Experience with Secure Multi-Processing in Java*

Dirk Balfanz
Princeton University
balfanz@cs.princeton.edu

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

September 29, 1997

Abstract

As Java™ is the preferred platform for the deployment of network computers, it is appealing to run multiple applications on a single Java 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 Java Virtual Machine 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 architecture, including its security features. We have implemented all the proposed changes in a prototype based on the 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) [5, 7] is typically used to run exactly one Java application at any one time, where an application can be a Web browser and Java applets are fetched as part of Web pages and are executed within the browser. Other applications such as downloadable stubs for remote messaging [13] or frameworks for push technology increasingly use this ability to execute mobile Java code.

Current Java implementations usually express their security policy in terms of code identity that is characterized by both digital signatures on the mobile code and the network origin (i.e., network location and address) of the mobile code [4]. 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 proposed 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 for 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 resembles a single-application environment. In Sections 4 and 5, we discuss features that are missing from the current JVMs and show what can be added and changed to make Java a more friendly environment for multiprocessing. 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.

*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. The discussion does not bind Sun Microsystems to any particular products or features.

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 what makes it particularly attractive to run multiple applications in one JVM.

For starters, 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.

Moreover, JVM sometimes does not run as a process of an O/S, but run directly on the bare hardware (e.g., JavaOS [10]). Thus, there may not always be an underlying operating system to host multiple JVMs. In current incarnations of these systems, the multiple applications available on the system are in fact parts of one big application, where it is difficult (if not impossible) to segregate the different parts to prevent undesirable interactions. This limitation effectively confines the system to running in single-user mode. To switch to a different user, 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.

To construct a true multi-user system, it is clear that the applications must be protected from each other, just like in any other modern operating system. This can only be achieved if they are truly different applications and not different parts of the same application.

Even if the JVM runs as a process within another O/S, we should be interested in running multiple applications in one JVM. Single address-space operating system is a focal point of current research in operating systems [1, 11, 8]. Using software-based protection instead of hardware-assisted protection through multiple address spaces, these systems promise superior performance [8, 12]. Many factors contribute to this performance gain. Context switching, for example, is much less expensive if performed within one address space, because caches need not be cleared, page-table pointers don't have to be adjusted, and so on. Inter-process communication is also much cheaper in a single address space. In fact, a multi-processing JVM could be a full-featured testbed for research on single address-space O/S's and at the same time the first mainstream modern single address-space operating system.

3 Running a Single Application

3.1 The Lifetime of an Application

A Java application can be invoked typically 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] The process that causes the launch of the JVM is the shell that let us type in the command `java` in the first place. 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 is the Unix shell in our example.

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 immediately after the JVM gains control from the O/S. 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 in order to execute the bytecode (for example, when an instance of another class is 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. Examples of initializations of classes which have notable effects to the Java application include:

- When the `System` class is loaded, three streams are created that point to standard input, standard output and error file descriptors of the JVM process.
- Also in the `System` class, so called *properties* are initialized. These are values that provide information about the "system", for example the running user, the Java version, the underlying O/S version. Some of these values are taken from 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 her apply equally to other platforms that Java has been ported to.

respective value of the JVM process (e.g. the running user), some of them are hard-coded into the JVM (e.g. the Java version), and some of them are acquired by some other means (e.g. the O/S version through a system call).

The JVM interprets one bytecode instruction after another, linking classes as needed, and performing necessary initializations before any new class is actually used. Our Java program may also choose to spawn new threads, which will be scheduled just as all the other threads that the JVM is already running.

How does a Java application (and, hence, the JVM process) ever finish execution? One way to do that 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 stop 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 execution, as there can be other threads still around, including threads that were created when the JVM started or by our application. 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 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 `System.exit()` call.

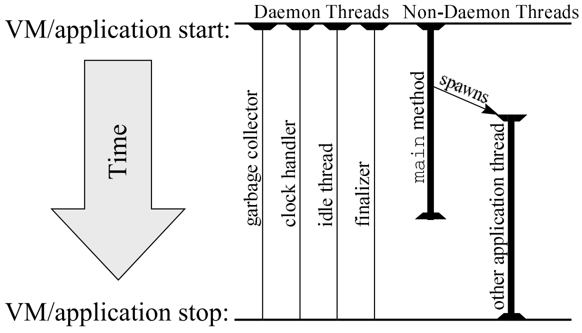

Figure 1: The lifetime of a JVM: once all non-daemon threads of an application have finished, the JVM exists even though daemon threads may still be running.

3.2 Event Dispatching

In the previous section we briefly mentioned AWT event dispatching. Now we take a more detailed look, especially at its interaction with the underlying windowing system. Our discussion assumes a underlying UNIX with the X windowing system, but our comments apply to other platforms also.

In X, a special process (the X server) has exclusive control over the high-resolution display. If an application wishes to draw something on the screen, it needs to communicate to that process what it wishes to draw. The X server will then draw on behalf of that application, making note which GUI component it drew on behalf of which application. When some input from the keyboard or mouse occurs, the X server will figure out which GUI component was the target of that input and notify the appropriate process (see Figure 2).

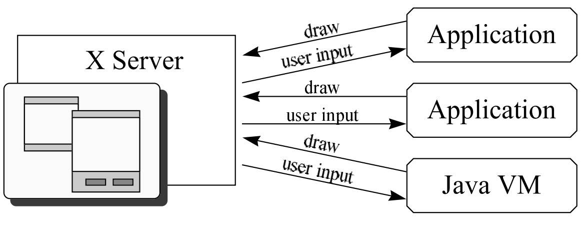

Figure 2: Event dispatching. Within the Java JVM, a single thread is used to execute all callbacks.

Let's now assume that one or more of the processes in the system are Java Virtual Machines running Java applications. Within one JVM we actually have a similar scenario. When an application wishes to draw something on the screen, it makes a call into the appropriate library (which will then contact the X server). When the JVM gets notified by the X server that some user input happened, an AWT event object is created which contains information about the event (for example, where a specific mouse click happened). This object is 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 Java has specifically been designed to execute possibly untrusted mobile code, great care has been taken to specify and implement a security model [3].

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 (which is a runtime exception) if the operation should not be allowed, effectively aborting the execution of that operation before any harm can be done.

Consider for example that some Java application executes 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 set the security manager and implement whatever security policy it likes (i.e. when the security manager should allow certain calls and when it should not).

However, the most common use of the security manager has turned out to be preventing applets (i.e., foreign code that is downloaded over the network and executed as part of a Java application) from gaining unlimited access to the system. In web browsers the security manager has been commonly written such that its main purpose is to find out whether a certain call is ultimately initiated by an applet (i.e. whether there is code anywhere in the call chain leading to the call of, say, `checkDelete()`, that was imported over the network). If the call was initiated by an applet, the operation is forbidden. Otherwise it is allowed.

In recent versions of the JDK and other Java platforms, this approach has evolved considerably [4]. The security manager no longer distinguishes between remote or local code, but follows in its decisions a policy that can be specified by the user in terms of the *code source*. In other words, depending on who signed the code and where the code came from, the user can specify which operations should be allowed for that code and which shouldn't. How exactly that policy is specified varies from system to system. As before, an application (such as HotJava) may get to install its own security manager, thus customizing security control.

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 make secure multi-processing. In this section, we discuss them individually, while in the next section we describe our approach to these issues.[3]

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.

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

In order to remain backward-compatible to work with existing applications, we want to execute the `main()` method of a class in order to start the 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, which is consistent with the current notion of a Java application, but allows multiple applications to run in one JVM.*

In UNIX, processes are allowed to do certain things depending on which user runs the process. If we have

[3] It is quite likely that a thorough thought experiment would reveal the same issues as our trial-and-error approach did, and the issues can be seen from different perspectives and this may lead to different solutions.

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 shouldn't (by default).

However, currently there is no explicit notion of a user running a Java application. Depending on the underlying O/S, the JVM may or may not have a meaningful user associated with it, which may affect the privileges of the JVM in the system. But that view is not propagated to the Java application. In fact, the view is sometimes partially hidden from the Java application. When running on top of UNIX, for example, a Java application cannot see files that the UNIX user who runs the JVM is not allowed to access, and an attempt to access those files results in a `FileNotFoundException` instead of a `SecurityException`. Hence, we need the following.

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 implements a (code source-based) security policy for the (only) one running application.

Another example is that certain threads that the runtime system creates on behalf of the user (e.g., the thread that communicates with the X server) are created in whatever thread group happens to be current when the need for them arises. However, as we will see in Section 5.4, this may conflict with the co-existence of multiple applications. Thus system code must be aware and can handle multiple applications appropriately.

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

In Section 3.2, we explained 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 that is associate 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.

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

To a Java application, the JVM acts like an operating system. Apart from providing system services, it holds certain system-wide state, including information about the operating system, a list of proxy hosts, and a list of principals known to the system. The JVM in turn makes such information available to the application as appropriate.

On the other hand, to the operating system the JVM is just another process, and duly holds process-wide state such as the running user-id, standard input, standard output, and error streams.

In current JVM implementations, the same process-wide state is mapped to any application that the JVM is running. For example, there is exactly one set of input, output, and error streams that all Java code in the JVM shares.

In a multi-processing JVM, while all the applications may share the same information about the underlying operating system or the list of available proxies, clearly different applications have different ideas about what the standard input and output streams are. In other words, application state and system state are two different things, and should not be mixed together, as is done in current JVM implementations.

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

One particular example of this desirable separation is case of 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 Java threads. A seemingly valid alternative approach would be to specify an application as a bounded piece of code, say, all the code that belongs to the classes that make up a given program. This latter approach was chosen for code source-based access control, and it does make sense there. However, an application may involve code from multiple of these sources, in particular, user code and system code. Intuitively, an application is started and run by a user, and is associated with this user during its whole life time, no matter which code is executed at any given time. Associating the application (and, hence, the user running that application) with specific code or classes would have been contradictory to this intuition.

Thus we conclude that threads provide a natural ground for the notion of an application. By the same token, threads give us a convenient way to distinguish two instances of the same program. running inside a single JVM (see Figure 3).

Furthermore, an application has the following properties:

- It has a lifetime, i.e. it starts execution at some point, and finishes execution at a later point.
- It is memory-protected from other applications. Java uses software-based protection that relies on the type system to provide basic memory protection and create (by means of class loaders) different name spaces for each application.
- It is associated with a user that is running the application.
- It holds application-wide state that is shared among all the threads that comprise the application. This state is likely different from that of another application. The state includes:
  - The aforementioned user identification.
  - Distinct standard input, standard output, and error streams.
  - A current working directory.
  - A set of properties.
- When an application creates a child application, the current application-wide state of the parent is inherited by the child.

To implement this concept of an application in Java, we created a class `Application` based on these criteria and implemented methods to start and stop applications, to query the running user and current directory, query and set the standard streams, and so on.

This 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 standard streams and running user. 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.

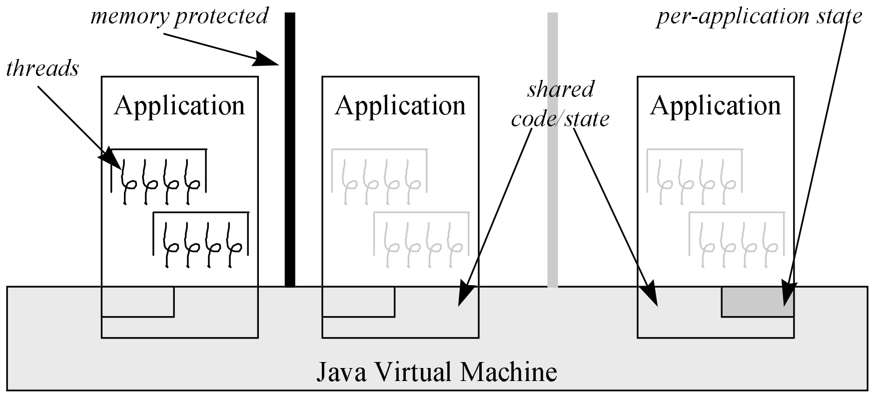

Figure 3: Applications. An application consists of a set of threads.

```
// 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 in Java. 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 the standard input and output streams must not be closed by the application. For example, if a shell chooses to launch an application with its standard streams redirected, and it creates streams for that purpose, then it is the shell's responsibility to close those streams after the application finishes.

[4] This phenomenon is in part 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.

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, login-in now works similar to UNIX's login program. 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 doesn't 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 "superuser". 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 that runs 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 that 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 [4], and introduced a new kind of *user permission* and then granting that permission to local applications.

When making access control decisions, if the JVM comes across code that has been granted this special permission, the JVM check 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.

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 to access files belonging to the user running the web browser. 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 was 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, i.e., code that is executed as the result of user input is executed by a thread that does not belong to any particular application.[5] For example, if Alice run a text editor and choose 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 4).

Note that, in current JVM implementations, some system threads, for example, the thread that communicates with the X server, are started in the JVM "on demand" – e.g., when a Java application first tries to open a window.

For our multi-processing needs, we cannot allow these threads to be randomly associated with whatever application that happens to be the first to use a graphical user interface. Therefore, we changed the runtime system so that these threads are created in a special system thread group, which does not belong to any application. This is the same thread group that also contains the garbage collector and idle thread.

The per-application event dispatcher threads, on the other hand, are created on demand. Whenever an application first opens a window, we create an event dispatcher thread for this 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.

5.5 Reloading System Classes

As we explained before, a Java application used to be a full-fledged process of the underlying operating system thus, quite naturally, some of the process-wide state including standard streams is in fact system-global state in Java. 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 provide a solution that also maintains backward compatibility for existing application code. We provide each application with the illusion that it has the JVM all for itself. To be more specific, we borrow a

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

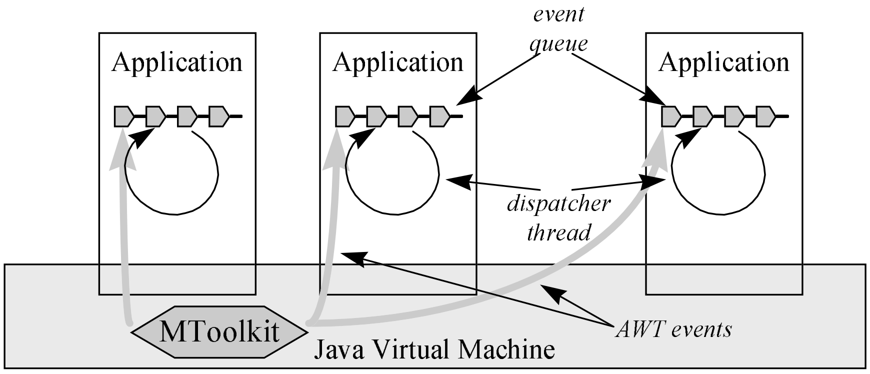

Figure 4: Multi-threaded event dispatching. Every application has its own event queue and a thread in the application's thread group delivers the events.

technique that has previously been employed to provide differently trusted code with a different view of what the system classes are [12].

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. This is similar to the case where two applets in a browser may happen to use different classes with the same name. Because the same class are loaded by different class loaders, the underlying JVM treats them as different classes. 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 feature, we were able to easily implement input/output redirection and pipes between applications.

Our experiment so far has not determined whether there are more classes that need to be re-loaded like the `System` class. To reach such a conclusion, it is necessary to go through the entire JDK class library and find out which of the JVM-wide state truly is JVM-wide, and which state is actually per-application state. Once per-application state (like the standard streams) has been identified, the classes defining that state have to be added to the list of reloaded classes.

Note that the `System` class contains state in the form of the system properties 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. The API for accessing the system properties is unchanged, as applications still use `System.getProperties()` to access the system properties. Figure 5 illustrates the relationship of shared and reloaded system classes.

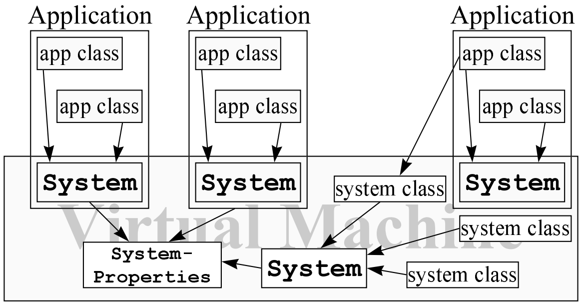

Figure 5: Reloading the `System` class. Every application, as well as the system itself, sees its own copy of the `System` class. Global properties are now held in a `SystemProperties` class that is shared between all applications.

5.6 Multiple Security Managers

We installed a security manager (the *system security manager*) in our multi-processing JVM that implements the following policy, primarily for the purpose of protecting applications from each other.

- A thread T may access another thread U if T's thread group is an ancestor of U's thread group. If this is not the case, T may only access U if it has the appropriate permission.
- A thread T may access a thread group G if T's thread group is an ancestor of G. If this is not the

case, T may only access G if it has the appropriate permission.

- Public members of a class can be accessed normally through the reflection API. Access to non-public members needs an appropriate permission and is controlled by the system security manager.

- For all other security-relevant decisions, the `AccessController`[6] is consulted, which effectively means that code needs to have the appropriate permission.

Note that, although applications can set their own security managers, but because we have a separate `System` class for every application and that is where the reference to the application's security manager is stored, applications in theory can still set their own security managers. However, those security managers will never be consulted by system code, because the system code that performs sensitive operations sees its own version of the `System` class that holds the system security manager.

It is non-trivial to make a system security manager work with application security managers. For example, today's security architectures prevent so-called *luring attacks* by making sure that even privileged system code cannot call into unprivileged code without losing its privileges. However, in our system, an application security manager is by default unprivileged code (if it were privileged, it could decide to turn off inter-application security). Therefore, most of the security checks will fail because the checker does not have enough privileges for the checks to succeed.

Consider, for example, an application that is not allowed to read files, but wishes to write text to the screen. In order to do that, the `Font` class needs to read in font characteristics from the file system. Since the `Font` class is trusted, it has enough privileges to read from the file system despite the fact that the application is not allowed to do so directly. However, as soon as the `Font` class calls into application code, like the application security manager, those privileges are lost, and file access will be – wrongly – denied.

It seems that application security managers in our architecture cannot be used to override behaviors of the system security manager and are best left to perform application specific security checks that are not covered by the system security manager. Moreover, applications can augment (or change) the effective system security manager behavior 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] The `AccessController` is the class that implements the code source-based access control in Sun's Java Development Kit.

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 rest of this section.

6.1 A Shell

As part of our prototype, we implemented a shell for executing Java applications. The shell executes an infinite loop in which it reads in a command line (provided by a terminal, see Section 6.2), interprets it, and possibly launches one or more applications as the result of its interpreting of the command line. A simple command (e.g. `ls`) will launch the application of that name, and wait for its completion before reading in the next command. A command followed by an ampersand (e.g. `hotjava &`) will start the application of that name (as a concurrent application) and immediately be ready to read in the next command.

The shell that we implemented uses pipes between applications and input/output redirection (with the syntax borrowed from UNIX). Normally, the input and output streams of the applications that the shell launches are not changed (and, hence, are the same as the shell's). However, in the case of pipes or input/output redirection, the shell temporarily changes its own standard input and output streams (to point to the appropriate pipe or file streams) before each application is launched. This causes the new application to have its input/output streams set to non-standard values. Afterwards, the shell's streams are re-set to their original values.

We equipped the shell with a few built-in commands such as `cd` and `quit`, and implemented utility applications including `ls` and `cat`.

6.2 A Terminal

The shell presented in the last section needs a way to communicate with the user. Usually, the Java system itself is run from a UNIX (or other) terminal, so the shell could just read from there.

However, there are a number of reasons for implementing an independent Java terminal. First, we might not always be in a environment that has terminals as part of their GUI. Even if we are, Java does not have much control over the terminals, other than reading and writing characters to them. For example, there is no standard way to turn off echoing of the underlying terminal (needed for password entry), or to provide functionality similar to the GNU readline library. It would be quite impossible to write text editors like `vi` in Java.

We implemented a simple prototypical terminal that has a few methods to read from and write to the terminal, and to switch echoing on and off.

Applications can make use of a terminal as follows: If only basic input/output is needed, then applications can just read and write to `System.in` and `System.out` (which are connected to the Java terminal, as inherited from the `Terminal` application itself).

If more control over the terminal is desired, applications can retrieve a reference to the terminal object itself. The shell, for example, uses the terminal's advanced `readString()` method when connected to a terminal, thus giving the user features like a history buffer. The login application uses the `turnEchoOff` method before asking for a password. Other applications like `cat` only use the standard streams, and therefore also work if they are not run from a terminal (such as when they are used in a pipe).

6.3 Porting the Appletviewer

As a final example, we moved the Appletviewer, which is a built-in program distributed with JDK and normally run as system code, to become an application as defined in our framework. More specifically, we moved the Appletviewer's classes off the system class path `CLASSPATH`, and this has the result that the classes are no longer automatically privileged. Also, we replaced all `System.exit()` calls with `Application.exit()`. This change will not be necessary if we change the semantics of `System.exit()` to only exit the current application.

A significant difference is that we no longer need the Appletviewer's security manager. Instead, the `AppletClassLoader` now implements the necessary methods to delegate permissions to the applets it loads, thus implementing the original Java sandbox security model. For example, an applet will get the permission from the Appletviewer to connect back to its own host.

Note that one can still assign special privileges to certain code sources (such as certain applets), in accordance with the new security model in JDK1.2. The underlying JVM do not distinguish between permissions granted by the Appletviewer and permissions granted by the user.

We successfully run multiple instances of the terminal, together with shells, the Appletviewer, and a number of applications 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 [9] used a safe language [6] in a single address space to provide security without a kernel. Various approaches of software-based protection has recently been reconsidered in the context of Java [12]. Our contribution is that our multi-processing Java environment must deal with distributed computing with 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 [1, 8]. As far as security is concerned, they use either a type-safe language [1] or provide other techniques such as proof carrying code [8] 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 [12]. In comparison, we focussed on the latter while not paying particular to performance tuning.

Recent advance in Java security has evolved from the original restricted sandbox model to a policy-based, easily configurable, fine-grained access control model [3, 4]. However, security policy is still limited to deal with code sources and not with users. This is because 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 was 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 at any one time. 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 relatively limited changes and additions we described in this paper, Java 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 object 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 [2].

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] 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.

[2] 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.

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

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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*, Saint-Malo, France, October 1997.

[13] A. Wollrath, J. Waldo, and R. Riggs. Java-Centric Distributed Computing. *IEEE Micro*, 17(3):44–53, May/June 1997.