

Robust Implementations of Java Technology for Internet Appliances and Embedded Devices

This paper will describe the aspects of a Java implementation that affect its robustness and hence its suitability for Internet appliances and embedded devices. It will explain the issues that affect the robustness of a Java implementation and how to look for them.

This is not a tutorial about Java technology or how to write Java applications. Its purpose is to explain how the considerable benefits of Java technology can be realized for developers of Internet appliances and embedded devices but only using suitable implementations.

INTRODUCTION

This paper will describe the aspects of a Java implementation that affect its robustness and hence its suitability for Internet appliances and embedded devices. It will explain the issues that affect the robustness of a Java implementation and how to look for them.

This is not a tutorial about Java technology or how to write Java applications. Its purpose is to explain how the considerable benefits of Java technology can be realized for developers of Internet appliances and embedded devices but only using suitable implementations. This is important because Java technology is still frequently thought of as an emerging technology and, in many people's minds, has yet to prove itself to be truly suitable for mass deployment in these kinds of devices. As we shall see by paying careful attention to the design and implementation of the Java technology that you select - in particular paying careful regard to its robustness - it is possible to deploy Java technology with confidence today.

We will first take a look at Java technology and review the reasons why it is attracting so much interest. Then we will briefly remind ourselves of the most important characteristics of internet appliances and embedded devices and, most importantly, remind ourselves of why they are so much different than the familiar desktop environments that most of us use daily at work and home. Next we shall consider the importance and meaning of robustness for these types of devices. With this in mind we shall then look at how Java technology can be deployed on these devices and the design and implementation issues that determine their robustness. Finally we shall summarize with a checklist of items to look for when evaluating Java technology for your device.

JAVA TECHNOLOGY

Java technology can be thought of in two main ways - as a programming language and as a platform for running applications, that is, a particular type of run-time system. To understand why Java technology is so important to anyone for whom robustness matters it is necessary first to appreciate the significance of both

these aspects. We will also review briefly how Java technology is actually implemented.

The Java language is a programming language that is frequently described as a better C++. In terms of syntax and semantics it is very similar to C++. However, it has several significant improvements over C++ (or C, for that matter) that have contributed significantly to the enthusiasm with which Java technology has been adopted.

For example, the Java language is a type safe language. Any operation you perform upon an object is guaranteed to be appropriate for the type of the object. You can't freely cast integers to be pointers (or references as they become in the Java language) or vice versa. Because you can't convert an arbitrary bit pattern into an object reference an application can only access data of the correct type within its own address space. The Java language has a rich set of exceptions built into it and the programmer can extend these. Most importantly, it is a requirement - enforced by the Java compiler - that any exception that might be thrown must have a handler written for it. Another major difference between the Java language and C++ is that in the Java language there is no "free" operation to release memory. This eliminates a common and very serious category of problems due to memory being released too early while it is still in use. It also eliminates the converse problem - which can be critical in a low memory environment - of failing to release memory once it has been used for the last time. The need for a "free" operation is eliminated by the presence of a garbage collector in the Java run-time.

One crucial fact about the Java language is that although Java programs are compiled in much the same way as C or C++ the standard Java compilers do not generate code that is either processor- or operating system-specific. Instead of generating machine instructions the compiler generates Java bytecodes that are a kind of generic machine code. This means that it is easy to deliver an application to any system supporting Java technology because there are no processor-specific dependencies. The price for this portability is that the run-time system has the responsibility to execute for loading and linking Java applica-

tions dynamically and then executing bytecode instructions on the target processor.

This brings us to the second view of Java technology - the Java platform as a run-time system. The Java platform presents a complete, self-contained programming environment. That is, it provides a full set of services and functions that enable the programmer to develop applications without requiring any knowledge of the operating system or processor that the application will eventually run on. It goes well beyond what you might be familiar with C or C++ in providing a comprehensive set of facilities including file I/O, networking and a graphical user interface. The benefit to the application developer is that it is possible to write powerful, sophisticated applications without making any calls to the operating system or supporting libraries.

This combination of processor and operating system independence enables Java technology to offer the renowned "Write Once, Run Anywhere" benefit. This means that an application vendor can target a class of devices with a single program, and a single development and test cycle, regardless of the CPU and OS that any particular device uses. All that is required is that the device be Java technology -enabled.

IMPLEMENTATIONS OF JAVA TECHNOLOGY

Now let's review how Java is implemented.

A Java implementation comprises two main components. The first is the virtual machine that is best thought of as a virtual operating system since it really performs all the functions that you would expect an

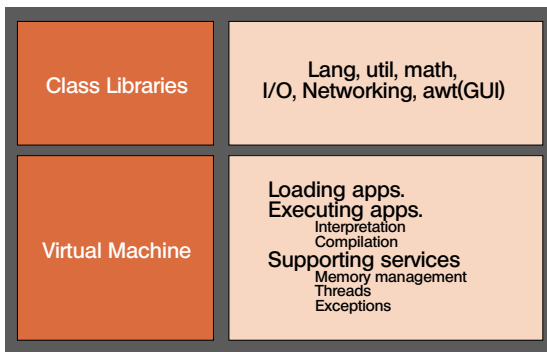


Figure 1. Java implementation overview.

operating system to provide. This gives Java technology its platform-independence. It must load the code that is to be run, execute the code, and provide all the supporting services. The virtual machine is complemented by an extensive set of class libraries. These are much richer than the standard C run-time library. The virtual machine will typically be written mostly in C or C++. Depending on the implementation there may be some assembly language as well. The class libraries are a mixture of Java and C. Both the virtual machine and the libraries will typically make extensive use of the underlying operating system.

An obvious question to ask at this point is why virtual

vendors choose to use C rather than the Java language as their implementation language. Or, as I have heard it put, why don't they eat their own dog food? There are several reasons for this. First is the bootstrapping problem of how to write a program in the language that you are implementing. Certainly when we began our own virtual machine development there was little choice. For example, there were no static Java compilers that could generate native machine code. Another reason is that virtual machines need to perform low-level memory and device access operations for which Java technology is not very convenient. Also, there can be a significant performance penalty for doing this. Today I do not find any of these reasons particularly convincing and our experience has shown that the Java language really is a much better language than C or C++. In fact, in our virtual machine development we are reworking components to use the Java language rather than C and I believe that other VM vendors may be doing the same.

Here is a simple architectural view of how things fit together. Underneath everything, of course, is the hardware device. Just above that is the operating system

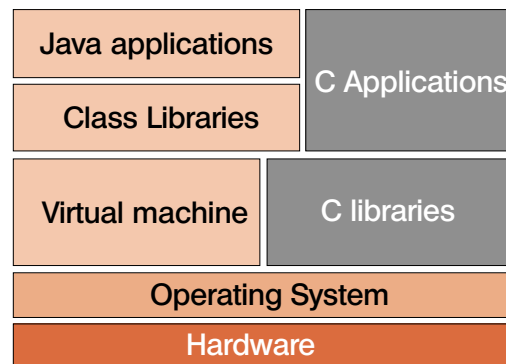


Figure 2. Embedded systems overview.

including the drivers that communicate between the hardware and software. The Java implementation comprises the virtual machine and class libraries.

A most important fact to bear in mind is that even when Java technology is being used on these devices it is most unlikely that all the code that is being run on the system is written in the Java language. This is for variety of reasons including the presence of legacy code and the fact that the Java language is not (currently) generally suitable for code that requires low-level hardware access or is timing-critical. Therefore the Java virtual machine, and the Java code it is running, must co-exist with other non-Java code in the system some of which may be performing functions that are critical to the overall operation of the system. We will discuss the implications of this later.

JAVA TECHNOLOGY FOR INTERNET APPLIANCES AND EMBEDDED DEVICES

Let us now turn to the types of devices of interest to us.



Figure 3. Types of devices.

Here are the types of the devices that my company has been working to deploy Java technology on. I'm sure you would find similar examples from other vendors. It is instructive to examine some examples of these devices and consider why the robustness of Java implementations running on them is so important.

The simplest use of Java technology that most people are familiar with is the running of Java applets inside a web browser. As anyone who has tried accessing the Internet from a phone or even a set-top box will know, surfing the web from an Internet appliance or embedded device is similar but not the same as running Explorer or Communicator on our PC or workstation. The most striking difference is the more limited screen displays available but this is really just one aspect of a more profound problem, namely the lack of resources on the device. In particular, the lack of memory poses considerable challenges.

Just as special web browsers often have to be written for Internet appliances and embedded devices, Java implementations also require special design considerations. It is far from certain that even if a VM is small enough to fit into the available ROM and RAM it will actually be able to run many applications. It is very likely that the VM will frequently run out of memory. Can it do so gracefully? Can it recover memory in order to continue running the application? If it can't continue, does it handle the problem gracefully or collapse in a heap? A particular concern is that as applets come and go, does the VM release the resources that were acquired for an applet or does it gradually accumulate more and more resources until eventually it - or, even worse, the underlying operating system - runs out. If the worst happens, can the VM be shut down and all its resources be reclaimed?

The second example concerns the cost - literally, in economic terms - of the failure to be robust. Let us consider a low cost, large volume consumer product such as a cell phone. Here cost and time to market are paramount considerations. These products are ultra price sensitive. In fact, cell phone manufacturers agonize over ever penny in the bill of materials. (In passing let us observe that this has profound implications for how much memory the VM requires, but that is a topic for another day.) Time to market is also vital - these products can have a shelf life of only a few months before they are superseded. Phone manufacturers are

now seriously starting to deploy Java technology in their products. What does this mean for the VM vendor? A lot of things, in fact, but, in particular, consider the cost of one critical bug in the VM. At best, it means the loss of functionality in the phone that diminishes its effectiveness and competitiveness. At worst, it means the phone becomes unusable and hence unsaleable.

The third example comes from a particular disk manufacturer that is building a set of storage devices that will be networked. Unlike the disk drives we are familiar with today, these are not sealed units because the manufacturer intends for third parties to develop software to run on these devices. This is clearly a very attractive way of adding value to a product and is a pattern that we are likely to see followed increasingly as Storage Area Networks (SANs) become more common. The problem is that users have certain expectations about their disk drives that are centered on the notion that a disk drive will preserve the integrity of the user's data. The challenge is how to ensure this once the lid is off, so to speak.

As it happens, in this particular case the problem is even worse because, for reasons that need not detain us here, the operating system that the manufacturer is using has no memory protection in it. In other words, all code - both applications and operating system - are operating in the same address space. From this it is immediately apparent that third party applications written in C or C++ could wreak havoc in the system - whether malicious or simply the inadvertent consequence of a programming error. Therefore, the manufacturer has concluded that the only safe programming language for applications to use is the Java language. The reliability of the disk drive has now shifted to the responsibility of the virtual machine that must run those applications.

	Desktop	Embedded
Processor	Fast	Slow
Memory	Unlimited!	Limited
Predictability	"Nice to have"	Vital
Robustness	Frequent crashes	Critical

Figure 4. Contrast with desktop.

CONTRAST WITH DESKTOP AND SERVER SYSTEMS

In summary, Internet appliances and embedded devices present many different realities to desktop computers or even servers. Many users are familiar with the experience of periodically having their PC hang or crash. For Internet appliances and embedded devices this is completely unacceptable, especially for mission critical applications. This poses much more stringent demands upon the software they are running. In addition, embedded devices are far more limited in terms of their resources, both memory and raw processing power. Consequently, software products that may perform reasonably within acceptable bounds of reliability on the desktop may be unusable in embedded devices.

FEATURES THAT MATTER FOR INTERNET AND EMBEDDED DEVICES

What are the features of a Java implementation that matter to builders of Internet appliances and embedded devices?

In the last few years I have given many presentations to builders of Internet appliances and embedded devices about Java technology and the benefits that it can bring to their products. Of course different customers have different priorities in terms of the features that matter to them and so I frequently present them with a list of issues and invite them to prioritize their importance.

- Compatibility
- Size
- Java application performance
- Responsiveness
- Tools
- Portability

In fact, although their answers are extremely informative, this is partly a trick question because I deliberately omit one feature. Interestingly, more often than not the audience actually fails to volunteer that extra feature.

The feature is robustness, or reliability. When I point out to an audience that they have omitted to mention this a typical response is to say that it is so important it is taken for granted. Without robustness they do not have a viable product.

ROBUSTNESS

ro.bust

adj [L robustus oaken, strong, fr. robor-, robur oak, strength] (1549) 1 a: having or exhibiting strength or vigorous health b: having or showing vigor, strength, or firmness <a ~ debate> <a ~ faith> c: strongly formed or constructed: sturdy <a ~ plastic> ...

Above is part of a dictionary entry for "robust". It has several meanings but the key issue for us is captured in the phrase "strongly formed or constructed: sturdy". Here is a definition that one of our engineers proposed:

Said of a system that has demonstrated an ability to recover gracefully from the whole range of exceptional inputs and situations in a given environment.

One step below bulletproof. Carries the additional connotation of elegance in addition to just careful attention to detail. Compare smart, opposite: brittle.

ROBUSTNESS REQUIREMENTS

Some of the attributes of a robust system will be that it:

- Works correctly
- Copes under stress
- Continues to be usable under all conditions

Taking this as our working definition we can now look at how each of these attributes can be addressed.

Requirements (1) - a system that works correctly

A necessary, but not sufficient, requirement for a robust system is that it functions correctly. For Java technology, there is a very precise sense in which this can be quantified. Any Java implementation should implement one of the standard Java specifications fully and completely. There is a range of specifications tailored to different types of devices. It is clear that the fundamental Write Once, Run Anywhere promise of Java technology can hardly be delivered if different VM vendors take different views about what aspects of the Java specifications they choose to implement on the same devices. For customers the only reliable guarantee of complete compatibility is that the vendor has passed the certification test suites produced by Sun and the Java Community Process. This is a significant undertaking. Therefore, to meet the first requirement for a robust implementation of Java technology the VM and libraries must be certified which means the VM vendor must be a Sun licensee. Note that this does not mean that the vendor is simply porting Sun's own VM source code, only that the VM implementation is fully Java standards compliant. Several companies have independently-developed VM technologies but are also Sun licensees.

Requirements (2) - a system that copes under stress

However, a VM that is functionally correct is not necessarily usable. The second requirement for robustness relates to the fact that, in Internet appliances and embedded devices, running out of resources is not simply a hypothetical possibility but a constant reality. This imposes a stress on the VM. Is the VM's use of resources limited? Can the VM itself be matched to a particular set of resource constraints? Does the VM incrementally consume resources over time or does it release them in an efficient manner? If a catastrophic error occurs can all the resources in use be reclaimed or are they lost forever?

IS THE VM SMALL ENOUGH?

In many respects, the single most important resource is memory. It is easy to understand how implementations that were designed initially with desktop or server systems in mind don't necessarily scale down well. Typically there are two types of difficulty. The first is that a programmer who has perhaps 128 MB or more of physical RAM backed up by several gigabytes of virtual memory may not be overly concerned about making his code memory efficient. The second is that the programmer may not be overly concerned about

checking that every time he asks the system for a resource it actually gives him one.

A further complication comes from understanding the differences between ROM and RAM. Although it is not the case universally, it seems a pretty good rule of thumb for a VM implementer to proceed on the assumption that RAM is much more scarce than ROM on a device. An example of an implementation strategy that conflicts with this is to compress the standard Java classes in ROM. This achieves impressive looking (for Java technology, at least) static code sizes but it means that large amounts of RAM are needed to uncompress those classes and execute them. This brings us to an important point about VM memory usage specifications, make sure to check or inquire how much memory is required not just for static storage of the VM and its related (class file) components but also to run real applications.

In general, therefore, it is the overall, dynamic resource usage of a VM implementation that is most important. The key questions are, first, whether - for each type of resource - the usage of that resource can be bounded and, secondly, whether usage of the resource simply increases over time or it can be reclaimed in an efficient manner. For the Java heap there is typically a command line option to specify the size of the heap. Running out of heap will first lead to the garbage collector running and ultimately, if that is unsuccessful, to a Java out of memory exception being thrown. The virtue of the Java exception mechanism is that it allows to applications to handle and recover from these types of errors. Two important points to note are that Java memory will only be reclaimed effectively if the garbage collector is both exact and compacting. Exact means that it computes precise information about what memory is still in use and not just a conservative approximation to it that can lead to dead memory being erroneously retained. Compacting refers to the ability of the garbage collector to defragment the Java heap. Otherwise, an all too real danger is that although sufficient free memory exists in the heap it will eventually become so fragmented that it becomes impossible create new objects from it.

In a similar way, Java technology provides a mechanism for handling Java stack overflows. To make more efficient use of the available memory it can be useful to break the Java stacks into fragments. This means that individual threads can have stacks of different sizes rather than all threads having a stack of the largest size required by any of them.

System memory is the memory used internally by the VM, for example to perform class loading. In a VM running inside a web browser the constant loading and unloading of Java applets can impose a considerable strain on this. The VM must be able to recognize and recover from running out of system memory. This is a more significant challenge for the VM implementer particularly if the class loader is written in C and manipulates C data structures in system memory. One solution mentioned earlier is to take advantage of Java's virtues and write this code in the Java language and use Java data structures. An intermediate solution is to

add structured exception handling to the source code. In particular, a combination of resource tracking, which logs where all resources are allocated, and "finally" clauses, which allow tidying-up to be performed when an exceptional condition occurs, is very effective. Since it is very hard to defragment a bunch of C data structures with random C pointers linking between them, the most attractive way to avoid fragmentation of the system memory area is to use Java objects instead wherever possible.

Overall, the total amount of memory that the VM will be use must be boundable, and should be configurable by the system builder.

Memory is, of course, just one type of resource that the VM typically calls upon the underlying operating system to provide. Because the Java language is a multi-threaded language threads are another important resource. When running on an operating system that supports threads it is natural to map each Java thread onto a thread of the underlying operating system. Although this can be beneficial in many circumstances there is an associated resource problem. Each OS thread carries with it some memory including its own stack. Since a Java application might cycle endlessly creating more threads, there is no way in which the total memory and thread usage of the underlying OS can be bounded. The alternative is to manage all the Java threads internally inside the VM using a single OS thread. (Actually it may be that the VM uses several threads but the key property is that the number is very small and bounded.) In conjunction with offering a solution to use discontinuous stacks mentioned above this VM implementation technique provides a very resource-efficient way of supporting large numbers of Java threads.

DOES THE VM LEAK RESOURCES?

In addition to bounding resource usage, it is equally important that resources can be reclaimed. Monotonically increasing resource usage will lead to inevitable disaster. For each category of resource the VM must have a strategy for recognizing when a resource is no longer used and then releasing it.

There are other kinds of resources in use, for example file handles and graphical resources, and other consumers of memory in a Java implementation. One in particular that is worth mentioning is the dynamic compiler such as a Just-In-Time (JIT) compiler. We shall return to this shortly.

What happens when a catastrophic error occurs?

Another consequence of stressing a piece of software is that various kinds of problems will occur. Some will have been anticipated with specific handlers written to deal with them. Nevertheless there always remains the possibility that some kind of unforeseen error will occur. A robust VM must have strategies for expecting the unexpected as well. Internal monitoring or policing of the VM can be used to recognize when a variety of serious, uncaught errors have occurred. One example is adding run-time checks for deadlocks as mutexes

are requested. Another is to turn the asserts that wise programmers add to their development code to validate that their code is operating as expected into run-time checks and exceptions in the production VM. Both of these strategies imply a trade-off between robustness, in the form of extra run-time checks, against performance. Only the individual customer can determine the appropriate balance for their particular product so these elements must be made configurable at build time or run time.

The VM also needs a way of recovering when external fatal errors occur or, equally, of allowing the rest of the system to recover if the VM starts behaving badly, for example locking up. When all else fails it must be possible for the VM to shutdown, to release all the system resources it has allocated, and to be restarted. Many operating systems will manage the freeing of system resources automatically when a process ends but some leading real time operating systems are notable exceptions.

Requirements (3) - a system that continues to be usable under all conditions

The third requirement for a robust VM is that the system not just be standards compliant, and use resources efficiently and be small enough, but actually be usable. We can subdivide this into three issues. Does the VM run applications fast enough especially under increasing loads? Is the VM responsive - does the Java application respond in a timely manner to events such as the user pressing a key? And does the VM provide good quality of service to the system as a whole - does it help or hinder the non-Java technology rest of the system from performing the actions for which it is intended?

DOES THE VM PROVIDE ADEQUATE PERFORMANCE?

Java application performance may not be thought of as a robustness issue but, as we have already discussed, a VM that provides poor application performance is essentially not usable for Internet appliances and embedded devices. For this reason it is desirable to include some form of acceleration within the VM. Compilation technology is the typical method of providing Java application acceleration. The typical compilation technique is an on-the-fly compiler, such as Just-In-Time (JIT) compilers, which dynamically translates Java application code into native microprocessor instructions that can then be executed at native microprocessor speeds rather than running through a VM interpreter. In general this is a critical technology for Internet appliances and embedded devices simply because the lower horsepower of the microprocessor means that for Java applications to be usable every bit of acceleration helps a great deal.

Unfortunately, conventional implementations of JIT compilers suffer from being too memory-hungry to operate successfully in low memory environments. JITs tend to introduce Java application startup delays as well. However with careful design it is in fact possi-

ble to implement a dynamic compiler that can operate very successfully in a small memory footprint. A compiler needs memory both to do compilations and to store the compiled code that it produces. It must therefore be able to cope with running out of memory in either situation. Naturally, refusing to compile anything because of insufficient memory might be technically valid but hardly of value. In fact a compiler can be built to use less than 100KB RAM and still deliver significant performance improvements for real applications.

Note that other compilation techniques such as compiling off-line or at load time predetermine the amount of memory that will be required to store compiled code, and again these techniques introduce other often undesirable effects such as startup delays. Either over- or under-estimating the memory that will be available will lead to sub-optimal use of resources. Also, it is not possible to adjust the amount of memory used by this type of compiler dynamically. In fact this is a highly desirable feature. There will often be situations where the VM is running out of memory and it is better, for example, to delete compiled code and switch to running via interpretation rather than raise a Java out of memory exception.

IS THE VM RESPONSIVE ENOUGH?

Many Internet appliances and embedded devices require the system to be responsive in the sense that it can respond to events in a timely manner. The types of events to be handled vary widely. At one extreme the hard real-time community, using software to control the engine in your car or industrial automation, require guarantees that events can be responded to within specified and deterministic time limits. This is because the consequence of failure to provide a timely response can be critical. The demands that this places upon the virtual machine are considerable and have led to the development of real-time specifications for Java technology that extend the Java semantics to provide hard real-time support. However, since this is not part of the standard Java specification it won't be covered further here. If you want further information go visit Sun's web site.

There are many situations where the failure to handle events promptly is not fatal but can seriously affect the usability of the system. This applies to any type of user interface. There are few things more frustrating than pressing a key or clicking a mouse button without an immediate response from the device. The ability of the VM to ensure that a Java application is responsive depends primarily on its ability to allow the high priority threads responsible for the event handling to be scheduled to run as quickly as possible after an event occurs. If the Java threads are being mapped onto host OS threads then the actual scheduling is the responsibility of the OS. Since the OS is typically a real-time operating system it can schedule a high priority thread promptly providing the thread is actually schedulable at this point. If the VM is scheduling the threads itself then it must be capable of scheduling high priority threads promptly when they need to be run. Either way, ensuring that an event-handling thread will actually be schedulable when required is not

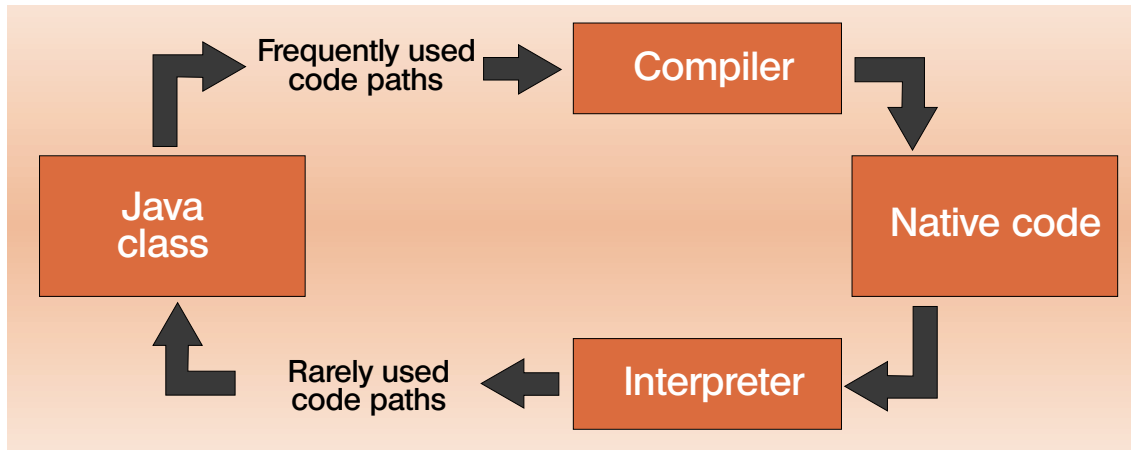


Figure 5.

always easy for a VM. In fact two key VM components, the garbage collector and the dynamic compiler, can easily block these threads from running unless they are designed and implemented carefully.

For garbage collectors, the problem is that a garbage collector needs to examine a thread's data in order to determine which Java objects are dead and live. The easy way to do this is to pause the application until the garbage collector has finished but this leads to long, unbounded pauses that eliminate any possibility of responsiveness. A common alternative is to break the one long pause into a sequence of smaller pauses in which the garbage collector runs incrementally. However, if an event occurs during one of these pauses there is still a danger that it will be responded to late or even lost completely. A third and more satisfactory possibility is to implement the collector as a normal thread running at a normal priority. Whenever a high priority event handling thread needs to run it can be run as quickly as the thread scheduler can switch threads. With careful handling of the case where the garbage collector is actually analyzing the event handling thread that needs to be run it is possible to provide very efficient responsiveness.

For dynamic compilers the problem is that compiling a thread's code while it is running is normally done by pausing the thread to wait for the compiler to finish. Compiling the thread early, at build time or load time, is a possibility but creates further problems. Compiling at build time can easily lead to a bloat in size and in any case is not possible for dynamically loaded code (one of the main attractions of Java technology). Compiling at load time can also use excessive memory in addition to significant startup delays. An alternative is to implement the compiler also as a thread that runs asynchronously with the thread whose code it is compiling. If at any point during the compilation the thread needs to run then it can be scheduled immediately because the compiler has not blocked it and the interpreter can execute the code.

DOES THE VM PROVIDE GOOD QUALITY OF SERVICE?

The final aspect of usability, our third robustness requirement, concerns the integrity and usability of the

device as a whole and how the VM can affect this. First the loading of unknown code from a possibly unknown source raises all kinds of security concerns. Fortunately, as we discussed here, many of these are addressed by the tight semantics of the Java language which make it illegal for a valid Java program to do the things that a virus writer would like to do, such as reading and writing data at random addresses. It is therefore essential to ensure that the Java code that is loaded is indeed a valid program. To this end as part of the process of loading Java classes a verifier is run to ensure strict conformance to the Java semantics.

It is quite possible for one application to prevent another application from running effectively simply by using too many of the resources, including CPU cycles, that the second application needs. A robust system must prevent this denial of service. One way to accomplish this is to bound how much of any type of resource can be used as we discussed previously. At present, the Java language does not provide standard mechanisms to do this at the level of an application, class or thread but the VM vendor can provide this type of configurability in the virtual machine.

To prevent problems in one application, class or thread fatally impacting another one proposed solution is to run each different app in a different instance of the virtual machine. Running each app in a separate OS process minimizes their potential to interfere with each other. Unfortunately, not all operating systems support processes, and in limited memory environments the overhead of running multiple virtual machines can be prohibitive. An alternative with similar benefits is to implement a multi-session virtual machine that provides the capability to run multiple Java applications, suitably isolated from each other, within a single virtual machine.

Last, the running of the VM must not have an adverse impact on the operation of the rest of the system that will often have timing-critical activities to perform. To prevent denial of service the resource usage of the VM as a whole must also be capable of being bounded as we have discussed before.

SUMMARY

Java technology has many powerful benefits for devel-

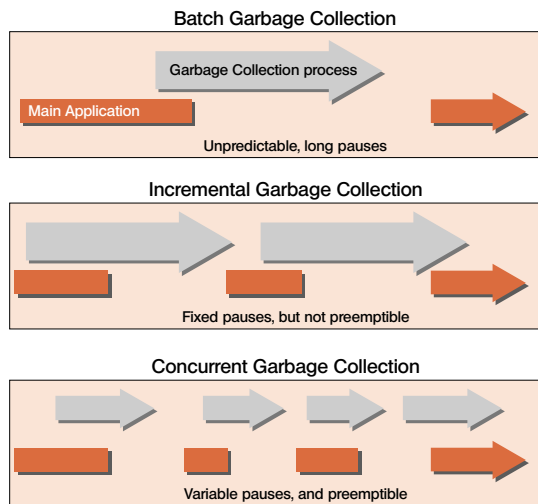


Figure 6.

opers of Internet appliances and embedded devices. However, these benefits can only be realized if the Java implementation is robust. We have identified a number of criteria for an implementation to be robust and discussed how through careful design and implementation a virtual machine can be built to satisfy these requirements:

- Works correctly
 - Certified
- Copes under stress
 - Bounded resource usage
 - Small memory footprint

- Resources can be reclaimed
- Continues to be usable under all conditions
 - Fast
 - Restartable
 - Responsive

Although possibly no single virtual machine meets every single requirement today there are implementations that come close and which developers can start using with confidence. It should also be clear that while there is a standard Java virtual machine implementation specification, there is much room and opportunity for JVM vendors to innovate and adjust their offerings for specific markets and applications, which is particularly true for Internet appliances and embedded devices ■

Dr. Hoskin was appointed to his current position in May 1999, having been a member of the Insignia team since 1992. As engineering director of the company's virtual machine business, he was instrumental in the design, development and successful release of the company's Jeode platform. Prior to joining Insignia Solutions, he held positions at several leading-edge systems software companies in the United Kingdom. Dr. Hoskin earned a doctor of philosophy degree in mathematics from Wolfson College, Oxford University; an advanced degree in mathematics from Churchill College, Cambridge University; and a Bachelor of Arts degree First Class from the University of Warwick, Warwickshire, graduating with honors.



Find all the information you need on our website

The following topics are available:

- About Dedicated Systems Magazine
- About Dedicated Systems Gazette
- Editorial content of Dedicated Systems Magazine
- Editorial schedule 2001
- Index of all articles since 1Q93
- Subscribing to Dedicated Systems Magazine?
- Contributing to Dedicated Systems Magazine?
- Call for papers
- Advertising information

<http://www.dedicated-systems.com>