—In Proceedings of Workshop on Reflective Middleware (RM2000), pp.19–20, New York, April 2000—

Wrapping Class Libraries for Migration-Transparent Resource Access by Using Compile-Time Reflection

Hiroshi Yamauchi* Hidehiko Masuhara† Daisuke Hoshina*
Tatsurou Sekiguchi* Akironi Yonezawa*

1 Introduction

Programming systems with mobile objects (or mobile agents) are becoming popular for constructing network applications. In those systems, a group of running objects can migrate across the network, which reduces communication overheads and gives more flexibility in application design. One of the problems is that most resources in those systems are not migration-transparent. When an application object migrates to from a host to another, the object can no longer access resources that are intrinsic to the former host (*i.e.,* what we call *stationary resources*). On the other hand, some resources such as a window on a screen (*i.e.,* what we call *cloneable resources*) can be copied to a remote host. They still are not migration-transparent because application programs have to clean up and re-initialize them at migration.

Middleware technologies such as CORBA provide fundamental mechanisms to remotely access stationary resources. In fact, several studies proposed migration-transparent class libraries by "wrapping" existing libraries[3]. It is nevertheless onerous to write such wrappers by hand. Therefore, some mechanisms that generate appropriate wrappers would be useful.

*Department of Information Science, University of Tokyo. http://web.yl.is.s.u-tokyo.ac.jp/amo/

†Department of Graphics and Computer Science, Graduate School of Arts and Sciences, University of Tokyo.

2 Our Approach

Overview

Our approach is to automatically generate *wrapper classes* for resource classes by using techniques similar to compile-time reflection, and let mobile objects use the wrapper objects instead. The wrapper objects are mobile; they can move across the network along with application objects. Depending on the types of the resources, they (1) serve as proxies of stationary resource objects or (2) copy cloneable resource objects and perform clean-up and re-initialization procedures at migration. We designed the system for our Java-based mobile object system[4].

Wrapper Classes for Resources

Figure 1 shows how a mobile object accesses system resources via wrapper classes. The detailed architecture of wrapper classes are described in the other literature[3]. In the figure, a mobile application object app on host A is using a window frame and a file. Instead of directly accessing objects of the original class library (shown as ovals), it uses *wrapper objects* (shown as rectangles). When app migrates to host B, the wrapper objects in the dashed line also move to B. As for a cloneable resource like frame, the wrapper object creates a clone of the original one and performs re-initialization. As for stationary resources like file, the wrapper object serves as a proxy for the object at A, by forwarding method invocation requests.

A wrapper class has the identical signature to the original resource class; its class name and method names are the same to the ones of the

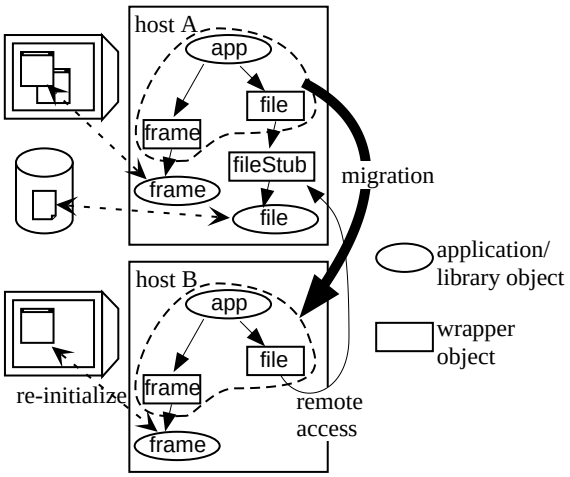

Figure 1: Wrapper objects for migration-transparent resource access.

original class, but it is defined in a different package. This helps application programmers to easily switch to wrapper classes by merely changing names of importing packages in their program.

One of the main purposes of wrapper classes is to convert parameter/return values at method invocation. A method of the wrapper class calls the method of original object after "unwrapping" objects in the parameter values, and "wraps" objects in the returned value before returning it to the caller.

For stationary resources, each method in a wrapper class forwards method invocation requests by using Java RMI mechanisms. Since most resource classes do not accept RMI in Java, our system also generates a *stub class* for receiving RMI requests. (The object fileStub in Figure 1.)

For cloneable resources, a wrapper class has a method that creates a copy of the original object at the migrated host and initializes the clone object.

Our system also generates several auxiliary classes for calling protected methods in original library classes and for executing static methods at an arbitrary host.

Metaclasses as Wrapper Generators

In order to generate wrappers, we use compile-time reflection techniques that are similar to OpenC++[1]. A metaclass in our system extracts type information of its base-class by using introspective reflection mechanisms[2], and generates definitions of wrapper class and auxiliary classes. There are two standard metaclasses for stationary and cloneable resources, respectively. The user writes a description file that associates an appropriate metaclass for each resource class in a library. The user can also extend metaclasses for cloneable resources in order to insert clean-up and re-initialization procedures in wrapper classes, although this feature has not been implemented yet.

3 Summary

In order to transparently access system resources from mobile objects, we designed and implemented wrapper generators by using compile-time reflection techniques. A wrapper generator in our system can be considered as a metaclass of resource classes, and generates wrapper classes that can be used as replacements of those resource classes.

Thus far, we have tested our system to file I/O, network communication and GUI component libraries in Java `java.io`, `net` and `awt` packages, respectively. The generated wrappers are used by mobile objects written in our mobile programming system JavaGO[4].

References

[1] S. Chiba. A metaobject protocol for C++. In *OOPSLA'95*, pp. 285–299, 1995.

[2] JavaSoft, *Java Core Reflection: API and Specification*, 1997.

[3] D. Hoshina, T. Sekiguchi, and A. Yonezawa. Class Library Supporting Continuous Operation to Resources across Migration. Submitted for publication. 2000.

[4] T. Sekiguchi, H. Masuhara, and A. Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. In *COORDINATION'99*, LNCS 1594, pp. 211–226, 1999.