Fully Time Deterministic Java™

Jean-Marie Dautelle[1]
Raytheon, Marlborough, MA, 01752

To use Java in Real-Time/Safety-Critical systems requires highly efficient predictable code execution. Techniques to achieve this include Ahead-Of-Time compilation, incremental Garbage Collection and the use of RTSJ-compliant Virtual Machine. However these are not sufficient as time-predictability can easily be ruined by the use of the standard library (lazy initialization, array resizing, etc.). To achieve true time predictability one must have a time deterministic library. This paper introduces such open source library (called Javolution) written by the author suitable for embedded or server-side applications and used for safety-critical application worldwide. Specifically, it will be demonstrated how this new library enables real-time data distribution (XML based) and allows for direct/immediate data exchange between Java programs and native C/C++ software.

Nomenclature

JVM = Java™ Virtual Machine
RTSJ = Real-Time Specification for Java™
GC = Garbage Collection
JCP = Java Community Process
StAX = Streaming API for XML

I. Introduction

The term "Real-Time" in the early days of computing meant "capable of simulating a process at a rate that matched that of the real process itself". But as computers were used for more than just simulation this term came with the imperative that an event is reacted to within a strict deadline. What was an interesting "feature" in the past became a key characteristic of modern day systems not only in the embedded world but for large distributed systems as well. The NASDAQ stock exchange for example is using real-time Java to prototype their next generation system[2]. On the military side, the DDG-1000 next-generation multi-mission destroyer is also "powered" by real-time Java software. Enhanced productivity, scalability and reliability are the main reasons why mission critical applications are switching from Ada/C/C++ to Java, saving hundreds thousands of dollars off development cost in the process[3].

Unfortunately, the real-time "chain" is as strong as its weakest link. And while most of the hurdles in using Java for real-time/safety critical systems have been overcome[4], still one major issue remains: The standard library itself! What is the point of having ahead-of-time compilation, real-time garbage collection, a highly time deterministic operating system if a simple call to add one element to a standard Java collection results in *delays of tens of milliseconds* because the collection had resized itself internally?

In this paper we discuss why the standard Java library is not time-predictable (not even RTSJ safe) and why a real-time Java environment should include a time-deterministic library such as the multi-platform Javolution[5] solution.

---

[1] Senior Principal Engineer, Raytheon, Dept. 30047, 1001 Boston Post Road, Marlborough, MA, 01752
Jean-Marie Dautelle is also an elected member to the Java Executive Committee (http://jcp.org)

[2] At JavaOne 2007, NASDAQ's CIO took the stage to brag on their current system, considered the fasted in the industry, which process more than 150,000 transactions per second.

[3] COTS Journal – July 2006 : "Software Modernization Is Key to Controlling Costs, Complexity"

[4] Validating Java™ for Safety-Critical Applications - AIAA 2005-6812

[5] Javolution (http://javolution.org) supports J2ME, J2SE/J2EE and even GCJ (GNU Compiler)

II. The Standard Java Library

One could argue that the strength of Java is its comprehensive standard library which includes components for networking, graphical user interfaces, XML processing, logging, database access and many other areas. These components are well tested, standardized (through the JCP) and available for any conforming implementation. The problem is that these components had never been intended to be used for safety critical systems. Even to this day, Sun Java license states explicitly that Java should not be used for the operation of "Nuclear facility"[6].

A. Throughput versus Time-Determinism

Because most Java applications were (and still are) server-type applications, throughput was paramount. As long as 99% of the time the operation was performed very fast, it did not matter that for the remainder 1% of the time the processing had to encounter significant delays because of:

- Arrays being allocated and copied (e.g. internal resizing of StringBuilder, Vector, ArrayList**)**
- Sudden burst of computation (e.g. rehashing of HashMap, HashSet).
- Long garbage collection pauses (even with incremental GC) due to memory fragmentation when large arrays were suddenly allocated.

B. RTSJ Memory Clash

In order to get the garbage collector "out of the picture", RTSJ provides specific memory areas not affected by garbage collection such as ImmortalMemory and ScopedMemory. All static instances are allocated in immortal memory (to be accessible by NoHeapRealtimeThread) and critical threads execute in scoped memory at a higher priority than the Garbage Collector. But there again using the standard library would be dangerous as memory allocation might be performed surreptitiously and could result into an IllegalAssignment error. Lets look at the HashMap class classic example. When a key-value association is performed a new entry object is dynamically allocated. If the map is static (in immortal memory) it cannot be used by threads in scoped memory and it cannot be used by threads running in immortal memory either without producing a memory leak when associations are removed (the standard library counts upon GC to recycle the memory of deleted entries, but GC is forbidden to touch immortal memory). To summarize, a simple class like Foo below as well as any class using it are unsafe for real-time threads.

```
public class Foo {
    // RTSJ Unsafe - Memory leaks when entries removed.
    //             - Error when new entries while in ScopedArea.
    static HashMap<Foo, Bar> map = new HashMap<Foo, Bar>();
}
```

Attempts have been made to automatically identify all No-Heap Safe classes of the standard library[7], but the problem had proven to be hard to solve.

[6] Sun Microsystems License Agreement - http://www.java.com/en/download/license.jsp
[7] No-Heap Safe Classes by Peter Dible - http://www.rtsj.org/docs/noheapSafe1/Noheapsafeclasses4.html

III. The Javolution Library (overview)

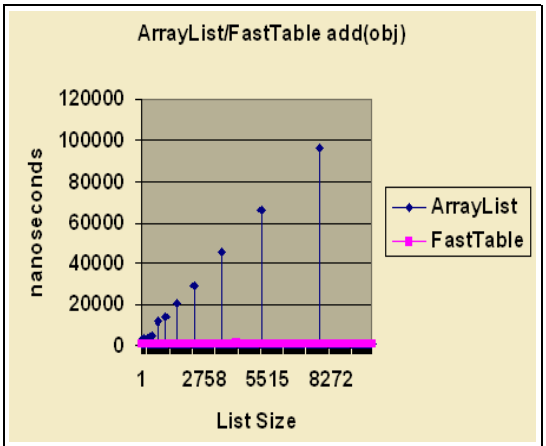

Figure 1: List.add(Object) Execution Time

The Javolution library provides *time-deterministic* and *RTSJ-Safe* alternative implementations of the standard library interfaces. The Javolution collections for example implement the standard collection interfaces and can be used as drop-in replacement. These collections have additional characteristics extremely valuable for real-time systems such as thread-safe without synchronization, support for custom key/value comparators, direct record iterations (no object creation), etc. Time-determinism behavior is achieved through incremental capacity increases instead of full resizing. In other words, resizing occurs more often but has less impact (on execution time or memory fragmentation).

An important aspect of Javolution implementation is that all classes are RTSJ-Safe. If an object has to perform some lazy initialization or increase its capacity this is always done in the same memory area as the object itself.

```
public class Foo {
    // RTSJ Safe - New entries are in ImmortalMemory,
    //           - Removed entries are recycled internally.
    static FastMap<Foo, Bar> map = new FastMap<Foo, Bar>();
}
```

When high-level components are implemented using Javolution components, these high-level components inherit from the same real-time characteristics guaranteed by the Javolution components (time determinism and RTSJ Safety). On the other hand high level components based on standard components might suffer from the same kind of time unpredictability which plagues the standard library.

IV. Real-Time I/O

Rarely safety critical systems work in isolation. Systems have to communicate with other similar systems, legacy systems or the hardware. But once again the "server" background of Java led to poor/incomplete support especially for the embedded domain. The RTSJ allows direct access to physical memory which means that device drivers could be created and written entirely in Java. But unlike C/C++, Java does not support struct/union which makes writing such driver difficult and error prone. Unlike C/C++, the storage layout of Java objects is not determined by the compiler. The layout of objects in memory is deferred to run time and determined by the interpreter (or just-in-time compiler). This approach allows for dynamic loading and binding; but also makes interfacing with C/C++ code or the hardware difficult. Javolution addresses this particular issue in the form of two public domain classes: Struct and Union. These two classes mimic the C struct and union types. They follow the same alignment rules, support the same features (e.g. bit fields, packing) and they make it extremely easy to convert C header files to Java classes (one-to-one mapping). Using these classes, embedded systems can map Java objects to physical address in order to control hardware devices or communicate through shared memory with external applications.

```
class Clock extends Struct { // Hardware clock mapped to memory.
    Unsigned16 seconds  = new Unsigned16(5); // unsigned short seconds:5
    Unsigned16 minutes  = new Unsigned16(5); // unsigned short minutes:5
    Unsigned16 hours    = new Unsigned16(4); // unsigned short hours:4
    . . .
}
```

V. Garbage-Free XML Serialization

Serialization is the process of saving an object onto a storage medium (such as a memory buffer) or to transmit it across a network connection in a particular form. In order to achieve platform/language neutrality the XML format had become the standard for such transformation. The Service Oriented Architecture Protocol (SOAP) for example relies heavily on XML serialization. Unfortunately, this process in Java is extremely messy (generates a lot of garbage). The Standard XML Readers/Writers are "String" based; the "String" class being immutable; its instances can only be recycled through garbage collection. This has a serious impact on performance and memory footprint as well (C/C++ XML parsers are about 2-4x faster because they don't suffer from this limitation). A solution to this problem was to slightly modify the StAX specification and use CharSequence instead of String[8]. This small change made it possible for XML readers/writers to use character buffers as CharSequence and made Javolution XML parsing/formatting as fast as its C/C++ counterpart.

With such clean (no garbage generated) serialization/deserialization engine, XML could then be used for time critical communications (no GC interruption).

VI. Separation of Concerns

Separation of concerns is very powerful programming principle and easier than it looks. Basically, it could be summarized as the "pass the buck principle". If you don't know what to do with some information, just give it to someone else who might know. A frequent example is the catching of exceptions too early (with some logging processing) instead of throwing a checked exception. Unfortunately, within the standard library they are still plenty of cases where the separation of concerns is not as good as it could be. For example logging! Using the standard logging, the code has to know which logger to log to? Why?

Javolution has a rather simple solution to this problem: "Context Programming"! It basically says that every thread has a context which can be customized by someone else (the one who knows what to do). Then, your code looks a lot cleaner and is way more flexible as you don't have to worry about logging, security, performance etc. in your low level methods

```
void myMethod() {
    ...
    LogContext.info("Don't know where this is going to be logged to");
    ...
}
```

Separation of concerns greatly facilitates writing safety-critical application because it allows for different behavior based upon the thread criticality *while still running the same code*!

Javolution has few out-of-the-box contexts already:

- *LocalContext* - To define locally scoped environment settings.
- *ConcurrentContext* - To take advantage of concurrent algorithms on multi-processors systems.
- *AllocatorContext* - To control object allocation, e.g. *StackContext* to allocate on the stack
- *LogContext* - For thread-based or object-based logging capability.
- *PersistentContext* - To achieve persistency across multiple program execution.
- *SecurityContext* - To address application-level security concerns.
- *TestContext* - To address varied aspect of testing such as performance and regression.

[8] A String being a CharSequence this change has very little impact on existing StAX code.

VII. Performance and Regression Tests

Too often unit tests focus on one aspect: "Validation". But although a code modification might not break your application; it may very well impact the performance significantly (for the better or the worst). External elements (JVM, O/S, memory available) are also likely to affect performance. For hard real-time applications missing a "deadline" can be seen as a critical failure. It is therefore extremely important to not only be able to measure the performance of your code but also to be able to detect automatically (regression tests) when any change you made in your code or runtime environment breaks your timing assumptions.

To facilitate such regression, Javolution provides a specialized context, the *TimeContext* capable of measuring and verifying the minimum/average/maximum execution time of any test case.

```
class MyTestCase extends TestCase() {
     ...
     public void validate() {
          long ns = TimeContext.getMaximumTime("ns");
          TimeContext.assertTrue(ns < 100); // Error if execution time is
          ...                               // more than 100 ns.
     }
}
```

Developers may create others types of contexts such as the memory context to check the memory footprint. By running the same test suite but within varied contexts, developers can focus on any particular aspect of interest such as behavior, performance, memory usage, etc.

VIII. Conclusion

Ensuring bounded response time is of interest to any interactive application even non real-time. But for safety critical applications it is crucial. As we have seen in this paper using a RTSJ VM is not enough. One may hope that more and more consideration will be given to time-determinism when implementing Java specifications. Fortunately, the community effort has already started with Javolution and the creation of the JSR-302 - Safety Critical Technology. The Javolution project has proven to be quite popular[9] and is currently being leveraged by developers from many reputable companies (Raytheon, Sun, IBM, Lockeed Martin, Thales, BEA, Blockbuster, etc.)

Finally, it should be noted that real-time is not incompatible with high performance. In many instances Javolution classes are faster than their standard counterparts, proving that you can be both real-time and real-fast!

[9] The Javolution web site has more than 1000+ visits a day and 2000+ library downloads a month.