Abstract

Single-language runtime systems, in the form of Java virtual machines, are widely deployed platforms for executing untrusted mobile code. These runtimes provide some of the features that operating systems provide: inter-application memory protection and basic system services. They do not, however, provide the ability to isolate applications from each other, or limit their resource consumption. This paper describes KaffeOS, a system that provides these features for a Java runtime. The KaffeOS architecture takes many lessons from operating system design, such as the use of a user/kernel boundary.

The KaffeOS architecture supports the OS abstraction of a process in a Java virtual machine. Each process executes as if it were run in its own virtual machine, including separate garbage collection of its own heap. The difficulty in designing KaffeOS lay in balancing the goals of isolation and resource management against the goal of allowing direct sharing. Overall, KaffeOS is up to 11% slower than the freely available JVM on which it is based, which is an acceptable penalty for the safety that it provides. KaffeOS is substantially slower than commercial JVMs, but exhibits much better performance scaling in the presence of uncooperative code.

1 Introduction

The need to support the safe execution of untrusted programs in runtime systems for type-safe languages has become clear. Language runtimes are being used in environments for executing untrusted code: for example, applets, servlets, active packets [38], database queries [13], and kernel extensions [5]. Current systems (such as Java) provide memory protection through the enforcement of type safety and secure system services through a number of mechanisms, including namespace and access control. Unfortunately, malicious or buggy applications can deny service to other applications. For example, a Java applet could generate excessive amounts of garbage and cause a Web browser to spend all of its time collecting dead objects.

To support the execution of untrusted code, type-safe language runtimes need to provide mechanisms to isolate and manage the resources of applications, analogous to those provided by operating systems. Although other resource management abstractions exist [3], the classic OS process abstraction seems appropriate. A process is the basic unit of resource ownership and control; it provides isolation between applications. On a traditional operating system, untrusted code can be forked as its own process; CPU and memory limits can be placed on the process, and the process can be killed if it is uncooperative. Current type-safe language runtimes do not support such functionality.

A number of approaches to isolating applications in Java have been developed by others over the last few years. An applet context [7] is an example...