Fault-Tolerance: Java's Missing Buzzword

Lorenzo Alvisi
Department of Computer Sciences
The University of Texas at Austin
Austin TX 78712

Abstract

Java has been described as a simple, object-oriented, distributed, interpreted, robust, secure, architectural neutral, portable, high-performance, multithreaded and dynamic language, prompting some to describe it as the first buzzword-compliant programming language. We submit that to deserve full certification—and in the process establish itself as the natural choice for developing large-scale distributed applications—Java misses a crucial buzzword: fault-tolerant. We outline some promising research directions for building reliable Java-based applications.

1 Introduction

Java may well be the most exciting technology of our time [2], but the excitement never appeared to leave its proponents speechless. Simple, object-oriented, distributed, interpreted, robust, secure, architectural neutral, portable, high-performance, multithreaded and dynamic [5]—these are some of the buzzwords that have been used to characterize Java since its first introduction. The vision of enterprise computing—a seamless integration of data and communication across thousands of machines to provide better services to citizens and greater opportunities to businesses—seemed to be at hand.

Remarkably, Java has substantially fulfilled many of the promises behind these buzzwords. Today, Java stands unchallenged in its ability to support true platform-independence and in its integration of security mechanisms. In addition, Java provides adequate support for distribution through Remote Method Invocation and run-time loading of classes. Performance is also becoming acceptable, thanks to the development of ever more sophisticated Just-in-Time compilers.

And yet, we submit that Java will fall short of what is required to realize the vision of enterprise computing until it explicitly addresses a conspicuous buzzword that it has so far overlooked: fault-tolerance.

2 Fault-Tolerance: the Ugly Duckling

Building distributed applications forces one to consider the possibility of partial failures. In fact, the kind of wide-area network applications that are likely to be programmed in Java are more vulnerable to partial failures than the relatively constrained applications that are common today. Furthermore, even as languages such as Java make it easier to develop distributed applications and to launch them on the Internet, they do not relieve programmers from the challenge of writing *correct* distributed algorithms. As distributed applications become common-place, we envision that fault-tolerance will become a pressing concern for many more application users and developers.

That Java makes no explicit provisions for fault-tolerance may be partly due to historical reasons: Java was originally conceived as a language for developing software for consumer electronics operating in an environment much more reliable than the Internet. Also, it probably does not help that fault-tolerance is hardly a source for sexy demos. Finally, in a marketplace that rewards the products that reach the market first, fault-tolerance is bound to be an afterthought at best.

There is no fundamental reason, however, that prevents Java from addressing fault-tolerance effectively. Indeed, Java's architecture provides an excellent opportunity to address the issue at the core of all fault-tolerance techniques: controlling nondeterminism.

2.1 Fault-Tolerance and Nondeterminism

If processes were deterministic, then tolerating failures would be trivial: a faulty process could be recovered simply by restarting it from its initial state and rolling it forward. In general, however, the state of a process depends on events that are non-deterministic. To recover a failed process, it is necessary to reproduce the non-deterministic choices that the process made before failing. For instance, in an asynchronous distributed system in which processes communicate by exchanging messages, the state of a process depends on the order in which a process delivers messages, which in turn depends on many factors, including process scheduling, routing, and flow control. Unless the order of message delivery is somehow recorded and reproduced during recovery, it will not in general be possible to restore a failed process to a state that is consistent both with the states of the other processes in the system and with the external environment.

Other examples of non-deterministic events that may affect a process state include preemptive scheduling of threads, readings of the processor clock, and responding to signals.

3 Fault-Tolerance in Java

There are two obvious levels at which non-deterministc events can be captured in Java:

1. At the virtual machine level
2. At the method invocation level

Capturing Nondeterminism in the JVM The idea of using a virtual machine to manage nondeterminism has been first explored by Bressoud and Schneider [1], who implemented a virtual machine for HP's PA-RISC architecture. In their scheme, fault-tolerance is achieved by replicating the computation on two independently failing processes, using a well-known technique called the state-machine approach [3]. To make this technique work, however, the two replicas must be deterministic. To ensure this, Bressoud and Schneider identified the non-deterministic commands processed by their virtual machine and designed protocols that guarantee that any non-deterministic choice is resolved identically at both replicas. For instance, their implementation guarantees that virtual-machine interrupts are delivered at the same point in the execution of both replicas, and that instructions that read the time-of-day clock return the same values for both replicas.

In Java, a virtual machine is already available for free. Typical implementations of the Java Virtual Machine (JVM), however, do not support deterministic replication of non-deterministic commands. Indeed, it is not obvious which non-deterministic commands are executed by the JVM, although it is reasonable to expect that many will occur at the Java Native Interface. One expects that once these commands have been identified, the techniques developed in [1] and [4] could be used to guarantee the reproducibility of non-deterministic choices during recovery.

Capturing Nondeterminism through Method Logging The logging of message ordering information described above can be generalized easily to distributed object computation systems. Instead of recording the messages delivered to a process, *method logging* protocols record the method invocations made upon an object. Not only do most of the message logging mechanisms apply to method logging, but method logging provides opportunities to improve upon these mechanisms.

Two of the main reasons that make capturing non-deterministic events a challenge are that these events are not easy to identify, and that they can be executed frequently, making it expensive to keep track of their effects.

The object-oriented context of method logging should help in capturing and limiting the effects of nondeterministic execution. For example, nondeterministic events could be encapsulated in method invocations that are tagged in their class definitions. The compiler could then use data flow analysis techniques to determine whether the value produced by a nondeterministic method invocation might affect the value of a parameter in a subsequent method invocation. If it does not, then the non-deterministic choice would not need to be recorded.

4 Conclusions

Java is in a uniquely positioned to emerge as the platform that will finally enable distributed computing to become, in the words of Ken Birman, a mass-market commodity. We believe that the key to Java's long-term success will depend on its ability to support the development of truly reliable applications, capable of tolerating both intentional security attacks and the less glamorous, but potentially as disruptive, spontaneous failure of subsets of their components.

References

[1] T. Bressoud and F.B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] Sun Microsystems Computer Company. Java computing home page. http://www.sun.com/java/, January 1998.

[3] Fred B. Schneider. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[4] J.H. Slye and E.N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing*, pages 250–259, June 1996.

[5] Sun Microsystems Computer Company. *Java Language Overview.* White paper available at ftp://ftp.javasoft.com/docs/papers/java-overview.ps.