Real-Time Java: An Introduction

Roberto Casadei
11153555@studentmail.ul.ie

CS4218 - Real-Time Systems
University of Limerick
Limerick, Ireland
Spring 2012

Within the ERASMUS Programme

***Abstract*—This essay aims to provide an introduction to Real Time Java, through an overview of the RTSJ (Real Time Specifications for Java) and a survey of the main issues regarding the specification itself and its implementation.**

First of all, a foreword about real time systems and the Java technology is provided as well as an historical summary for the specification development. An outline of the NIST Requirements for Real-Time Extensions for the Java Platform is also provided.

Secondly, we'll see the main objectives and the principles underlying the RTJS specification and the mechanisms used to achieve them as well as the most significant abstractions.

Next, we'll take a look at the main implementations of the specification, focusing on the approaches used to overcome predictability and timeliness issues, so that we'll be able to perform an evaluation of the technology by a practical perspective as well.

Finally, based on what we have described so far, we'll be able to draw some conclusions, emphasizing on the limitations and the strengths of Real Time Java.

***Keywords*-Real Time Java (RTJ); Real Time Specifications for Java (RTSJ); real-time systems; Java Real-Time System (RTS); IBM Websphere RealTime; JamaicaVM; garbage collector;**

I. Introduction

Real time is a quantitative notion of time [1]. Real time systems have to deal with real time. The logical correctness of such systems not only depends upon the correctness of their outputs, but also upon their timeliness as logically correct results are incorrect if provided late [2].
The main taxonomy divides real time systems into two types:

- Hard real time systems: if a validation that the system meets its deadlines all the time is strictly required by the user
- Soft real time systems: if no such validation is necessary or statistical requirements about the timeliness are sufficient or acceptable

but multiple definitions of these terms can be found in the literature [3].

Real Time Java (RTJ) is a term used to refer to the overall set of technologies that supports the development of real time systems using the Java programming language. Of course the Java platform (as it is) is unfit for this goal, especially due to the general-purpose scope of the platform and the performance penalties caused by the garbage

collector.
In fact, there are some drawbacks that must be overcome [4]:

- Class loading and interpreted methods delays
- Large and unpredictable garbage-collection delays
- Lack of guarantees for thread priority enforcement and thread scheduling

This is why the JSR-1[1] was proposed in 1998.

Its aim was to extend *The Java Language Specification* and *The Java Virtual Machine Specification* to enable Java for real time system development through

> ”an Application Programming Interface that supports the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints”

[5].

In late 1990s, the National Institute of Standards and Technology (NIST) promoted the creation of a Requirement Group for providing a basis for Real-Time Extensions to the Java Platform. Several companies and organizations were involved in this attempt to provide a set of guiding principles and requirements to enable Java for real time.
Based on this effort, the JSR-1 draft specification was published in 1999, whereas in 2002 The Real-Time Specification for Java standard was approved, resulting in the first commercial implementation in 2003 by TimeSys, which merged into the RTSJ RI[2] in early 2004. Next, other implementations were released, such as IBM Websphere Real Time and JamaicaVM. In 2006, RTSJ 1.0.2 was available, primarily resulting from the experience collected by the RTSJ implementators.

[1]Java Specification Request. They are the actual descriptions that are meant to become specifications for the Java platform.

[2]Reference Implementation. Model implementation against with the various implementations are developed and against with the standard process primarily looks for feedback.

Eventually, the RTSJ 1.1 proposal has been issued, being developed under JSR-282, currently inactive.

The fact of RTSJ being born from the first JSR may be seen as the realisation of the importance of facing the development of real time systems with the benefits afforded by the Java technology [6], such as:

- the broad industry acceptance
- the *”write once, run everywhere”* philosophy
- the massive collection of libraries
- the object orientation for high-level software construction
- and the ”security sandbox” provided by the JVM

What is important to note is that the RTSJ specification targets both the language *and* the virtual machine, thus providing an extension to the Java platform that collapse into what we can call Real-Time Java or the Real-Time Java platform.

II. Overview of NIST Requirements for Real-Time Extensions for the Java Platform [7]

Several representatives from companies and organizations have contributed to identify and select the most important aspects to consider in the attempt to make Java a suitable choice for a wide range of real time applications. One of the first consensus was to develop a Requirements Document that clearly specifies the timeliness properties of the Java platform, precisely and independently from the actual implementations [8].

This also produced the need for a common well-defined terminology as some real-time terms are often used with slightly different meanings (and sometimes contradictory), fact that is also self-evident in the literature. This is particularly true and sensible for the meaning of ”hard real-time”. The large part of the Requirements Document is so focused on providing a conceptual common basis for discussion.

In particular, some distinctions are significant and deserve to be emphasized:

- scheduling vs. dispatching
- schedulable entities vs. non schedulable entities
- individual timeliness vs. collective timeliness
- hard deadline vs. the significance of not meeting a hard deadline

The last item is quite instructive and interesting, and follows from the definitions of hard deadline and soft deadline. According to the NIST requirements, the latter is about the degree of timeliness in relation to the extent for which the deadline is satisfied, usually measured in terms of lateness (completion time minus deadline) or tardiness (absolute value of lateness). The hard deadline is a special case of a soft deadline where the ”more” and ”less” timely collapses to ”timely” and ”not timely”, respectively.

Next, the Requirement Group considers the Java traits that provide the motivations for its use by the real time community. Foremost among these are:

- The productivity boost allowed by the high level of abstraction (languages such as C are deficient by this point of view)
- The ease of use and mastery with respect to other more complex languages such as C++
- Support for component integration, reuse, and distribution
- Support for application portability

The guiding principles are useful to understand the direction of this real time extension as well as the role of Java in this field.
The first principle is a continuation of those which have led the design of the Java language and sets the case for preferring the ease of use over performance and efficiency. The reason is quite straightforward and derives from the importance of preserving one of the key advantages of Java, which also supports the motivation of the extensions themshelves.
The second principle is more ambitious and addresses the creation of very long-lived software, with ”useful lifetimes that span multiple decades” [7].
The third principle puts forward another trade-off, promoting requirements that are intended to be both ”pragmatic” and ”visionary” [7]. This aspect is crucial because it requires Real-Time Java to support the current practices as well as an environment that allows for future improvements along with the ability to keep the pace with the state-of-art and even to advance it.

The real-time applications are several and various, with potentially different requirements that make it difficult to put them all under a monolitic system. The Requirement Group recognize that and suggest to organize the overall set of functionalities in:

- Core functionality: provides a basis for more specific functionality and is meant to satisfy the demands of common real-time applications
- Profiles: set of related functionality that addresses particular needs by extending the core; e.g.: distributed real-time application support, fault-tolerance support, predictability support [9] [10] and so on

In particular, profiles are meant to be easily pluggable into the core system as a unit. They address needs at the system level, while for what concerns the user functionality, one of the intended goals of Real-Time Java is to enable the creation and integration of reusable real-time software components. Clearly, a component model for RTJ is required to fit with the new memory management model and scheduling model, possibly leveraging on new design patterns to achieve the necessary level of reuse [11].

The requirements are not massive and touch few key points, just to mention a few of them:

- Profile management
- Garbage Collector and preemption
- Relatioship between real-time Java threads and non-real-time Java threads
- Asynchronous events

Finally, from the set of core requirements, a set of goals and derived requirements are specified.

III. RTSJ Overview [5]

The Real-Time Specifications for Java is the main continuation of the NIST real-time extension requirements. Based on the effort carried out by the NIST Requirement Group, The Real-Time for Java Expert Group (RTJEG) defines new guiding principles and a set of seven enhanced areas with extended semantics.
The principles set the scope of the specification and are useful to understand the design choices as well as the main challenges:

- *Applicability to Particular Java Environments*: the specification must not be rescrictive for what concerns its use in particular Java environments and versions (such as J2ME); this is fundamental for targeting at embedded systems and for preserving the advantages which make Java a compelling solution in the real-time scenario
- *Backward Compatibility*: non-real-time Java programs must be able to execute on RTJS implementations
- *Write Once, Run Anywhere*: it's a continuation of the general Java principles, but it may be necessary to sacrifice binary portability in favour of predictability
- *Current Practice vs. Advanced Features*: the specification should not restrict implementations to advance in the state-of-art
- *Predictable Execution*: predictability is the central issue in real-time and should be given the first priority in all trade-offs
- *No Syntactic Extension*: the Java language is kept untouched, no new keywords or syntactic extensions are provided
- *Allow Variation in Implementation Decisions*: the specification shouldn't impact on the flexibility for implementers to provide their algorithms and their timing constraints

From these principles it is clear that the objective is to shift the advantages of the Java platform to the real-time systems field while satisfying the need for predictability. In addition, this has to be done in a way to *not* hinder the evolution of RTJ or, rather, the use of RTJ once future approaches to real-time computing are available, through excessively prescriptive choices.

Based on the principles, the RTJEG has identified the main areas that require a semantic extension in order to accomodate the real time needs for the Java platform. The effort has to be focused in order to improve these seven areas:

1) *Thread Scheduling and Dispatching*: a framework for the management of real-time Java threads is prominent along with enough flexibility for what concerns the scheduling infrastructure (mechanisms, algorithms and so on); a base scheduler (priority-based and preemptive) is also required in all the implementations
2) *Memory Management*: the memory model must be extended in order to support memory management mechanisms; in particular, the overhead caused by the garbage collector is recognized, resulting in the need to allow the allocation and the reclamation of objects outside the garbage-collected heap
3) *Synchronization and Resource Sharing*: safe real-time synchronization must be supported; in practice, conforming RTSJ implementations must provide implementations of the `synchronized` keyword that ensure that no priority inversion will never happen (using algorithms such as the *priority inheritance protocol* or the *priority ceiling emulation protocol*)
4) *Asynchronous Event Handling*: a real-time generalization of Java's asynchronous event handling mechanisms to satisfy the necessities of an "asynchronous world"
5) *Asynchronous Transfer of Control*: addresses how to asynchronously and efficiently transfer the control to another location from the current locus of logic execution (e.g. through an extension to Java's exception

handling mechanism)

6) *Asynchronous Thread Termination*: it's about allowing for safe asynchronous termination of real-time threads without the troubles (i.e. inconsistent state and deadlock) that are associated with the currently deprecated `stop()` and `destroy()` methods of Java's `Thread` class; this can be done by leveraging on the safety afforded by asynchronous event handling and asynchronous transfer of control mechanisms
7) *Physical Memory Access*: the RTSJ defines classes to enable direct byte-level access to physical memory

We can see how the enhanced areas are actually about three main issues: real-time scheduling, supporting asynchronous-ness safely, and memory management. The seven areas are so developed, resulting in a set of classes which models the concepts that provide the foundation for RTJ, as well as extensions for the virtual machine.

IV. RTSJ Design Overview: Main Abstractions [5]

First of all, a number of changes are brought about some standard Java classes in order to adjust their semantics in relation to real-time issues. Most notably:

- `getPriority()` and `setPriority()` of `Thread`, so that their behavior is consistent with the one that is required by the real-time subclasses
- `ThreadGroup` class behavior and methods

The RTSJ classes are implemented in the `javax.realtime` package.
The definition of *schedulable object* is given with respect to the base scheduler. `RealtimeThread` and `AsyncEventHandler` instances are schedulable objects. They implement the `Schedulable` interface, which provides methods for accessing the scheduling parameters, the associated scheduler, and performing the feasibility analysis. Note that there is difference between a `Schedulable` object and a *schedulable object* (defined with respect to the scheduler).
The `RealtimeThread` class extends the `java.lang.Thread` class. Real-time threads can be passed `SchedulingParameters`, `ReleaseParameters`, and `MemoryParameters` instances at construction, specifying their timing requirements and the memory area to be used.
`RealtimeThread` objects represent real-time threads (RTT) that are usually used for soft real-time scheduling, whereas for hard real-time scheduling typically objects of the specialized `NoHeapRealtimeThread` (NHRT) class are used as these threads cannot reference any object in the heap and consequently are never touched by GC delays; due to this restriction, they may also (if they are given a higher priority) safely preempt the garbage collector.

The base scheduler, that is the scheduler required by the specification, is fixed-priority preemptive with at least 28 unique priority levels. It is represented by the `PriorityScheduler` class. Its `instance()` method returns the same instance as `Scheduler.getDefaultScheduler()`, if no other scheduler is set to default. The base-scheduler class adds methods to cope with priorities to those that are made available by its parent `Scheduler` class, which represents a generic scheduler and provides abstract methods for dispatching schedulable objects and performing the feasibility analysis.

The concept of *memory area*, modeled by the `MemoryArea` class, is also introduced to represent the location in memory where objects are allocated. According to the specification, there are four types of memory areas:

- *Scoped Memory*: for objects whose lifetime is bound by scope
- *Physical Memory*: typically useful for per-

formance; a number of classes are provided to model a range of physical (memory) addresses
- *Immortal Memory*: area which is outside the garbage collector scope and is accessibile by all the threads; for example, the `System` properties are moved here
- *Heap Memory*: the `HeapMemory` class allows to access the singleton object that represents the heap

The utility of providing memory areas which are not affected by the garbage collector is self-evident considering that garbage collection delays may be in the order of hundreds of milliseconds [12] and that the majority of real-time systems use time units smaller than ten or hundred milliseconds [6]. Of course, it is not only an issue of time and efficiency as some systems might tolerate certain delays *if these were predictable* [13]: in this sense, avoiding garbage collection having place is a way to eliminate the problem from the root.

For what concerns the asynchronous event handling, the main abstractions are, not surprisingly, `AsynchEvent` and `AsyncEventHandler`. The former is used to represent a generic asynchronous event, something that happens internally to the RTSJ implementation or externally (such as a hardware interrupt or a signal); when it is `fire()`-d, the associated handler instances are scheduled and their `handleAsyncEvent()` method is called. The specification does not mention about the implementation of these mechanisms, but there are some issues that need to be overcome, such as avoiding the overpopulation of threads and the priority management for event handlers [14].

Timers are specialized asynchronous events; they are modeled by the `Timer` class. Timers are driven by time, i.e. by a `Clock` object. There is always at least one instance of the abstract `Clock` class, that is the *system real-time clock*, and is made accessible (as a singleton) through the `Clock.getRealtimeClock()` static method. As real-time systems often need high-resolution clocks [4] [6], the class `HighResolutionTime` allows for a nanosecond accuracy.

The specification requires `synchronized` code to apply priority inversion control strategies. Inversion control problems arise for example when a higher-priority thread contends for a resource held by a lower-priority thread. Such problems can be avoided through execution eligibility control algorithms, which are represented by the `MonitorControl` class. The main subclasses are `PriorityInheritance` and `PriorityCeilingEmulation`, and stand for the homonym algorithms; the former is required, whereas the latter is optional, though there may be cases where using both the algorithms is useful [15].

V. Quick Look at Some RTSJ Implementations

A. Java RTS

Initially under the name of Project Mackinac [16], **Java RTS**[3] is Oracle's commercial implementation of the RTSJ specification, compatible with Java Standard Edition 5.0. It runs on Solaris 10 or Linux with real-time POSIX[4] APIs.
The following are the main characteristics of Java RTS:

- Support for non-real-time, soft real-time, and hard real-time logic as required by the RTSJ specification
- Provides further extensions to the basic RTSJ, under the `com.sun.rtsjx` package
- Features to reduce the runtime jitter, such as precompiled methods and mechanisms that help avoiding the delays of the JIT compilation, such as the Initialization-Time-Compilation (ITC) scheme

[1]Java SE Real-Time System.
[2]Also known as POSIX.4: it's the part of POSIX that addresses the needs of real-time systems.

- Provides a non-real-time serial garbage collector and, since version 2.0, a fully concurrent *real-time garbage collector* (RTGC) which can be configured to exhibit a deterministic behavior
- A number of features depend on the host operating system
- Throughput performance for non-real-time logic up to 85% as fast as an equivalente non-real-time Java system
- Predictability performance for hard real-time logic: maximum latency of 20 microseconds and maximum jitter of 10 microseconds

Compilation [17] and garbage collection [18] are topics that deserve to be lingered over.
Normally, Java classes are initialized at a first-use basis and for methods the compilation is triggered once some internal counters reach certain values. These operations (static initialization and Just-in-Time compilation) introduce a jitter. In order to overcome these issues, classes are initialized at the application startup and methods can be assigned for ITC compilation, which causes them to be compiled at class initialization time. This assignment is performed by passing a list of method compilation information to Java RTS by command line or programmatically; the list is composed by tuples that specify: class name, method name, method signature, thread type (such as `rrt` or `nhrtt`).

Real-time garbage collectors (RTGC) are designed to perform their job of memory recycle while preserving the real-time system with a predictable behavior. In order to accomplish this goal, a wost-case execution time must be given for any task: this implies that the worst-case performance of several operations (such as memory access and thread preemption) must be known to support a deterministic execution [19]. Frequent drawbacks include the lack of support for RTGC preemption and the need of reconfiguration even when new low-priority threads are added into the system.
Sun's RTGC solve these defects by providing the ability to be preempted and ensuring the determinism only for critical tasks (where the criticality is based on priority), so that the addition of low-priority tasks doesn't impact the system's deterministic behavior. This and other RTGC functionalities can (and often need to) be tuned by setting multiple parameters; in particular, for hard real-time, the `RTGCriticalReservedBytes` parameter must be configured to prevent critical tasks to block while accessing to the memory because the RTGC has not completed in time its memory freeing job.

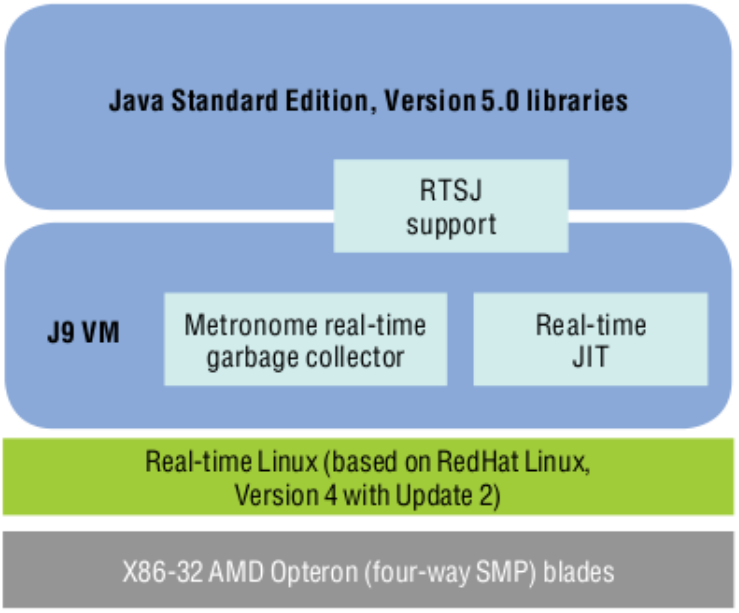

Figure 1. IBM Websphere Real Time Architecture [4]

B. *Websphere Real Time*

IBM Websphere Real Time is a fully-conformant RTSJ 1.0.2 implementation that provides both a Java Runtime Environment and a Software Development Kit to enable Java for the development of soft and/or hard real-time systems. Similarly to Java RTS, it also recognizes the garbage collector and the JIT compiler as the main sources of predictability issues.

The main traits of IBM Websphere Real Time version 3.0 are the following:

- Compatibility with Java Standard Edition version 7.0
- Supported platforms: standard Linux, IBM AIX[5], and Real-time Linux
- Incremental garbage-collection technology (called *Metronome*) to help ensuring predictability and avoiding the necessity of using complex memory management mechanisms (such as scoped memory); the (worst-case) pause time of the garbage collector can be configured to be as short as 1 millisecond, whereas for smaller latencies (down to 50 microseconds) the RTSJ `NoHeapRealtimeThread` features can be used (so to avoid GC delays)
- Enhanced-JIT and *Ahead Of Time* (AOT) compilation support

Again, it is instructive to go deeper into the details.
We already know that Just-In-Time compilation is one of the contributors to non-determinism. If the JIT compiler runs on the application thread, this is particularly true [20]. For this reason, in Websphere Real Time the JIT compilation is performed on a low-priority thread, allowing higher-priority threads to preempt it; a queue is used for the methods that need to be compiled. Moreover, the JIT compiler is aware of the garbage collector and can yield precedence to it, so that it can complete its work.

JIT compilation can be avoided or reduced through the static AOT compilation, which is a technique similar to the Java RTS' Initialization-Time-Compilation model. It turns out that AOT compilation can produce more-optimized code than the JIT compiler as the latter is subject to strict time execution boundaries, hindering it in producing strongly-optimized code; in addition, the AOT technique allows to avoid the use of the JIT compiler in environments (such as Java 2 Mobile Edition) where it might cause an undesired memory footprint [20].

Metronome is the RTGC available in Websphere Real Time. As a RTGC, it is in contrast to the so-called Stop-The-World (STW) garbage collectors [4] which, as the name suggests, stop the entire application in order to perform their memory recycling, potentially causing tasks to miss their deadline. The first intuitive step towards a solution is to break the memory recycling task down into small pieces so that the application doesn't suffer a single long pause anymore; instead, the garbage collection consists of more manageable units of work with short delay. This management adds more complexity to the operation because the application may change the heap during the overall garbage collection.
Two approaches are used to avoid a long pause caused by a STW GC [12]:

1) *Generational* garbage collection: also known as *ephemeral GC*, it divides the heap into two (or more) regions (called *generations*) and objects are moved on one of them based on their age; it's like having two boxes, one for "young" objects and one for "old" objects. Then, the basic principle is that, as the generation of short-lived objects is the more likely to be changed (i.e. containing garbage), the effort can be concentrated here. Instead, long-lived objects tend to be more stable, so the garbage collector can look at them more sporadically. Focusing on a small part of the heap is undoubtedly an advantage, as it implies less work to do. However, there are multiple issues concerning the promotion between generations [21] that need to be coped with.
2) *Incremental* garbage collection: according to this approach, the garbage collection is done *incrementally*. This means simply that just a portion of the garbage is found and deleted at each execution. Practically, it consists of dividing the work that would be performed

[1]IBM Advanced Interactive eXecutive: is a family of Unix-based operating systems developed by IBM

by an equivalent STW GC into small increments and executing these chunks so that a certain utilization ratio is guaranteed for the application execution. It should be noted that this approach is subject to different overhead sources and issues; for example, assignments in the heap need to be tracked and managed and the memory need to be compacted in order to avoid an excessive memory fragmentation.

Metronome is an incremental GC. The target utilization can be set by passing the `-Xgc:targetUtilization` parameter, which limits the CPU percentage assigned to the GC in a given window of time, to the application launcher.

C. JamaicaVM

Aicas JamaicaVM is a Java implementation built specifically for embedded and realtime applications.

Its main features [22] are here summerised:

- Guarantees for hard real-time without any restriction to the use of the Java language
- Support for Java SE 6
- The RTSJ specification is not fully supported: some classes are not implemented and some relaxations (which can be disabled by setting the `strictRTSJ` mode) are realized
- ROMable code: the JamaicaVM can be linked with all the Java class files into a standalone executable. As all the data required for execution is included in the standalone program, there's no need for filesystem support: consequently, the packed executable can be loaded into ROM or flash memories. The `jamaicabuilder` is used to create these standalone images
- Optimized execution performance through appropriate compilation techniques
- Small footprint: the memory consumption is kept as small as possible. The VM occupies alone less than 1MB (depending on the use of the standard library), and a number of precautions are taken to limit the memory usage (e.g. such as the compaction of Java class files)
- Tools for performance analysis are provided: for example, the *JamaicaVM Thread Monitor* can be used to track different VM-related events and look at the realtime behavior of the application, helping to fine-tune its execution timing
- Availability across different platforms: different Real-Time Operating Systems (RTOS) and non-RTOSs (Linux, Solaris, and Windows)
- A free *Personal Edition* is available

Let's analyse some design and implementation choices of JamaicaVM.

First of all, there is no distinction between non-real-time threads and real-time threads [22]: all the threads (also `java.lang.Thread`) that run in JamaicaVM are "realtime threads" and they are scheduled/preempted according to their priorities. The removal of the strict separation between the real-time part and the non-real-time part of an application (one of the main consequences of the RTSJ specification [22]) has been possible through the use of a real-time garbage collector [23]. JamaicaVM's RTGC is released at each memory allocation and can be preempted by higher priority threads, allowing critical tasks to perform their work without waiting the GC to complete. Its main characteristics [19] [23] are the following:

- It is basically a simple incremental, mark-and-sweep [24] GC
- Support for constant-time root scanning by mantaining all the root references in the heap (all the threads have their own root array in the heap)
- Fragmentation reduction through a memory layout with 32-bytes fixed-size blocks in the heap; objects are represented through linked lists whereas arrays are represented through trees
- Balanced ratio between GC overhead and

GC work by performing garbage collections increments after memory allocations; the number of increments can be fixed in order to avoid the GC jitter (with the drawbacks of increased overhead and possibility of `OutOfMemoryErrors`)

These and other implementation choices have shown to improve the garbage collector's runtime performance by 30% and the memory usage caused by garbage-collection-related information by 20% [23].

In addition to its RTGC, JamaicaVM offers a number of optimizations [22] that allow to minimize the footprint and the memory usage of an application and to increase its performance as well.
The *static compilation* performed by the JamaicaVM builder allows to produce machine code from bytecode: the former is about 20 or 30 times faster than interpreted code, but it is also less compact. For this reason, a selective static compilation can be used to set a compromise between memory footprint and fast execution; this works by using profiles, which can be generated through `jamaicavmp`.
Another method for reducing the memory footprint is called *smart linking* and works by removing the unused bytecode (i.e. classes and methods) from an application. The same consideration applies to resources such as language locales and timezone information, which can be excluded to the resulting bytecode (or machine code).

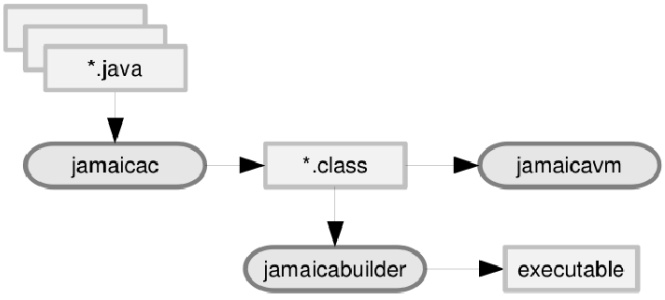

Figure 2. JamaicaVM tool chain [22]

VI. More Issues

We have seen so far how the different implementations have been trying to eliminate or reduce the primary sources of unpredictability in the Java platform, namely the **automatic garbage collector** and the **dynamic class loading**.
Predictable hard real-time behaviors require different guarantees [19]:

- worst-case upper-bound execution time for all Java operations
- predictable-succeeding executions (e.g. memory operations must not fail because a sufficient amount of memory freeing work has not been done)
- worst-case delay for thread preemption

For what concerns worst-case performance, it turns out that the **general-purpose libraries** provided by the programming languages such as Java or C++ are far from being perfect as they are built primarily to provide good *average-case* performance [25]. Libraries must provide worst-case guarantees for both execution time (for which blocking method calls have to be avoided) and memory usage and try to minimize them. In other words, reusable libraries needs to be developed on purpose for real-time, enabling for a predictable knowledge of worst-case upper-bound times.
An example of RTSJ-compliant library is *Javolution* [26], which provides high-performance and time-deterministic (in the microsecond range) classes equivalent to those in the Java standard library.

Concerning **multiprocessor support**, the RTSJ does not take any precise position [27]. There are several issues regarding scheduling and dispatching that need to be considered.
The implementations that we have looked at previously do provide support for Symmetric MultiProcessor (SMP) systems. For example, in IBM Websphere Real Time each processor has its own queue and checks are performed in order to ensure that priority is enforced across the queues;

load balancing is performed to mitigate queue overloading as well.

Predictability is essential, but often also **good performance** is required. A number of tests can be carried out in order to verify that a RTSJ implementation respects the time requirements [28]. These tests should consider of course the GC delay, the memory allocation time, the scheduling and dispatching latencies, the context-switch delay, the time involved in priority enforcement and in priority inversion avoidance, and so on.
Good RTSJ implementations and RTSJ-compliant libraries can contribute to obtain an acceptable efficiency.

The development of real-time systems is concerned at different levels:

- Application level
- System level
- Hardware level

Upper levels are built on the basis of the lower levels.
The RTSJ is concerned with both the application level (through its API) and the system level (through its implementation). A *real-time operating system* (RTOS) is not strictly required as far as the scheduling activities are carried out by the JVM. The RTSJ specifies a series of requirements for the **underlying operating system** [29] [30], but they are not too strict. We have seen that JamaicaVM is available for multiple RTOSs, whereas IBM Websphere Real Time can run on the top of *Real-time Linux*, which helps to reduce latencies for real-time applications through different features [4]:

- Efficient timer mechanisms with higher-resolution: a new 64-bit type (`ktime_t`) has been defined on purpose as well as new timer-related functions
- Priority inversion avoidance through Priority Inheritance
- Complete kernel preemption (`PREEMPT_RT` patch): many spin-locks are converted to mutexes to allow the preemption of parts of the kernel that were un-preemtible before; interrupt handlers are converted into real-time kernel threads and so they can be preempted as well
- Multiprocessor real-time scheduling

From this example, it is clear that an RTOS can help real-time applications to exhibit a more deterministic behavior.

Real-time systems can be also classified on the basis of their time quantum [6]. The RTSJ provides a class (`HighResolutionTime`) to deal with nanosecond time resolution, nevertheless it is clear that it has no use if the real-time clock doesn't support that **time accuracy**. Moreover, it turns out that the most time-critical part of the application is often the more important and usually the smallest [12].

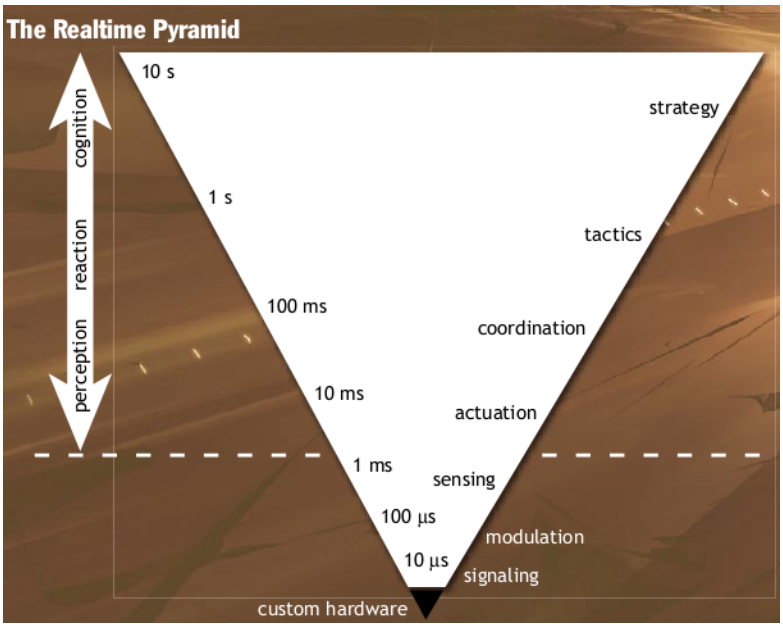

Figure 3. Time-criticality pyramid [12]

VII. Conclusion

We have looked at Real-Time Java by different perspectives. The principles underlying the RTSJ [7] [5] helped us to understand its direction and goals. The quick look at the main RTSJ abstractions gave us a taste of the design choices and the approach used to support the modelling of a real-time application. The survey of the characteristics of three RTSJ implementations, namely Java Real-Time System [16], Websphere Real Time [4], and JamaicaVM [22], has provided practical information as well as a presentation of the main challanges. Other issues have been mentioned, even though not exhaustively, to provide a broader view of the state of affairs: inadequacy of the standard Java library [25], multiprocessor support [27], efficiency issues [28], underlying platform [30], time resolution.

We now have several elements that can be used to draw some conclusions.

The RTSJ has had the undoubted credit for setting out Java in the context of real-time systems. A number of implementations (also commercial ones) has been created and interests about Real-Time Java are maturing. Case studies [31], evaluations [32] [33], profiles [9], and applications [34] [35] have been developed, effectively making RTJ worth to be considered.

One key aspect is the potential of Real-Time Java for **productivity**: the high-level constructs and the libraries can help the programmers with the modeling of the problem. However, RTSJ is not an easy framework to master as there are some features which are quite **complex** and/or error-prone (e.g. the scoped memory [31]). On one hand, this "elevation of timeliness characteristics and feasibility analysis to a first-class design aspect" [33] can be positive, by allowing a finer control of real-time elements; on the other hand, it requires the developers to be well-acquainted with real-time concepts, increasing the **abstraction** gap between the domain problem and the platform.

Concerning the Java standard library, there are some issues. The first is that, as it has been built before RTSJ was released, its classes are unaware of the new memory model and are designed in a way that is inadequate in the RTSJ context. Secondly, as the library classes are quite coupled, it is difficult to include only the used portion of it [33], impacting on the **memory footprint** which is one of the main Java drawbacks for what concerns real-time embedded systems. Project Golden Gate [33] compared the platforms VxWorks/C++ and Timesys Linux/Java with respect to memory and disk footprint; the RTJ platform disk usage was more than the double of the C++ counterpart, whereas its memory usage was 7780K against 156K. This is also caused by the JVM's memory footprint, and we have seen how JamaicaVM try to reduce the footprint through a light VM, code compaction and (for the library) the smart linking technique [22] [31].

The Real-Time Java technology does **not** seem to be **mature yet**.

In 2008, D. Hober and S. Elzer [36] said that

> There are currently very few benchmark measurements of how efficient and predictable a Real-Time Java system truly is.

Four years later, the situation has not changed dramatically.

Predictability is a key aspect in real time. The RTSJ specification is more about avoiding unpredictable traits than actually defining predictable behavior supports (this is why appropriate profiles [9] [10] are needed).

Some work has been done on the main sources of unpredictability (i.e. garbage collection and compilation). Some applications might find it acceptable, some other ones may require additional determinism and better performance.

REFERENCES

[1] R. Mall, *Real-Time Systems: Theory and Practice*. Prentice Hall, 2009.

[2] P. A. Laplante, *Real-Time Systems Design and Analysis*, 3rd ed. Wiley-IEEE Press, 2004.

[3] J. W. S. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[4] M. Dawson, M. Fulton, G. Porpora, and et all, "Creating predictable-performance java applications in real time," IBM, Tech. Rep., 2007.

[5] J. Gosling, G. Bollella, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*. Addison Wesley Longman, 2000.

[6] P. C. Dibble, *Real-Time Java Platform Programming*. Prentice Hall PTR, 2002.

[7] L. J. Carnahan and E. Marcus Ruark, "Requirements for real-time extensions for the java platform: Report from the requirement group for real-time extension for the java platform," National Institute of Standards and Technology, Tech. Rep., 2000.

[8] A. Wellings, *Concurrent and Real-Time Programming in Java*. Wiley, 2004.

[9] T. Bøgholm, R. R. Hansen, A. P. Ravn, B. Thomsen, and H. Søndergaard, "A predictable java profile: rationale and implementations," in *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ser. JTRES '09. New York, NY, USA: ACM, 2009, pp. 150–159. [Online]. Available: http://doi.acm.org/10.1145/1620405.1620427

[10] Y. Jin, "Predictable real-time java profile based on rtsj," in *Knowledge Discovery and Data Mining, 2009. WKDD 2009. Second International Workshop on*, jan. 2009, pp. 811 –815.

[11] M. Alrahmawy and A. Wellings, "Design patterns for supporting rtsj component models," in *Proc. of the 7th international workshop on Java technologies for real-time and embedded systems (JTRES'09)*. ACM Press, 2009.

[12] D. F. Bacon, "Realtime garbage collection," *Queue*, vol. 5, no. 1, pp. 40–49, Feb. 2007. [Online]. Available: http://doi.acm.org/10.1145/1217256.1217268

[13] R. Pedersen and M. Schoeberl, "Exact roots for a real-time garbage collector," in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, ser. JTRES '06. New York, NY, USA: ACM, 2006, pp. 77–84. [Online]. Available: http://doi.acm.org/10.1145/1167999.1168013

[14] A. Wellings and A. Burns, "Asynchronous event handling and real-time threads in the real-time specification for java," in *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*, 2002, pp. 81 – 89.

[15] A. Wellings, A. Burns, O. dos Santos, and B. Brosgol, "Integrating priority inheritance algorithms in the real-time specification for java," in *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on*, may 2007, pp. 115 –123.

[16] "The real-time java platform. a technical white paper," Sun, Tech. Rep., 2004.

[17] "Sun java real-time system 2.2: Compilation guide," http://docs.oracle.com/javase/realtime/doc_2.2/release/JavaRTSCompilation.html, 2009.

[18] "Sun java real-time system 2.2: Garbage collector guide," http://docs.oracle.com/javase/realtime/doc_2.2/release/JavaRTSGarbageCollection.html, 2009.

[19] F. Siebert, "Hard real-time garbage collection in the jamaica virtual machine," in *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on*, 1999, pp. 96 –102.

[20] M. Fulton and M. Stoodley, "Compilation techniques for real-time java programs," in *Code Generation and Optimization, 2007. CGO '07. International Symposium on*, march 2007, pp. 221 –231.

[21] R. E. Jones and C. Ryder, "A study of java object demographics," in *Proceedings of the 7th international symposium on Memory management*, ser. ISMM '08. New York, NY, USA: ACM, 2008, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/1375634.1375652

[22] *JamaicaVM 6.1 User Manual: Java Technology for Critical Embedded Systems*, aicas GmbH, 2012.

[23] F. Siebert, "Realtime garbage collection in the jamaicavm 3.0," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, ser. JTRES '07. New York, NY, USA: ACM, 2007, pp. 94–103. [Online]. Available: http://doi.acm.org/10.1145/1288940.1288954

[24] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens, "On-the-fly garbage collection: an exercise in cooperation," *Commun. ACM*, vol. 21, no. 11, pp. 966–975, Nov. 1978. [Online]. Available: http://doi.acm.org/10.1145/359642.359655

[25] T. Harmon, M. Schoeberl, R. Kirner, and R. Klefstad, "Toward libraries for real-time java," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, may 2008, pp. 458 –462.

[26] "Javolution: Java library for real-time, embedded and high-performance applications," http://javolution.org.

[27] A. Wellings, "Multiprocessors and the real-time specification for java," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, may 2008, pp. 255 –261.

[28] J.-M. Dautelle, "Validating java™ for safety-critical applications," Raytheon Company, Tech. Rep.

[29] P. Dibble and E.-K. Lung, "Os platforms for rtsj," http://www.rtsj.org//docs/OSPlatforms.html, 2005.

[30] M. Dawson, "Challenges in implementing the real-time specification for java (rtsj) in a commercial real-time java virtual machine," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, may 2008, pp. 241 –247.

[31] E. Y.-S. Hu, E. Jenn, N. Valot, and A. Alonso, "Safety critical applications and hard real-time profile for java: a case study in avionics," in *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, ser. JTRES '06. New York, NY, USA: ACM, 2006, pp. 125–134. [Online]. Available: http://doi.acm.org/10.1145/1167999.1168021

[32] T. Kalibera and et a l., "Real-time java in space: Potential benefits and open challenges."

[33] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz, "Project golden gate: towards real-time java in space missions," in *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, may 2004, pp. 15 –22.

[34] S. G. Robertz, R. Henriksson, K. Nilsson, A. Blomdell, and I. Tarasov, "Using real-time java for industrial robot control," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, ser. JTRES '07. New York, NY, USA: ACM, 2007, pp. 104–110. [Online]. Available: http://doi.acm.org/10.1145/1288940.1288955

[35] N. Juillerat, S. M. Arisona, and S. Schubiger-Banz, "Real-time, low latency audio processing in java," in *International Computer Music Conference*. ICMC, 2007. [Online]. Available: http://diuf.unifr.ch/main/pai/sites/diuf.unifr.ch.main.pai/files/publications/2007_Juillerat_Mueller_Schubiger-Banz_Real_Time.pdf

[36] D. Hober and S. Elzer, "Does the real-time java specification form a viable real-time programming language?" 2008.