JAVA: A PARADIGM SHIFT IN TELEMETRY SOFTWARE ARCHITECTURES

Kent Yang, Senior Software Design Engineer
L-3 Communications Telemetry & Instrumentation
15378 Avenue of Science
San Diego, CA 92128

Phone: 858.674.5100, ext. 4180
e-mail: kent@ti.L-3Com.com

ABSTRACT

In the commercial marketplace, Java has emerged as the preeminent standard for platform-independent application development. Many reasons for this include platform independence, robustness, effective distribution models, security effectiveness, and a rich set of application programming interfaces (APIs). This paper will provide insight into the design of the Java platform as it relates to the development of telemetry systems. Specific elements of Java will be explored to better understand how to take advantage of a Java platform. The paper will conclude with examples showing Java's effectiveness on overall software development and deployment, the benefits of reduced new code implementations, and how deploying this unique software platform will change the software paradigm in the real-time environment.

KEY WORDS

Java, Software Architectures, Networking, Security.

INTRODUCTION

Implementing new commercial software standards such as Java into telemetry processing systems will have many lasting effects. As telemetry and space applications move rapidly toward the global marketplace, the need to provide robust and cost-effective software solutions for high-performance real-time acquisition and processing systems is vital. Using platform-independent software based on commercially available standards is one of the most effective ways of leveraging technology across multiple platforms.

This paper will provide insight into the Java platform architecture for telemetry and space system design. Specifically, the paper will examine five key areas of a Java platform: the language, the runtime environment, the security model, the networking API, and

performance. This paper will show that Java is much more than just another programming language. It will show that key features in the Java platform architecture will benefit overall software development and deployment by reducing new code implementations. By using this unique software platform, Java will change the software paradigm in the real-time environment.

LANGUAGE

The Java language is the core component of the platform that lays the groundwork for many platform-independent features. Java, as a language, offers much more consistency and robustness to the architecture. Java derives its heritage from C and C++, but the reasons why C and C++ are not successful in fulfilling the Internet needs of the new software paradigm is due in large part to multiple implementations on different platforms. Unlike C and C++, which is subject to operating system (OS) interpretations and multiple standards, Java solves the problem by standardizing language definition, syntax, and APIs across all platforms.

For instance, Java guarantees a standard size for all its primitive data types. This means developers no longer need to worry about whether an integer is 16 or 32 bits and whether it is signed or unsigned. Java only supports signed primitive data types. Thus, by standardizing features and syntax, Java ensures there will be no more ambiguities in regard to whether multiple inheritance, templates, and exception handling are supported. Additionally, the Java language provides an extensive and consistent set of standard APIs for handling graphical user interface (GUI) development, networking, and database access to name a few. With the standardization of these APIs across all platforms, developers only need to learn one set of APIs. In contrast to the Java language, using C and C++ means developers have to learn APIs for networking and GUI development specific to each platform. Standardization on the Java language means developers are no longer burdened with writing code to deal with platform differences. Instead, developers can focus on developing the application.

In an effort to make Java more robust, Sun made exception handling part of the Java language. Adding exception handling directly to the language ultimately made the application more robust because it forced developers to deal with error conditions. It also provided a clean way for developers to write error-handling code without confusing the logic flow of the code.

But the real benefit of exception handling is the opportunity for real-time applications to handle errors more gracefully than coming to a sudden halt of the runtime application. In the worst case, an operating system can crash from an error as trivial as not being able to write to a file or overstepping an array boundary. Handling exceptions is, of course, critical to the implementation of a telemetry or space application. At the very least, real-

time telemetry or space applications should trap for all exceptions at the application level to prevent catchable errors from crashing the system at runtime.

DIRECT MEMORY ACCESS

Another decision made by Sun to help the Java platform be more secure and robust was the removal of direct memory access. Essentially, the Java language simply removed the notion of a pointer and disallowed pointer arithmetic. All memory access is indirectly done through object handles. Sun discovered that supporting direct memory access by other programming languages often turns out to be the most problematic area and was a source for many bugs. In fact, a whole cottage software industry grew out of the need to address these problems. Software studies have been conducted, and 50-70% of all bugs can be attributed to pointer arithmetic and allocating and freeing memory. Even if we assume the more conservative 50% estimate, it is worth knowing that using Java to develop a telemetry application could reduce the number of bugs by as much as 50%. Telemetry applications also benefit from a security standpoint. By not supporting direct access to memory, Java prevents hackers and inadvertent users from snooping and corrupting memory. Unfortunately, other programming languages like C or C++ cannot make the same claim.

RUNTIME

At the heart of the Java platform's runtime architecture is the Java Virtual Machine (JVM). The JVM is a software component of the Java platform that serves as a runtime interpreter, a memory manager, and a security enforcer. Aspects of the security enforcer in the JVM will be discussed in the Security section. In this section, the focus will be on the runtime interpreter and the memory manager.

Java applications do not run directly on the user's machine. Instead, the JVM acts as the intermediary to translate bytecode instructions for a virtual processor to native platform-specific instructions. This feature of the Java platform gives Java its most powerful feature, runtime platform independence. A telemetry system software solution, using the Java platform, will be capable of running on over 90% of the machines in existence today, and more importantly, over the Internet. This is because Java is supported on many of the most popular platforms and Internet browsers today, and the list is growing.

The Java runtime/JVM also supports runtime memory management. It was mentioned in the last section that the Java language does not support direct memory access. The Java runtime function actually goes a step further in memory management. Part of the Java runtime function is a background thread called the Garbage Collector. The Garbage Collector thread keeps track of all the indirect memory allocations via object instantiations. It works in the background as a low priority task to automatically reclaim

memory resources that are no longer being used. This completely frees the software developer from having to deal with memory de-allocation because the Garbage Collector is doing all the cleanup work. Ultimately, this will help reduce the amount of memory leaks in the system.

SECURITY

Unlike other languages, Java is the only one designed to deal with security issues. The architects of the Java platform had the safety of the global community in mind when they defined their implementation of the platform. This ensured that the security features were tightly coupled to the development of the Java platform. If Java is capable of protecting millions of Internet users at large, it should be trusted to handle security in the real-time telemetry and space application environment. At the very least, it provides the best solution today.

Java handles both compile time and runtime protection; whereas most other languages only provide compile time protection. At compile time, the Java compiler checks to make sure that programs do not try to access arbitrary memory locations and that memory access is done through object handles. The Java compiler also validates that variables have been properly initialized before they are used. In the runtime environment, the JVM acts as an intermediary between the Java binaries (bytecode) and the native OS and verifies that the bytecode being loaded is free from viruses prior to execution.

Because of JVM's indirect execution of bytecode, it protects private system resources from illegal access. The security mechanisms are built directly into the language and Java platform. The core Java API will consult the Security Manager for permission to access system resources at runtime. The public interfaces of the Java API shield developers from knowing that the Security Manager is working behind the scenes. At runtime, the JVM's Security Manager component acts like a policeman and enforces the currently configured security policy. It prevents illegal access to memory, file system, or custom resources. For example, a security policy can be configured and enforced by the Java runtime system to protect users from inadvertently modifying a telemetry database. The security API can be used to extend protection to custom resources like a decom, a bit sync, or a ranging board. The security files are easily configurable text files that can be easily maintained by a system administrator. There is no need to modify or recompile source code just to change the security policy. Best of all, the Java platform provides the ability to set a security policy in the runtime environment that can be customized to each individual user. This is perfect for a telemetry or space system because often multiple levels of access are necessary not only for security reasons, but also to protect system integrity. For instance, in a telemetry system, a system administrator may be granted all access privileges; whereas a normal user may only be granted access to view telemetry

data. In this scenario, the administrator can do all the necessary setups in preparation for a mission and not have to worry about casual users accidentally reconfiguring the system.

NETWORKING

In today's telemetry and satellite markets, the overall system environment is becoming more and more heterogeneous. As telemetry and space markets continue to grow to support the global community, the main challenge becomes interconnecting these heterogeneous systems to the larger networking community. This includes local area networks (LANs) and wide area networks (WANs) as well as Intranet and Internet networks.

Heterogeneous solutions in the past involved designing complicated new protocols on top of TCP/IP, and because of platform differences in the representation of data, developers often have to deal with issues of rearranging the data (byte swapping). Even though Java provides a rich set of networking APIs that includes support for low-level network protocols like TCP/IP and higher level protocols like HTTP and FTP, the best solution for new application design is to use remote method invocation (RMI). RMI is similar to remote procedure calls (RPC) offered by other platforms. Both offer public networking interfaces that resemble making a local function call. In fact, the users of these APIs are precluded from having to know anything about how the data is packaged for network transfer. To the end user of the APIs, it is exactly the same as making a local function call. The difference between RMI and RPC, however, is that Java RMI is much easier to use, it is platform independent, and it supports object-oriented programming. RPC implementations, on the other hand, are platform specific and do not support objects. For example, RPC implementation on the PC Windows platform is different from the UNIX platform implementation. In order to connect the two systems without Java, the developer must resort to the method of designing a proprietary protocol on top of TCP/IP. Networking systems using Java RMI is a perfect fit for large telemetry and space applications because it makes a heterogeneous network of machines and systems appear homogeneous. Basically, RMI can be used to connect all the JVMs running on different machines and make them appear as one unified system.

Another benefit to the Java platform architecture that is network related is the new software distribution model. In the age of telecommunications, systems must be remotely monitored and controlled. From its inception, the Java platform has always provided a new software distribution model via applets. Applets are applications downloadable from the Internet that are capable of connecting back to the server for services. This software distribution model is perfect for the mobile application users of the telemetry system and for remote operations in space applications. The latest code can always be downloaded via the Intranet or Internet, eliminating the headache involved in installing and re-installing software. As soon as the latest software is available, it can be downloaded and

executed inside of a browser. This capability makes it possible to support telemetry and space systems worldwide.

PERFORMANCE

Probably the biggest criticism of Java by skeptics is performance. What critics of Java fail to realize is that Java applications do not always have to be exclusively executed in an interpretive environment. It is true that the Java binary file is not a native executable. In fact, Java binary files contain instructions (bytecode) for a virtual machine (software machine). In other words, another piece of software, the virtual machine (VM), is busily working behind the scenes interpreting the bytecode instructions as native instructions. However, this does not mean that a Java compiler cannot compile Java source into a native executable file that can be loaded by the native OS and executed like any other native executable. In fact, most integrated development environments (IDEs) for Java on the market today support the generation of native executables. This means that entire telemetry systems can be compiled for multiple target platforms into native applications, which essentially makes the performance issue a moot point.

Sun and independent third-party partners are working feverishly on trying to improve Java's runtime performance. Sun has come up with a runtime optimizing VM technology called HotSpot™, which profiles code at runtime and optimizes performance deficient areas (hot spots). Sun claims the performance will be equal to natively compiled C or C++ code.

Another very popular solution to address the performance problem is the Just in Time Compiler (JIT) compiler. A JIT compiler is essentially a virtual machine that will compile bytecode into native code at load time so that it can achieve better performance at runtime. Both solutions have their drawbacks. With the HotSpot technology, the profiler is yet another thread competing with the application for processor time. The JIT compiler's drawback is that it is slower to load. Neither one of these disadvantages outweighs the benefits of using the Java platform. Besides, as previously mentioned, one can always compile Java code to a native platform executable, which makes performance a non-issue.

Java Runtime Performance Benchmarks

Figure 1 shows benchmarks performed on four different Java runtime environments: Sun JDK VM, Sun HotSpot VM, Symantec JIT VM, and Native. The tests were conducted on a PC running Windows NT and consist of timing Java's disk input/output capabilities. The tests were executed 100 times, and the average is shown on the graph.

As expected, the best performance is when Java source code is compiled natively into a platform-specific executable and is then executed as a native application. The only drawback is that the binaries are no longer platform independent. Because real-time

performance is often critical to a telemetry or space application, compiling Java natively presents the best option. This does not mean that other components of a telemetry system, which are less dependent on performance and require more flexibility, cannot still rely on VM technologies. For instance, mobile or remote users should be able to download client code from the Internet and execute it using the browser's VM technology. Besides, the Java source code is still platform independent and can be compiled for other target platforms.

Not far behind in performance is Symantec's JIT™ VM technology. According to the data, there was only a 10-millisecond difference between the natively compiled Java application (EXE) and the Java bytecode executed using the JIT VM technology. Of the three VM technologies, the JIT VM presents the best solution in performance. But because it is a third-party VM implementation, it is not free like Sun's standard Java platform VM or Sun's HotSpot VM. Considering the overall costs of a telemetry system, the cost for JIT VM technology should be negligible. It should be noted that JIT technologies, in general, may not be supported on all platforms. However, there are numerous third-party vendors working hard to provide JIT solutions on all platforms.

Surprisingly, Sun's HotSpot VM technology did not perform well in the benchmarks. In fact, the tests show that the HotSpot VM technology has less than a 1 millisecond edge over the standard VM platform. This could be attributed to the length of the tests and the limitation in the scope of the test to disk I/O. Longer tests might have given the runtime profiler in HotSpot more time to identify problem areas. Also, to be more thorough, the tests should be expanded to include benchmarks on networking and graphics. Sun's HotSpot technology is still relatively new so it may take some time for us to see the real potential of this technology.

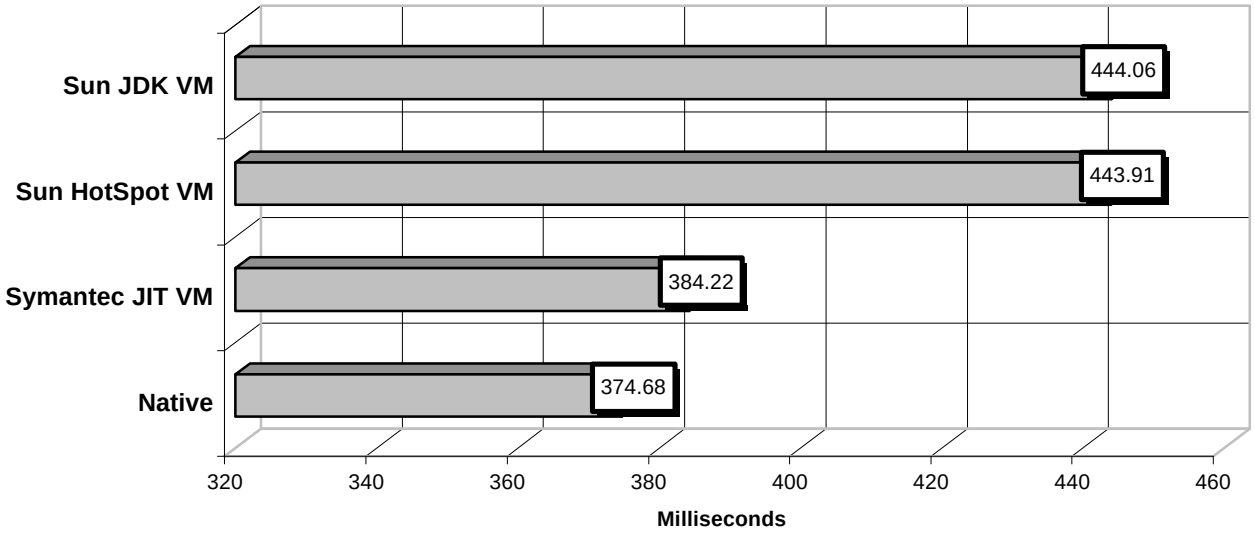

Figure 1. Java Runtime Benchmarks

CONCLUSION

Today's telemetry and space markets present an array of challenges. These challenges include supporting multiple platforms, interconnecting systems in a variety of networking environments, supporting legacy systems, and providing for ease of expansion. This paper showed that the Java platform successfully addresses all these issues and presents itself as the most viable platform for deploying telemetry and space solutions. Following is a summary of Java's benefits for telemetry and space applications.

1. Customers today are looking for ways to balance the cost and performance of a new system. As long as the system provides a path for upgrades, the design of a new system can have a mixture of platforms to match the needs of different users. For example, with the PC industry continuing to blaze a trail in price/performance, one possible solution might be to create a telemetry system that combines PCs and high-power workstations. The common system users can use PCs for real-time data monitoring, and the more sophisticated system developers can use the high-power UNIX workstations to do database development and real-time telemetry data processing. PCs tend to be more cost-effective and are more than capable of local processing at reduced data rates. What's more, PCs are much easier to use. Additionally, there might be requirements to support field engineers in remote locations. In this case, the telemetry software can take advantage of the new software distribution model. Java was specifically designed to support the new software distribution model, and this model is perfect for client applications that can be downloaded from the Internet and executed on PC laptops. The ability for a telemetry system to successfully support this mixed environment is due in large part to Java's platform-independent architecture at the language level and the Java runtime environment.

2. Using Java does not compromise performance. In fact, there are a variety of VM options to maximize performance in deploying real-time telemetry applications. The various options can also be mixed and matched according to system performance requirements and the needs of the user. For example, using the previously mentioned scenario, the server application code for real-time telemetry data processing can be compiled into a native application to optimize the application for runtime performance on the server. Yet, mobile users can still download client code in the Java platform's independent bytecode format from the Intranet or Internet. This code can be executed inside an interpretive runtime environment. Sun and third-party vendors are committed to continuing to improve the performance of the Java runtime environment. Meanwhile, system users can easily update their runtime environment to take advantage of emerging technologies as they become available.

3. Java is the only platform designed with security provisions. The Java security model is highly configurable and can be easily extended to support custom resources and custom security policies on a user-by-user basis. With the addition of exception handling and the removal of direct memory access, telemetry systems should experience more stability, and the likelihood of a system crash at runtime is minimized.

In conclusion, the real benefit of the Java platform is flexibility. A Java-based telemetry or space solution will provide the best value and extend the longevity of any telemetry or satellite system. Code can be written once and can be compiled for multiple target platforms without additional porting. Companies can then better focus engineering resources to improve the product and add features. Using Java also means companies can maintain one code base for their products, which will result in less development time, less testing time, higher quality software, and faster time to market. And so, with proper distributed application design techniques, high performance, scalable, and flexible systems can be achieved with Java.

ACKNOWLEDGEMENTS

A sincere thanks to Vivian Shelton of L-3 Communications for her encouragement and editorial support.

REFERENCES

Flanagan, David, Java Power Reference, 1st edition, Oreilly & Associates, Inc., Sebastopol, CA, March 1999.

Oaks, Scott, Java Security, 1st edition, Oreilly & Associates, Inc., Sebastopol, CA, May 1998.