Real-Time Java Commercial Product Assessment

October 2000

Alan Piszcz
Kent Vidrine

MITRE
Washington C3 Center
McLean, Virginia

Real-Time Java
Commercial Product Assessment
October 2000

Alan Piszcz, Kent Vidrine
The MITRE Corporation
McLean, VA 22102

e-mail: { apiszcz | kent }@mitre.org

ABSTRACT

Traditional real-time operating system vendors have recently started to consider Java as a potential platform in real-time operating system products and embedded solutions. Specification status and implementation towards an industry standard application-programming interface are split between two consortiums striving to introduce different paradigms of Java integration with real-time (RT) services. This paper provides a background and understanding of the direction of real-time Java in the commercial market place and includes status information about the specifications. Additionally a preliminary review of a few of the products with respect to timing jitter, and priority issues are examined.

1. INTRODUCTION

The Sun Java environment continues to grow in popularity throughout industry, academia and defense contractor organizations. Universities are transitioning language and programming coursework from PASCAL and C++ to Java. The Java environment's continued success, maturing and growing APIs point toward the platform of choice for software development. Originally developed to foster platform independent software, it is now being considered for unique hardware specific applications in the real-time application area. The Java platform must be modified and extended to support the requirements of traditional real-time developers. These requirements include fine grain control of system resources, memory, scheduling, I/O, event handling, interrupt services, and thread control. Balancing these requirements with the Java platform non-real-time features, dynamic class loading, extensive API support for non real-time features and garbage collection are some of the goals for real-time Java.

"The Java® platform – i.e., language, class library, virtual machine – promises "Write Once Run Anywhere," which, if even reasonably approximated, would substantially mitigate that problem. In addition, the Java language is widely believed to offer productivity advantages over C/C++ (being cleaner, simpler, safer, more dynamic and extensible, etc.)

However, Java was not intended for real-time programming, and the platform has fatal problems of omission and commission for most of that domain – including, but not limited to: unpredictable task execution times (e.g., complicated flow control, lack of analysis tools, JIT translation, method caching and other optimizations); unpredictable memory requirements (e.g., conservative garbage collection, failure to defragment, complex library services); unpredictable task interactions (e.g., poorly defined priority structure, uncontrolled priority inversion, non-incremental garbage collection); weak vocabulary (e.g., messages, semaphores, interrupts, I/O, signals, timeouts). Furthermore, Java distributed system mechanisms and platforms – RMI, Jini, JavaSpaces – suffer from additional disabilities for most of the real-time domain." [Jensen 1999]

Implementations of the above features will be used to verify the RTJ platform and its ability to satisfy real-time goals. Two approaches currently exist for real-time implementations using Java. The first approach relies on an existing real-time operating system for services and allows a Java Virtual Machine (JVM) to run as a task on a real-time operating system (RTOS). The second approach is based on the development of a RTOS JVM that will operate without an underlying RTOS. Device manufacturers are just beginning to consider single device solutions that include the JVM, RTOS and execution engine. Real-time issues were raised early [Nilsen1995] with respect to constraints and design inherent in the JVM. Two consortiums have been created in 1998 and 1999 to begin development and practical implementation guidance for the real-time Java platform.

The National Institute of Standards and Technology (NIST) workshop developed a set of principles for real-time core requirements. Greg Bollella and James Gosling [Bollella 2000] provide a side bar discussion describing the Real-time Specification for Java (RTSJ)

and the NIST requirements. The nine columns depicted in Table 1 maps the RTSJ into the NIST core requirements. The NIST core requirements are:

1 The specification must include a framework for the lookup and discovery of available profiles.
2. Any garbage collection that is pro-vided shall have a bounded preemption latency.
3. The specification must define the relationships among real-time Java threads at the same level of detail as is currently available in existing standards documents.
4. The specification must include APIs to allow communication and synchronization between Java and non-Java tasks.
5. The specification must include handling of both internal and external asynchronous events.
6. The specification must include some form of asynchronous thread termination.
7. The core must provide mechanisms for enforcing mutual exclusion without blocking.
8. The specification must provide a mechanism to allow code to query whether it is running under a real-time Java thread or a non-real-time Java thread.
9. The specification must define the relationships that exist between real-time Java and non-real-time Java threads.

Table 1 How RTSJ features satisfy the NIST core requirements

	NIST core requirements								
RTSJ features	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Scheduling	N/A		S					S	
Memory management	N/A	S	S						
Synchronization	N/A		S				S		S
Asynchronous event handling	N/A			S	S				
Asynchronous transfer of control	N/A								
Asynchronous thread termination	N/A		S			S			
Physical memory access	N/A								

In Table 1 the 'S' indicates that the RTSJ has satisfied the NIST core requirement. "As Table 1 shows, the RTSJ satisfies all but the first requirement, which is not relevant because the RTSJ does not include the notion of profiles. Access to physical memory is not part of the NIST requirements, but industry input led us to include this feature." [Bollella 2000]

2. REAL-TIME JAVA SPECIFICATIONS

There are two competing specifications that provide real-time services for the Java platform.

The Real-time Specification for Java [Bollella 2000] was produced by the Real-time for Java Experts Group under the auspices of the Java Community Process. It has the support of several large corporations, including International Business Machines (IBM) and Sun Microsystems. The specification has been published in book form and is publicly available in electronic form from [RTJ.ORG]. IBM is one of the first organizations with an offering in this space [IBM 2000]. More reference implementations are anticipated in 2001 and 2002.

The Real-time Core Extensions produced by the Real-Time Java Working Group [RTJWG], is currently in draft form. This specification effort has the support of Hewlett-Packard, Microsoft, and other corporations. There is no known reference implementation of this specification available.

Many commercial vendors do not consider the specifications complete until a compliance approval process is in place, this may include a reference implementation and requires a test suite. Given the incomplete status of the two aforementioned specifications, several commercial product vendors and universities have created their own implementations that do not conform to either specification. Several products were evaluated and are discussed in this paper.

3. OVERVIEW OF COMMERCIAL PRODUCTS

The current situation of the market place is in flux due to maturity of the specification, specification compliance testing ability and market demand or lack of it for real-time Java. Another key driver is the proliferation of soft real-time consumer products, these include digital cameras, set top boxes, and digital music systems. The current technology feature space is difficult to separate cleanly. However an attempt is made for products listed in Table 2 by grouping them into three classes.

RTOS JVM:
RTJ JVM executing on the hardware directly.

JVM on NON-JAVA RTOS:
Typically the JVM operates as a task on a RTOS or in some cases an operating system which has been modified to provide deterministic performance. An example is LINUX and its extensions by companies like TimeSys and MONTAVISTA.

DIRECT EXECUTION JVM RTOS DEVICES: Devices that claim a built in (silicon) RTOS kernel (Java or other) and JVM functionality.

The commercial vendors are positioning and integrating traditional strengths of existing product lines such RTOS features with Java strengths in network centric applications and graphical user interface tools.

Organization	Product
RTOS JVM	
ESMERTEC	Jbed, Jbed RTOS, JBED IDE
NewMonics	PERCH, PERC, ROMizer, QuickPERC
JVM on NON-JAVA RTOS	
Accelerated Technology	NUCLEUS Jvi
IBM	Embedded VM w/Realtime Ext.
INSIGNIA	Jeode
Hewlett Packard	Chai
LYNX	Lynx-OS
Mantha Software	KadaVM , KADA
Mentor Graphics Embedded	VRTX
Micro Digital Inc	SmxJVM
Microware Systems	PersonalJava for OS9
MONTAVISTA	Hard Hat LINUX with Kaffe JVM
QNX	Neutrino
Sun	CHORUS OS, J2ME
Tao Group	Intent Java Technology Edition
Timesys	Kaffe
Wind River Systems	Personal Jworks, VXWorks
DIRECT EXECUTION JVM RTOS DEVICES	
AJile	AJ-PC-104, aJ-100/101/102
IMSYS	cjip
Patriot Scientific Corporation	PSC1000

Table 2 Commercial Vendors/Products

Figure 1 [ajile2000] provides an overview of JVM RTOS options with respect to memory models, and interpreted versus direct execution. Other vendors such as IMSYS and Patriot Scientific Corporation listed in the Table 2 are developing products with similar capability as the fourth column in Figure 1.

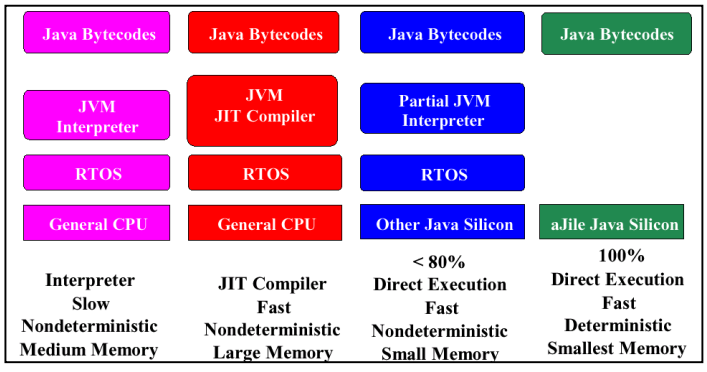

Figure 1 Execution alternatives for embedded Java

Several commercial products were evaluated. These were selected based on potential specification implementation, availability and cost to obtain an evaluation target and associated development environment. The three platforms are NewMonics PERC, ESMERTEC Jbed and AJILES AJ-PC104.

NewMonics – Perc

NewMonics, Inc. produces a toolkit called PERC®. PERC contains a clean-room implementation of the JVM and several development/deployment tools. According to the NewMonics web site (http://www.newmonics.com/), the PERC Virtual Machine (PVM) is a JVM implementation that provides features that are necessary for real-time performance. The main claims made by NewMonics about the PVM are that it provides better garbage collection and priority inversion avoidance than the standard JVMs that Sun Microsystems provides. Other claimed benefits include its small memory footprint and faster context switching time. This provides for behavior that is more predictable than the behavior of the same applications that execute in the context of a Sun JVM.

NewMonics states: “An application does not need to be modified to operate with PERC. Any standard Java application may run on either a PERC VM or a standard JVM. One can ostensibly obtain better performance using PVM than a standard JVM. To build an application that uses PERC, one simply writes standard Java code in the development environment of choice, compiles it with any Java compiler, and then executes the application using PVM. No additional/proprietary API is necessary (or provided).”

Esmertec – Jbed

Esmertec AG produces the Jbed product line (http://www.esmertec.com/). There are several Jbed versions. The various versions include an Integrated Development Environment (IDE), the Jbed RTOS, a proprietary Java API, and special-purpose boards based on ARM7, PowerPC, and 68xxx processors. The programmer must work with the Jbed IDE because it is not practical to write and compile with one tool and then switch to the Jbed IDE to create the binary file that the Jbed board needs to run the program. For this evaluation, Jbed Lite version 1.2.1 was used.

The API that Jbed uses to create real-time tasks does not resemble either of the real-time specifications. Jbed provides a re-implementation of the standard

java.lang.Math class that has a smaller memory footprint but does not offer all of the methods that the standard Math class provides. Existing applications that use the java.lang.Math class must be rewritten to use the Jbed com.jbed.FMath class.

The Jbed IDE provides tools for the programmer to write Java code, debug, stress test the system, build target executable files, manage projects, and monitor the Jbed system. With this IDE, the programmer can create either an open system or a closed system. An open System is one in which the executable file must be downloaded to the target board at board boot-time. A closed System is one in which the machine instructions must be permanently burned into the board's ROM.

Jbed uses the com.jbed.* API to create real-time tasks. There is very limited detail about this API in the documentation set and only a few source code examples. However, the abstractions that the API provides are simple and straightforward. Physical devices are either actuators or sensors. These devices are accessed via standard Jbed classes. The object that does real-time work is known as a Task, which is a subclass of the familiar Java Thread class. Jbed provides the following types of real-time tasks:

- OneshotTimer is a task that only executes once
- PeriodicTimer is a task that executes at regular intervals
- HarmonicEvent is a task that executes when another task is started
- JoinEvent is a task that executes when another task has stopped (completed)
- InterruptEvent is a task that executes when there is a hardware event at a given interrupt vector
- UserEvent is a task that is executed programmatically by the application

The Jbed scheduler conducts admission tests to determine whether to allow a new Task to start. Rather than using execution time analysis, Jbed determines whether to admit a task based on three values that must be provided when a new real-time task is created:

- Deadline - the amount of time within which a given task must execute
- Duration - the amount of time provided for a task to actually perform its job
- Allowance - the amount of time allotted to a task to perform exception handling when the duration is exceeded

Also of interest: Jbed does not allow a Thread (a.k.a. a normal Java thread that is not a real-time entity) and a Task (a.k.a. a real-time entity) to synchronize on the same object. Jbed uses priority inheritance to prevent priority inversion. Tasks are scheduled using the Earliest Deadline First method, while Threads are scheduled using round-robin and priority scheduling. Jbed does not provide for programmer-managed heap memory objects. Finally, Jbed supports the JDK 1.1.8 API.

aJile - aJ-PC104

aJile Systems produces the aJ-PC104 system (http://www.ajile.com/). This product consists of a set of tools for project management, target binary file creation, and execution environment management. The target environment is a proprietary board which includes the Rockwell-Collins JEM2 chip with a PC/104 interface. aJile also provides a limited, and poorly documented API for the creation of real-time tasks.

The JEM2 chip does direct bytecode execution so no RTOS is necessary. At the time the evaluation was conducted, there was no garbage collection provided by the system. This was the major hindrance to evaluation of this product because virtually every non-trivial Java application in existence depends heavily on garbage collection. Furthermore, the board has eight megabytes of RAM which is a constraint for JVM based applications. The lack of garbage collection in an environment that has so little memory makes this an especially acute problem. In order for applications to run on this board, they must be carefully coded to reuse object instances and minimize object creation. The aj-PC104 product does provide the programmer with the ability to create and manage heap memory entities, but only for variables that are system data types – not for object instances. The aJ-PC104 board supports two simultaneous JVMs. This is so that one VM can support real-time threads and the other can support normal threads.

To build an application for aJ-PC104, the programmer uses any development environment to write and compile the Java code. Then the programmer must use aJile's JEMBuilder tool to create the binary file that the board uses. The execution environment is managed using aJile's Charade tool.

4. EVALUATION METHODS

To evaluate these products, several benchmarks originally used on another effort [Obenland 2000] were selected and ported from C++ to Java. When attempting to run these benchmarks, several problems were uncovered that had nothing whatsoever to do with product performance.

The first type of evaluation that we selected was a measure of timer jitter. We conducted the jitter tests by varying the CPU load and the number of simultaneous threads. This illustrates the ability of each platform to perform periodic tasks. To create the CPU load, we implemented the functions contained in the Whetstone benchmark suite. Each thread periodically sampled the system clock. Deviation from expected sample times represented the actual jitter present in the system. Figures 2 and 3 in section 5 represent the test results.

The second evaluation method was a measure of the product's implementation of priorities and ability to avoid priority inversion. To measure this, we created two tests.

The first priority test, referenced in this paper as the "Uniqueness" test, used standard Java library calls to determine the number of priorities available to the system. The program then started one thread at each priority level. Each thread then proceeded to execute some arithmetic operations designed to keep the processor busy. On completion, the threads displayed the current system time. In theory, in a system with ten priority levels, the thread at priority ten will complete before the thread at priority nine, which will complete before the thread at priority eight, etc. In practice, this is not always the case.

The second priority test, referenced later as the "Priority Inversion" test created three threads. One thread was assigned the highest priority level, another thread received the normal priority level, and the final thread received the lowest priority level. The threads were all started nearly simultaneously. The high and low-priority threads synchronized on the same object while the medium-priority thread went straight to work. The low-priority thread was guaranteed to obtain the lock before the high-priority thread. A system that avoids priority inversion would ensure that the low-priority thread completes as quickly as possible so that the high-priority thread can then continue. If priority inversion takes place, then the expected completion order of the threads is: medium, low, high. If priority inversion is avoided, then the completion order is: low, high, medium. Clearly it is desirable to avoid priority inversion.

Other tests that were attempted:

Hartstone – resulted in null pointer exceptions in Jbed, ran cleanly in PERC, could not run in aJ-PC104 because of lack of garbage collection

VolanoMark – PERC VM crashed with Dr Watson error, JDK 1.3 ran much longer but eventually crashed with a Java exception, could not run on aJile (garbage collection) or Jbed (license)

5. EVALUATION RESULTS

Ideally, when evaluating products, one runs exactly the same tests on each product under exactly the same conditions and then compares the results. Java makes the promise, "Write Once, Run Anywhere." This creates the hope that various environments that run Java will deliver on that promise and thereby facilitate the execution of fair and equal evaluation. Our testing revealed that this promise is not yet reality in the real-time application domain.

Differences in APIs, the lack of a command-line interface in Jbed and aJ-PC104, incomplete versions of the standard Java API, and the lack of garbage collection in the aJ-PC104 environment made it very difficult to evaluate consistently. API differences often required code modifications. This resulted in different application code for the same functional test between products. Those differences represent a threat to the validity of the tests. To be as fair as possible, we limited the scope of code differences between tests, but we could not eliminate them.

The initial assessment of what is a fair comparison rules out tests for raw speed. This is because the products under evaluation execute on diverse platforms where computing speed is not necessarily of highest importance. What is important is determinism - the ability to set and meet deadlines.

5.1 Jitter Test

The first jitter test used the standard Sun Microsystems Java Development Kit (JDK) Version 1.3.

To perform the jitter time measurements a precision timer is required. JDK 1.3 offers this as described below. However all JVMs evaluated required JDK 1.1.x which did not have the native timer feature. Typically this feature is provided through a custom API for timing. A series of time measurements using the Thread.sleep() call followed by a System.currentTimeMillis() call. The JDK 1.3 has a new class called Timer that can be used to create periodic tasks.

The JDK scheduler implementation appears to support a granularity of 10 milliseconds. However, on average, the scheduler did awaken the threads at the right time. For example, an attempt to sleep for 49 milliseconds for 7 iterations resulted in the following series of clock queries: 50, 50, 50, 50, 50, 40, 50. The average of these numbers is 48.6.

5.1.1 Esmertec - Jbed

The Jbed scheduler granularity is 1 millisecond. We reached this conclusion based on the fact that all attempts to create periodic tasks that execute more often than once per millisecond resulted in actual execution periods near 1 millisecond.

Jbed reserves at least fifteen percent of the total CPU time to handle scheduling and garbage collection. We reached this conclusion based on the fact that any attempt to create a series of tasks such that the total of all durations and allowances exceeded eighty five percent of the deadline resulted in admission failures.

Jitter did not significantly increase with high CPU loads. However, jitter did significantly increase when the number of tasks increased. Jitter also increased proportional to increases in the deadline value.

Figures 2 and 3 illustrate some of these findings. Note that the timer jitter present in the system increased significantly as the number of threads increased.

While testing, ArrayIndexOutOfBoundsExceptions and NullPointerExceptions were thrown by code that ran without exceptions on other vendors' VMs.

5.1.2 aJile - aJ-PC104

Evaluation of this product for susceptibility to timer jitter clearly illustrated its primary limitations. The test was designed to show whether the internal clock performs well, and it does. However, other limitations became quite obvious during this test.

To conduct this test, we attempted to use source code that was furnished by an engineer at aJile Systems (see Appendix X). The code sample that was provided to us used aJile's PeriodicThread and PianoRoll classes. After extensive testing, we determined that all attempts to use those classes yielded execution-time behavior that was unpredictable. This behavior often included: repeated and unsuccessful attempts to start the application, no output whatsoever, and unexpected program termination.

On the rare occasions when the aJile classes did execute, the PeriodicThread class executed with consistent, but poor results. Typically, a PeriodicThread would sometimes wake up and execute right on time, but would often execute as much as 2 full seconds late.

We made other attempts to test the timer using a simple Thread.sleep() call with multiple threads. All tests using this method resulted in expected behavior. Threads with periods ranging from 1 to 1000 milliseconds *always* executed on time. The summary of the jitter tests performed on the aJile board:

- Timer granularity was 1 millisecond
- Every periodic thread executed exactly on time
- OutOfMemoryExceptions occurred with 8 threads, so the scheduler could not really be stressed

This series of tests illustrated two things quite clearly. First, the aJile board's limited RAM, combined with the lack of garbage collection, severely restricts the kinds of tasks that can be accomplished. Even with object reuse and limited string creation, out-of-memory errors occur quite often. Second, use of the classes that are provided by aJile often results in unpredictable behavior.

5.1.3 NewMonics - PERC

This product showed high susceptibility to jitter when the CPU was busy. The timer appears to work with 1 millisecond granularity when there is no CPU load, but balloons to approximately 25 millisecond granularity when the CPU is busy. Since there is no API for a timer, the experiment was conducted using the standard Thread.sleep() call.

Priority Test
Table 3 displays the results of the priority tests that were conducted on the four products.

The Jbed board honors the priority levels and preserves uniqueness of completion.

The aJ-PC104 board's scheduler honors thread priorities but does not avoid priority inversion.

PERC's scheduler honors thread priorities and uses priority ceiling emulation to avoid priority inversion.

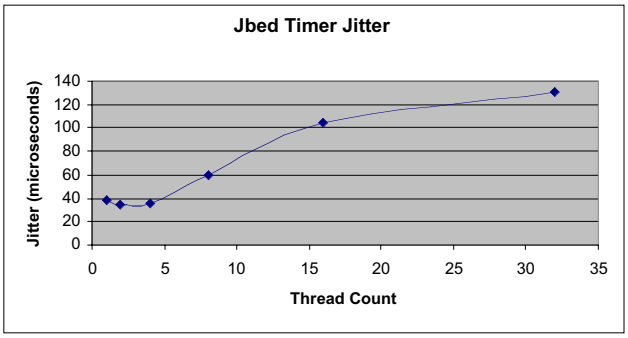

Figure 2 Jbed Jitter versus thread count

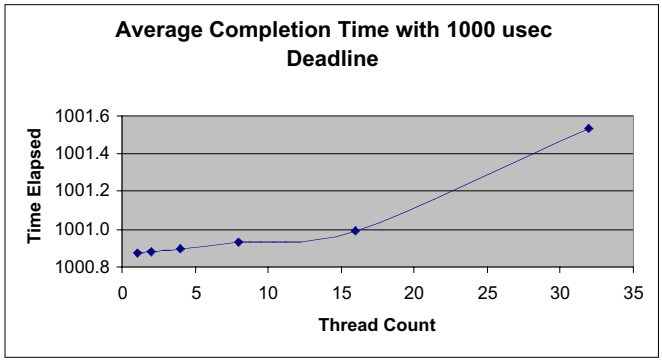

Figure 3 Jbed lateness versus thread count

Table 3 represents the various and products priority issues.

	Priority Levels in the System	**Uniqueness Test** (Thread completion order)	**Priority Inversion Test**
NT/JDK 1.3	1-10	**10, 8, 9, 6, 7, 5, 4, 3, 1, 2**	Inversion occurred
Jbed	1-10	10, 9, 8, 7, 6, 5, 4, 3, 2, 1	Inversion avoided
AJ-PC104	1-10	10, 9, 8, 7, 6, 5, 4, 3, 2, 1	Inversion occurred
PERC	1-10	10, 9, 8, 7, 6, 5, 4, 3, 2, 1	Inversion avoided

Table 3 Priority maintenance

Based on the results in the table above, two of the three products evaluated operated correctly for both priority uniqueness and inversion tests. These are Jbed and PERC.

6. RECOMMENDATIONS

The current situation for RTJ implementations meeting either specification is unclear in the near term without compliance test suites. In order to get industry involvement more than a specification is needed.

- Open source research implementations of the specification feature set are needed. The authors recommend at least two implementation be initiated. One University effort and one commercial effort.
- LINUX is a strong contender for many of the commercial vendors that wish to provide RTOS capabilities and JAVA. A closer examination is needed to understand where the platform consisting of LINUX with real-time services and a JVM fit into future platforms for the DoD.
- New market spaces are being created at an increasing pace around devices with soft real-time features that need network access. The feature set of Java integrated into new microprocessors that will directly execute Java byte code need investigation. These systems need to be assessed against the DoD mission space for applicability. Do these systems offer enough performance to replace the legacy RTOS and JVM systems being sold today for specific DoD implementations?
- Track and monitor vendor implementation and conformance to a specification. For example: aJile announced that developing a reference implementation and compliance test suites [aJile 2000]. This includes IBM VisualAge Micro Edition Real Time Extensions for the J9 Virtual Machine [IBM2000].
- Industry/consortium developed test suite is need for certification and compliance testing.

7. SUMMARY

The long term outlook for Java deployment is strong. As Java is applied to real-time systems its API needs to be developed in a consistent and open process. This paper provides some initial insight into the specification issues, real-time implementations and a start into evaluation of products. Some of the issues raised in this assessment are questioning the ability of the vendors to reach the NIST requirements for a RTJ platform.

The trade space is currently 1) JVM/LINUX, 2) JVM/RTOS, 3) RTOS with a JVM task and 4) emerging direct execution processors with silicon based JVM and

RTOS. Issues include changing mind share of legacy RTOS vendors and integrators, seriously measuring performance, development productivity and maintenance costs of this trade space.

ACKNOWLEDGEMENTS

The authors would like to thank Dr. Janos Sztipanovits of the DARPA Information Technology Office for support of this effort.

REFERENCES

[aJile 2000]
Le Ngoc D., Hardin, D, “Low-power, Direct Java Execution for Real-Time Networked Embedded Applications”, aJile.

[Bollella 2000]
Bollella, G., Gosling, J., Brosgol, B., Dibble, P., Furr, S., Turnbull, M., “The Real-Time Specification for Java”, Addison-Wesley, 2000.

[IBM 2000]
International Business Machines, J9 JVM and J2ME with real-time extensions http://www.ibm.com/embedded/

[Jensen 1999]
Jensen, D, “Real-Time Java Status Report”, The Mitre Corporation, January 1999

[Obenland 2000]
Obenland, K, “The Use of POSIX in Real-time Systems, Assessing its Effectiveness and Performance”, Proceedings of Embedded Systems Conference, The Mitre Corporation, September 2000

[Nilsen1995]
Nilsen, Kelvin, “Issues in the Design and Implementation of Real-Time Java”, NewMonics, Nov 1995, http://www.newmonics.com/pdf/RTJI.pdf

[RTJ.ORG]
The Real-time for Java Experts Group, http://www.rtj.org

[RTJWG 2000]
Real-time Java Working Group, “Real-Time Core Extensions”, J Consortium Specification No. T1-00-01, http://www.j-consortium.org/rtwg/s1.pdf.