5] implements behavioural reflection in this way. A reflective container that provides flexible, yet tight, control over components could be implemented using *Kava*. Aspects of the execution of code that can be controlled reflectively include field access, method invocation and execution, and exception handling. In addition it could invoke intrusion-tolerant services as required. The proposed reflective container implementation is called a *Glasshouse*.

4 Intrusion-Tolerant Containers

Such a reflective container could also be used to support intrusion tolerance. An intrusion tolerant system can tolerate some successful attacks but still provide a service [6]. There are two aspects to this problem. The first is to provide intrusion-tolerant services for use by the components, and the second is to implement some form of local intrusion detection and ability to intervene in the execution of a component. This could be achieved by associating a metaobject instance with each component that would perform the following functions:

- Automatically invoke intrusion-tolerant services such as a transaction service, calls to authorisation servers to check if clients can access the component, persistence service, etc. This helps prevent intrusion through attacks on services the components rely upon.
- Monitor the execution of the component and pass monitoring information as required to a distributed intrusion detection system. This helps prevent intrusion by components that are being treated as insiders and are executing within the container.

5 Related Work

I am not describing a single completely novel idea. Reflective middleware, security enforcement through the rewriting of code, and intrusion detection systems are not new concepts. What I am describing is a proposal to build an integrated platform that draws upon these three areas. The aim is to have a flexible platform for experimenting with approaches to tolerating intrusions by outsiders and insiders. In this section I briefly review some related work in the area of reflective middleware and security.

5.1 Reflective Middleware

JavaPod [7] is the closest reflective middleware system to our proposed work. It is an example of reflective container-component style middleware. It defines three main concepts: server, container and connector. A server provides execution support for containers which, in turn, provide execution support for components. Containers represent the system part of components. Through component encapsulation and interposition they manage properties like persistency, synchronization, replication, etc. Containers

use the server-provided services. For example, if a server provides a database, a container makes the binding between the database and a persistent component. Connector abstracts different types of binding between components (client-server, one-to-many, etc.). JavaPod is implemented using a reflective extension to Java that supports object extension. Although JavaPod supports transparent adaptation of components it does not support intrusion detection.

The idea suggested here is to develop a similar programming model but one that explicitly supports intrusion detection, and is implemented using our own portable reflective technology [5].

5.2 Security

There are two aspects to security in our proposed system. This first related to enforcement, and the second relates to intrusion detection.

5.2.1 Enforcement

Using code rewriting to implement security is increasingly seen as a flexible and powerful way implement security enforcement for applications [8][9][10][11][12]. For example, Java bytecode rewriting has been used to implement fine grained access control [11], resource monitoring and control policies [8]. Early approaches suffer from a lack of generality. Their whole implementation needed to be changed if the non-functional properties they are enforcing were varied, for example a new type of security model needs to be enforced rather than just a different instance of the same security model. More general approaches such as *SASI* and *Naccio* provide greater control over code execution and more flexible policy specification. *SASI* and *Naccio* extend this early approach by separating rewriting policy from rewriting mechanics.

SASI [10] uses a security automaton to specify security policies and enforces policies through software fault-isolation techniques. The security automaton is merged into application code by a rewriter. It adds code that implements the automaton directly before each instruction. The rewriter is language specific (the authors have produced one for x86 machine code, and one for Java byte code). Partial evaluation techniques are used to remove unnecessary checks. One of the problems the authors found when applying SASI was that the lack of a "friendly" policy language made it difficult to use. This has recently led to the development of PoET/PSLang (Policy Enforcement Toolkit/Policy Specification Language) [12] which uses an imperative, event-oriented language with a Java style syntax.

Naccio [9] allows the expression of safety policies in a platform-independent way using a specialised language and applies these policies by transforming program code. A policy generator takes resource descriptions, safety policies, platform interface and the application to be transformed and generates a policy description file. This file is used by an application transformer to make the necessary changes to the application. The application transformer replaces system calls in the application to calls to a policy-enforcing library. Naccio has been implemented both for Win32 and Java.

Naccio provides a high level way of specifying application security that is platform-independent but it is limited in what can be controlled. For example, Naccio cannot specify a safety policy that prevents access to a particular field of an object by other

objects. Also because Naccio relies on renaming of methods there is the possibility that the enforcement mechanisms could be bypassed.

The approach I intend to puruse is similar to SASI, PoET/PSLang and Naccio. It also uses rewriting to achieve control over application level components in order to enforce security. However, in this approach the existing Java security model is extended by defining a new type of Java security permission that can take the current state of a component into account when determining if an operation should be allowed or disallowed (I have already partially explored this approach with colleagues in [13]). A drawback of this approach is that it enlarges the trusted computing base as the Java compiler has to be considered trusted whereas a specialised policy language compiler may be smaller in size and therefore easier to verify. However, the advantage is that it does not require programmers to learn a new language in order to specify security policies and it can make direct use of application level abstractions.

5.2.2 Intrusion detection

The idea suggested here is to use metaobject instances to monitor the behaviour of applications. This is similar to a host-based approach to intrusion detection. What is different is that sensors are embedded within components instead of embedded in the execution environment. There is recent work by [14] which explores a similar area. Our approach differs in that in his approach the instrumentation must be done manually which means it must be reapplied manually when the base application changes, it makes it difficult to focus on what is the application code and what is the monitoring code etc. By using a reflective technology it is possible to write enforcement rules using application level entities, and apply these monitors automatically.

6 Summary

This position paper outlines of some thoughts on the design of a reflective container for components that supports intrusion-tolerance. Our approach is to use our reflective technology to implement a container-component style programming model that supports transparent invocation of intrusion-tolerant services and supports monitoring and enforcement of security on components.

Acknowledgements

I would like to acknowledge the financial support of the IST Programme RTD Research Project IST-1999-11583 MAFTIA (Malicious- and Fault- Tolerant Internet Applications). The term intrusion-tolerant container was provided by Robert Stroud, and some of the ideas in this paper have drawn upon discussions with him.

References

[1] Inc. Sun Microsystems, "Enterprise java beans specification".

[2] N. Brown and C. Kindel, "Distributed component object model protocol – dcom/1.0", Tech. Rep., Microsoft Corporation, 1998.

[3] Pattie Maes, "Concepts and experiments in computational reflection", in *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'87)*, Orlando, Florida, 1987, pp. 147–155.

[4] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow, *The Art of the Metaobject Protocol*, Massachusetts Institute of Technology, 1991.

[5] Ian Welch and Robert Stroud, "Kava – using byte-code rewriting to add behavioral reflection to java", in *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, San Antonio, Texas, 2001, pp. 119–130.

[6] C. Cachin, J. Camenisch, M. Dacier, Y. Deswarte, J. Dobson, D. Horne, K. Kursawe, J.-C. Laprie, J.-C Lebraud, D. Long, T. McCutcheon, J. Muller, F. Petzold, B. Pfitzmann, D. Powell, B. Randell, M. Schunter, V. Shoup, P. Verissimo, G. Trouessin, R. J. Stroud, M. Waidner, and I. S. Welch, "D1: A reference model and use cases", Tech. Rep. D1, MAFTIA Project, 2000.

[7] Eric Bruneton and Michel Riveill, "Javapod: an adaptable and extensible component platform", in *RM2000 - Workshop on Reflective Middleware*, New York, USA, 2000.

[8] G. Czaijkowski and T. von Eicken, "Jres: A resource accounting interface for java", in *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'98)*, Vancouver, Canada, 1998, pp. 21–35, ACM.

[9] David Evans and Andrew Twyman, "Flexible policy-directed code safety", in *Symposium on Research in Security and Privacy*, Oakland, California, 1999, pp. 32–45, IEEE Computer Society.

[10] Úlfar Erlingsson and Fred B. Schneider, "Sasi enforcement of security policies: A retrospective", in *1999 Workshop on New Security Paradigms*. 1999, pp. 87–95, ACM.

[11] R. Pandey and B. Hashii, "Providing fine-grained access control for java programs", in *ECOOP'98*, Lisbon, Portugal, 1999, pp. 1405–1430, Springer-Verlag.

[12] U. Erlingsson and F. B. Schneider, "Irm enforcement of java stack inspection", in *IEEE Symposium on Security and Privacy*, Oakland, California, 2000.

[13] Ian Welch and Robert Stroud, "Using reflection as a mechanism for enforcing security policies in mobile code", in *ESORICS'00*, Toulouse, France, 2000, Springer-Verlag.

[14] Eugene H. Spafford and Diego Zamboni, "Design and implementation issues for embedded sensors in intrusion detection", in *RAID 2000 - Third International Workshop on the Recent Advances in Intrusion Detection*, Toulouse, France, 2000, Extended abstract, not published in proceedings.