Experience with Secure Multi-Processing in Java*

Dirk Balfanz
balfanz@cs.princeton.edu
Department of Computer Science
Princeton University

Li Gong
li.gong@sun.com
JavaSoft Division
Sun Microsystems, Inc.

Abstract

As the Java™ platform is the preferred environment for the deployment of network computers, it is appealing to run multiple applications on a single Java-enabled desktop. We experimented with using the Java platform as a multi-processing, multi-user environment. Although the Java Virtual Machine (JVM) is not inherently a single-application design, we have found that the implementation of the Java Development Kit (JDK) often implicitly assumes that the JVM runs exactly one application at any one time.

In this paper, we report the limitations we encountered and propose improvements to several aspects of the Java technology architecture, including its security features. We have implemented all the proposed changes in a prototype based on an in-house beta version of JDK 1.2. Our prototype uses a Bourne shell-like command line tool to launch multiple applications (such as Appletviewer) within one JVM.

1 Introduction

A Java™ Virtual Machine (JVM) [6, 8] is typically used to run exactly one Java application. One property that distinguishes the Java platform from other environments is its support for *mobile code*: A Java application such as a Web browser can execute foreign code (Java applets) securely within the browser.

Current Java implementations usually express their security policy in terms of code source, which is characterized by both digital signatures on the mobile code and the network origin (i.e., network location and address) of the mobile code [5]. When multiple applications run in the same JVM, we need to provide a security framework that accommodates the need to both protect the Java system from (potentially malicious) mobile code and protect (potentially mutually hostile) applications from each other.

In this paper, we report our experiences in attempting to implement multi-processing capabilities for the Java platform. We describe changes and additions to the Java system APIs (part of JDK) that we found necessary and useful. We also suggest how user-based access control (to specify policies about *who* is allowed to do what) can be introduced to work with the existing code source-based access control (*which code* is allowed to do what).

To demonstrate feasibility, we have implemented all the changes we propose in a prototype, which uses a Bourne shell-like command line tool to launch multiple applications from within one JVM. Our prototype is based on an in-house beta version of the Java™ Development Kit 1.2 (JDK).

The rest of this paper is organized as follows. We first argue for the case of running multiple applications inside one JVM. Then, in Section 3, we examine how a JVM currently executes an application, paying special attention to issues that intrinsically render it a single-application environment. In Sections 4 and 5, we discuss features that are missing from the current JVM and show what can be added and changed to make the Java platform a more friendly environment for multi-processing. Finally we discuss related work and conclude the paper with some directions for future work.

Our work necessarily includes a variety of issues, such as the refinement of the Java security architecture. These issues, however, are outside the scope of this paper, so we only touch upon them briefly. A more comprehensive version of this paper is available as a technical report [1].

2 The Case for a Single JVM Solution

It is plainly obvious that it is desirable to run multiple Java applications on a single desktop at the same time. In fact, this can be done already by just launching multiple JVMs from within the underlying operating system. Therefore, the question we must tackle first is why is it particu-

*This work was performed in part when Dirk Balfanz visited JavaSoft during the summer of 1997. This paper does not necessarily reflect the official opinions of Princeton University or Sun Microsystems, and does not imply any products or features from Sun Microsystems.

larly attractive to run multiple applications in one JVM. Here are a number of reasons:

- A small device or an old computer system may be under-powered and equipped with inadequate memory such that it is crippling to try to start multiple JVMs.
- Sometimes, there is no underlying O/S in which to launch multiple JVMs, and all we have is a single JVM that runs on the bare hardware. JavaOS [11] is an example of such a system. Since most of the criticism that we apply to the JDK also applies to JavaOS, it will become clear that JavaOS is not very good at running multiple applications, let alone applications run by different users. For example, to switch to a different user in JavaOS, the previous user must be logged off and sometimes the machine has to be rebooted to make sure that the system state is correctly initialized.
- A single JVM promises to allow for fine-grained sharing between the various components of the system, including sharing between applications.
- A mutli-user JVM is, in some sense, a single address-space O/S. Single address-space operating systems are a focal point of current research in operating systems [2, 12, 9]. Using software-based protection instead of hardware-assisted protection through multiple address spaces, these systems promise superior performance and improved functionality [9, 13]. A multi-processing JVM could be a full-featured testbed for research on single address-space O/S's.

3 Running a Single Application

3.1 The Lifetime of an Application

A Java application is typically invoked by typing `java MyClass` on the command line.[1] This will start the JVM, which is a process in the underlying operating system (O/S).[2] Before the O/S transfers control to the JVM, it makes sure that the process is properly initialized. This initialization includes setting of open file descriptors for standard input and standard output, user id's, and process id's. The values for a number of these parameters are taken from the application that created the JVM process, which in our example was the Unix shell.

The JVM itself is a multi-threaded process. Java uses either a kernel- or user-based thread library to start up a number of threads as soon as the JVM starts running. These threads include a garbage collector, a thread to execute finalizers, and an idle thread to fall back on when no other work needs to be done. But most importantly, it includes one thread that interprets Java bytecode, starting with the first instruction of the `main` method in the class we specified (in our example `MyClass`).

Whenever a new class is needed to execute the bytecode (for example, when an instance of that new class is to be created), that class is dynamically *linked* into the JVM – the external class file representation is converted into an internal representation that the JVM can use to call methods of this class, access members, and so on. Often, some initialization of these classes is performed. For example, during initialization of the `System` class, streams are created that point to standard input, output, and error.

Our Java program may also choose to spawn new threads, which will be scheduled just like all the other threads already running inside the JVM.

How does a Java application (and, hence, the JVM process) ever finish execution? One way to do this is to call the `System.exit()` method, which will cause the underlying `exit` system call to be invoked. This will stop the JVM process itself, and therefore all the threads that were active within it.

Java threads may also come to a "natural end". For example, when the `main` method returns, the associated thread is destroyed. However, that does not necessarily mean that the JVM will stop executing, as there can still be other threads around, including the threads that were created when the JVM started. To prevent the JVM from running forever, Java distinguishes between *daemon* and *non-daemon* threads. When a thread is created, it is marked as either daemon or non-daemon. Whenever a thread finishes execution, the JVM checks to see if there is at least one non-daemon thread remaining. If so, the JVM continues to execute all the threads. If all remaining threads turn out to be daemon threads, the JVM exits, stopping all those daemon threads in the middle of whatever they were doing.

The threads that are created at JVM start-up time are all daemon threads. The thread that executes the `main` method is a non-daemon thread. This has the (desirable) effect that when the `main` method returns, the JVM usually exits. This whole life cycle is illustrated in Figure 1.

Note that sometimes an application implicitly creates non-daemon threads that run forever, which will cause the JVM not to exit unless the Java application calls `System.exit()`. This is for example the case when a Java application uses AWT components. When the AWT toolkit is initialized, a non-daemon thread is started that dispatches events and calls into call-back code provided by the application. This thread will run forever and keep the JVM from exiting until explicitly destroyed by the

[1] The actual command may not be called `java`, depending on which particular product is used.

[2] In our explanations, we usually assume that the underlying O/S is Unix, but apart from slight differences in the jargon, all concepts explained here apply equally to other platforms Java has been ported to.

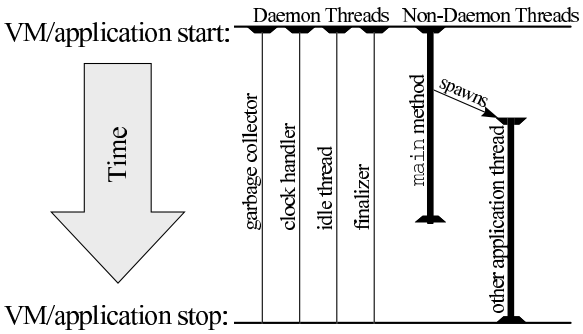

Figure 1: The lifetime of a JVM: once all non-daemon threads of an application have finished, the JVM exists even though daemon threads may still be running.

`System.exit()` call.

3.2 Event Dispatching

In the previous section we briefly mentioned AWT event dispatching. Let's now take a more detailed look. We will assume an underlying Unix with an X server, but our comments apply to other platforms as well.

When a Java application wishes to draw something on the screen, it makes a call into the appropriate library (which will then contact the X server). On the other hand, when the JVM gets notified by the X server that some user input happened, an AWT event object is created. The event object contains information about the event (for example, where a specific mouse click happened). It is then put on a queue. A centralized *event dispatcher* thread will pick up events from that queue and call the appropriate methods to handle the event. For example, if an `ActionEventListener` is registered with a GUI button and the button is clicked, then the `actionPerformed()` method is called on the listener. Note that *all* callbacks are called from a *single* event dispatcher thread.

3.3 Security Policies

Since the Java platform has specifically been designed to execute possibly untrusted mobile code, great care has been taken to specify and implement a security model [4].

The Java class libraries are written in such a way that all sensitive operations call into a centralized object, the *security manager*, to check whether the callee should be allowed to invoke this operation. The security manager throws a security exception if the operation should not be allowed, effectively aborting the execution of that operation before any harm can be done.

Consider for example some Java application executing the following code:

```
File f = new File("/temp/foo");
f.delete();
```

The delete method is implemented like this:

```
public void delete() {
  securityManager.checkDelete();
  realDelete(); }
```

This assumes that `realDelete()` is a private method (and therefore not accessible directly from the application) that actually deletes the file. Every application is free to supply the security manager and implement whatever security policy it chooses (i.e. when the security manager should allow certain calls and when it should not).

In practice, which policy the security manager applies has changed with time. In early JDK implementations, the security manager distinguished between "remote" and "local" code (i.e., code that was donwloaded over the network and code that belonged to the local Java class libraries). Local code would always be allowed to make security sensitive calls, while remote code was considerably limited in that regard. In newer versions of the JDK, security managers follow a user-definable strategy that grants different permissions to different code [5]. The user can usually express policies in terms of where code comes from and/or who digitally signs the code. We say that the security manager implements a *code source-based user-definable security policy*.

4 Additional Support Needed for Secure Multi-Processing

During the course of our experiment, we found that a number of new features are needed to allow secure multiprocessing. In this section, we discuss them individually, while in the next section we describe our approach to these issues.

The first thing that we notice is that the execution of the JVM starts when we start an application, and stops when the application is finished. So, what we need is

Feature 1 *A way to launch applications into a running JVM, and a way to end them without exiting the JVM.*

In order to remain backward-compatible we want to execute the `main()` method of a class to start an application, and the "end" of an application should be defined just as it is now – no non-daemon threads of that application remaining – but it should not necessarily cause the JVM to exit.

Feature 2 *A definition of what an application is, consistent with the current notion of a Java application, but allowing multiple applications to run in one JVM.*

In Unix, processes are allowed to do certain things depending on which user runs them. If we have multiple applications running in one JVM, we also may want to grant

different applications different permissions, depending on who runs them. For example, Alice and Bob might both run the very same piece of software. When run by Alice, it should be allowed to read Alice's files, while when run by Bob it should not (by default).

However, currently there is no explicit notion of a user running a Java application.[3] Hence, we need

Feature 3 *A notion of a* user *running Java code.*

How do we associate a particular application with a user? In Unix, a user logs on to the system, and as a result of the authentication process, a shell (either graphical or command-line) is provided that runs as the authenticated user. After that, every application that is launched from the shell inherits the user-id. If we follow this scheme, we need to log on to the JVM, and we need a (graphical or command-line) shell.

Feature 4 *A notion of* logging on *to the JVM and a shell to launch other applications.*

Once we can run multiple applications and associate them with different users, we need to grant these applications privileges based on who run them. This should work in conjunction with the existing approach of basing security policies on code sources.

Feature 5 *A way to combine code source-based security policies with user-based policies.*

Some of the Java system primitives today implicitly assume that there is only one application running. For example, an application can exit the virtual machine by calling `System.exit()`, since the "system" is the same as the application. Along the same token, there is only one security manager that oversees the one running application. So, in general, we need

Feature 6 *Multi-application-aware system code.*

In particular, we explained in Section 3.2, how GUI events are dispatched. Assume that two users, Alice and Bob, are running the same program, say a text editor, within one JVM. When Alice wants to save her file, she selects the appropriate menu item. This will cause the event dispatcher thread to execute the code associated with the Save File menu item. When Bob tries to save *his* file, the very same thread will execute the very same code. Thus, there is no way of distinguishing between the two cases. However, such a distinction is crucial as, for example, we would like to avoid saving Bob's file in Alice's directory and vice versa. Thus we need

[3] It is possible to query the system properties for the user that runs the JVM, but that may or may not succeed depending on the operating system you're on, and has no implications whatsoever for the JVM.

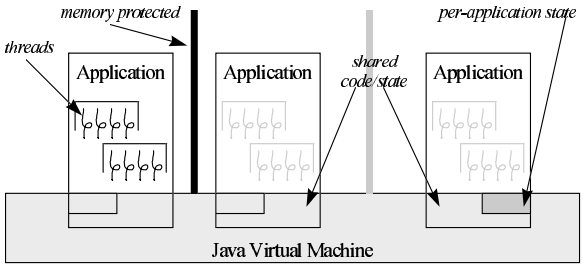

Figure 2: Applications. An application consists of a set of threads.

Feature 7 *Multi-application-aware event dispatching.*

In a multi-processing JVM, while all the applications may share the same information about "the system" (such as the version number of the O/S or the list of available HTTP proxies), clearly different applications have different ideas about what for example the standard input and output streams are. The latter is part of an application-wide state, while the former is system-wide state. In current JVM implementation there is no distinction between the two, but in a multi-processing JVM we need

Feature 8 *Distinction between application state and system state.*

One particular example of this desirable separation is the security manager. Every application is allowed to set its own security manager, making it essentially part of an application-wide state. However, there should be a system-wide security manager that governs the interactions between applications and decides which users have what permissions.

Feature 9 *A way to combine JVM-wide security policies with application-wide security policies.*

In the next section we present an improved system architecture that addresses all of the above mentioned shortcomings. We have implemented the architecture in a prototype and have written several small applications as a testbed (see Section 6).

5 Running Multiple Applications

5.1 Defining Applications

We define an application to be a set of threads. Threads not only give us a natural notion of an application but also are a convenient way to distinguish two instances of the same program running inside a single JVM (see Figure 2).

To implement this concept of an application, we created a class `Application`. An application object contains a class loader that separates the application's name space

from that of other applications. It also contains a thread group to accommodate all threads of the application, and application state such as user-id, current directory, standard input and output streams, etc.

The application class is typically used as follows:

```
1 String[] args = {"arg1", "arg2"};
2 Application app =
    Application.exec("MyClass",args);
3 app.waitFor();
```

Line 2 causes a new application to be created and started. What happens behind the scenes is the following. First, a thread group is created for the new application, together with an instance of class `Application` to hold the application-wide state. The new application state is initialized by copying values from the current application. This includes, among other things, standard streams and running user. The application's main class (`MyClass`) is loaded by a newly created class loader (this will have the effect that all other classes in the same application will also be loaded by that class loader). Then, using the Java Reflection API, the `main` method of class `MyClass` is called. This happens within a new thread in the newly-created thread group. The arguments `args` are passed to the `MyClass.main` method. Since the `main` method is executed in its own thread, the `exec` method returns immediately. Line 3 waits for an application to finish.

The new application is allowed to create threads only in its own thread group. This prevents applications from stepping on each others toes and makes it easy to associate a given thread with an application instance.

An application can be stopped using the following code:

```
// within MyClass.java
System.out.println("bye, bye");
Application.exit(0);
// we will never get here
```

The static `exit` method will find the application instance that corresponds to the currently running thread, schedule that application for destruction, and block the current thread. A background thread will eventually clean up the application, stop all threads, and close all windows that are associated with the application.

If the application does not explicitly call `exit()`, then the JVM will call the `exit` method as soon as there are only daemon threads left in the application's thread group.

It is worth noting that, in Unix, the `exit` system call closes all open file descriptors that the exiting process has. It is not always appropriate to follow this semantics here. For example, every application has at least the standard input and output streams open. It might very well be that multiple applications have their standard streams point to the same device, say a terminal. If one application now closes one of these streams, then other applications that used the same stream will no longer be able to use it.[4]

Therefore, until special APIs are available for safe stream duplication, applications may only close streams that they opened. Streams that are passed to them, like standard input and output streams must not be closed by the application.

5.2 Defining Users

Every application is associated with a user, and the `Application` class provides calls to query and set the user of the currently running application. Special privileges are needed to set the user, and these privileges are not normally granted to applications. A newly started application will inherit the running user from the currently running application.

In our prototype, our login program works similar to Unix's login mechanism. It has the necessary privileges and resets its own running user-id to be the one that it has successfully authenticated. It then spawns a shell (which will have the same running user) and waits for the shell to finish.

Note that it does not matter which user is running the login program. In fact, it might even be some sort of "null" user for bootstrapping purposes. In particular, it is not necessary to have the login program be executed by an all-powerful "super user". All we need to do is grant the login program the privilege to set its own user. This can be done through code source-based security policies, since it is the *program* that is granted the privilege, not the user running it.

5.3 User-Based Access Control

We chose to implement user-based access control as part of code source-based access control. The idea is to extend the range of the security policy so that (1) the security policy can grant permissions to a particular user and (2) the policy can also grant certain *code sources* the privilege to exercise the permissions of the running user.

We used Sun's JDK 1.2 security framework [5], and introduced a new kind of *user permission*, which can be granted to certain code sources (such as local applications).

When making access control decisions, if the JVM comes across code that has been granted this special user permission, the JVM checks the permission granted to the currently running user in addition to checking the permission granted to the code's source. The permissions granted to the code itself and the permissions granted to the user that runs the code are combined to determine whether access to certain sensitive system areas should be granted.

[4] This phenomenon is due to the fact that if one creates a new stream out of an old one using standard stream APIs, then closing the new one almost inevitably causes the old one to be closed as well.

For example, our locally installed text editor (and, in fact, all local applications) would get the permission to exercise the permissions of the user who is running it, whereas remote code (such as applets) would not. This enables the text editor to access files that belong to the user running it, but would not allow applets running in a Web browser to access these files. As a result, we can specify policies like the following.

1. All local applications can exercise their respective running users' permissions.
2. The `backup` application can read all files.
3. User Alice can access all files in `/home/alice`.
4. User Bob can access all files in `/home/bob`.

Details of how exactly the user-based access control is implemented are beyond the scope of this paper.

5.4 Event Dispatching

Recall that, in current JVM implementations, the event dispatcher thread executes *all* callbacks. This means that code invoked as the result of user input is executed by a thread that does not belong to any particular application.[5] For example, if Alice runs a text editor and chooses to save the file, the code that tries to save the file may not run under Alice's user-id. This is clearly inadequate for protection purposes.

Therefore, we reworked the AWT event-dispatching to have the following features.

- When an application opens a window, the system makes note about which application the window belongs to.
- When an event occurs in a GUI element, the enclosing window and its application are found. Then, the AWT event is put on the particular event queue of that application, where it will be picked up and dispatched by a thread that belongs to that application.

This redesign also improves responsiveness, as each application's event dispatching is now independent from other applications (see Figure 3).

The per-application event dispatcher threads are created on demand. Whenever an application first opens a window, we create an event dispatcher thread for that application. Since that thread is a non-daemon thread, we now have the same semantics for application-exit that we had before. An application that does not use the AWT may just return from its `main` method and will be automatically destroyed by the system. An application that *does* use the AWT has to call `Application.exit()` in order to finish.

[5] Whichever application happens to open a window first would implicitly start the event dispatcher.

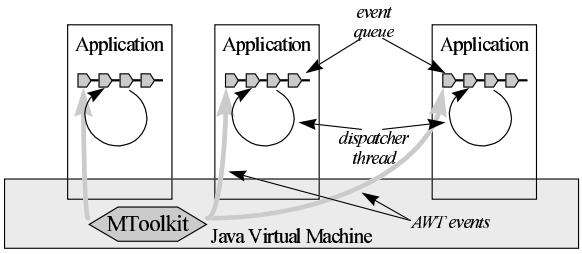

Figure 3: Multi-threaded event dispatching. Every application has its own event queue and a thread in the application's thread group delivers the events.

5.5 Reloading System Classes

As we explained before, some of the process-wide state, including standard streams, surfaces as system-global state in JDK. For example, the static variable `System.in` and its siblings are shared throughout the JVM.

Our multi-processing system, however, must maintain per-application state. For example, different applications may have different ideas about what their standard input and output streams are.

We implemented a solution that also maintains backward compatibility for existing application code. We give each application the illusion of having the JVM all for itself, yet provide each application with a slightly different view of it. This technique has previously been employed to provide differently trusted code with a different view of what the system classes are [13].

In our implementation, every application gets its own copy of the `System` class. We use a special class loader to re-load and re-define the `System` class, albeit from the same class material. Since we use a new class loader for every application, to the JVM the different incarnations of the `System` class are just different classes that happen to have the same name. Now we have a new copy of the `System` class for every application, we can set the respective `System.in`, `System.out`, and `System.err` streams to point to different things for different applications.

Based on this mechanism, we were able to easily implement input/output redirection and pipes between applications.

Note that the `System` class, in the form of system properties, also contains state that is truly JVM-wide. To make sure that such system properties are available to all applications, we placed them in a new class called `SystemProperties` that is shared between all applications (see Figure 4).

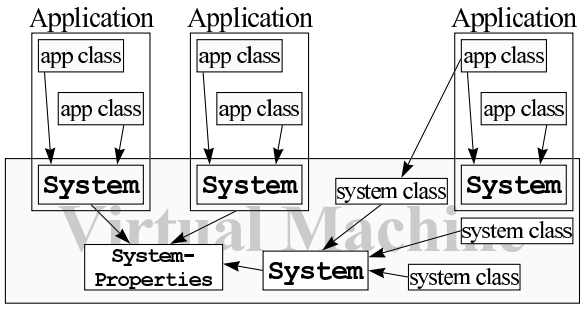

Figure 4: Reloading the `System` class.

5.6 Multiple Security Managers

We installed a global security manager (the *system security manager*) in our multi-processing JVM. It defines the rules of the game as follows:

- Applications are not allowed to access other application's threads.
- All other access control decisions are relayed to the `AccessCotroller` which means that code source-based user-specified policies are applied.

Note that applications can still set their own security managers. This is because the reference to the security manager is stored in the `System` class, of which there is a fresh copy for every application. However, these security managers would never be automatically consulted by system code. System code always consults the system security manager.

It is non-trivial to make the system security manager work with application security managers, if the latter override methods defined in the former and expect to be consulted for security decisions previously performed only by the system security manager. This is because application security managers, as application code, no longer have the same privileges that the system security manager enjoys. As a result, these overridden security check methods can fail when they should not.

We found no good use for application security managers. Instead, augment (and sometimes change) the effective behavior of the system security manager via different means, such as delegation. The details of this aspect of the security architecture and API are beyond the scope of this paper.

6 Useful Tools

As proof of usability of our multi-processing JVM, we built a few demonstration tools that included a shell, a terminal, and an application-level Appletviewer. We discuss them in some detail in the longer version of this paper [1], while a few remarks should suffice here.

Unlike Unix, applications must not close their standard streams upon exiting (see Section 5.1). Hence, it is up to the shell whether the standard streams of a process should be closed upon exit (e.g. when the stream was just a `PipeOutputStream` going to another process) or not (e.g. when standard output points to a terminal that other applications are also using).

The Appletviewer is normally part of the JDK system classes. We moved it out of the system library, effectively marking all its code as untrusted. We removed the security manager and replaced its functionality (the implementation of the traditional sandbox) by permission delegation.

We successfully run multiple instances of the terminal, together with shells, the Appletviewer, and a number of processes connected through pipes in our prototype.

7 Related Work

The idea of using software-based protection as the fundamental building block for system security is not new. For example, the operating system Pilot [10] used a safe language [7] in a single address space to provide security without a kernel. Various approaches of software-based protection have recently been reconsidered in the context of Java technology [13]. Our contribution is that our multi-processing Java environment must deal with distributed computing in the presence of mobile code and needs to explicitly address both code source-based and user-based security policies.

Many of today's research operating systems either use a single address space or load code into the kernel's address space to increase performance and enhance functionality of applications. As far as security is concerned, they use either a type-safe language [2] or provide other techniques such as proof carrying code [9] to achieve basic security properties such as memory protection. However, few of those systems can at this point go beyond memory protection and provide secure services [13]. In comparison, we focussed on the latter while not paying particular attention to performance tuning.

Recent advances in Java security have evolved from the original restricted sandbox model to a policy-based, easily configurable, fine-grained access control model [4, 5]. However, security policies are still limited to deal with code sources and not with users. This is because the JVM as it stands now is normally used in a single-user environment. Our work here expands into a multi-user environment, and solves the new problems we encountered.

In the paper we glossed over details of the underlying security architecture. While the issues of a multi-user Java environment and user-based access control are naturally interdependent, we tried to focus in this paper on the multi-

processing aspect. We do have a user-based security architecture in place, but it is beyond the scope of this paper to describe it in detail.

8 Conclusion and Future Directions

In an attempt to use the Java platform as a multi-processing, multi-user environment, we have found that the implementation of the Java Development Kit (JDK) often implicitly assumes that the Java Virtual Machine (JVM) runs exactly one application. The challenge in realizing a multi-processing Java environment is, apart from identifying and correcting these implementation weaknesses, to come up with an architecture that integrates multi-processing, mobile code, and security. Our experience shows that, with a few limited changes and additions we described in this paper, the Java platform can become an effective multi-processing, multi-user environment.

There are a few directions for further study. For example, it is conceivable that the notion of an application as a set of threads can be extended to include threads of other JVM's, possibly on other hosts.

Moreover, in our multi-processing environment, it is very appealing to use shared objects as an inter-application communication mechanism. However, such sharing of objects between different applications in different name spaces is still a delicate task and its impact on the correctness of the Java type system needs more research [3].

Finally, it appears worthwhile to further investigate the implications of reloading certain system classes. For example, there might be a hidden assumption that there is only one copy of certain classes such as `Class` and `String`. Moreover, the impact of class reloading on the safety of the Java type system is not very well understood. Our prototype so far only involved the Java class library, but it seems likely that similar implicit assumptions have been made during the implementation of the virtual machine itself.

References

[1] D. Balfanz and L. Gong. Experience with Secure Multi-Processing in Java. *Technical Report 560-97, Department of Computer Science, Princeton University*, Princeton, September 1997.

[2] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuchynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Colorado, December 1995. Published as ACM Operating System Review 29(5):251–266, 1995.

[3] D. Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[4] L. Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, May/June 1997.

[5] 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*, pages 103–112, Monterey, California, December 1997.

[6] J. Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Menlo Park, California, August 1996.

[7] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, April 1980.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Menlo Park, California, 1997.

[9] G.C. Necula and P. Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, Washington, October 1996.

[10] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and Stephen C. Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, April 1980.

[11] S. Ritchie. Systems Programming in Java. *IEEE Micro*, 17(3):30–35, May/June 1997.

[12] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, October 1996.

[13] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.