A Java Framework for Spatial Embedded Systems

L. Apvrille[†, ††, †††], P. Sénac[†, ††], M. Diaz[††]

[†] ENSICA, Ecole Nationale Supérieure d'Ingénieurs de Constructions Aéronautiques
Département Mathématiques appliquées et Informatique
31056 Toulouse Cedex 5, France
{apvrille, senac}@ensica.fr
+33.5.61.61.86.{87, 77}

[††] LAAS-CNRS
Laboratoire d'Analyse et d'Architecture des Systèmes
7, avenue du colonel Roche
31077 Toulouse Cedex 4, France
{lapvrill, senac, diaz}@laas.fr
+33.5.61.33.62.56

[†††] *This work has been partially supported by :*
Alcatel Space Industries
Service DI/IAG/T
26, av J.F. Champollion – BP 1187
31037 Toulouse Cedex, France
Ludovic.Apvrille@space.alcatel.fr
+33.5.34.35.49.88

Abstract - Satellite constellations will soon play a fundamental role in global broadband and heterogeneous communication architectures. These constellations will offer ubiquitous access to interactive multimedia services. Spatial embedded systems must evolve accordingly to provide a robust real time support, including a dynamically extensible runtime environment. In other words, these systems must allow new services or features to be dynamically loaded and linked to the embedded software. Object technology answers the need for dynamically extensible systems. But can it also answer the constraints of embedded real-time systems ? First experiments with Java indicate that an efficient real time scheduling of Java tasks can be implemented on top of a real-time kernel. The paper also brings solutions for an efficient and robust memory management
Thus, the paper shows that it is realistic, proficient and cost effective to use object technology for designing the new generation of real-time embedded systems.

Introduction

As communication needs increase, constellations of L.E.O. (Low Earth Orbit) satellites [14] definitely seem to be an efficient answer for the fast deployment of a global broadband telecommunication infrastructure. In the near future, global Internet connectivity will be available using small handset terminals that will connect mobile users to the nearest constellation satellite networks available. Thus, the selected satellite will either directly transmit information from a mobile terminal to a ground station connected to earth based networks and vice versa, or route it to a next satellite for constellations supporting ISL (Inter-Satellite Links).
Because of their high cost, the reliability and validity of these systems is a fundamental issue. Moreover, the constellations' estimated life duration combined to the steady evolution of ground networking technology, entails the need for integrating reconfiguration capabilities in telecommunication satellites. These features can greatly benefit from advanced high level software runtime environments which can be dynamically extended with new capabilities and that free software designers from hardware concerns. Ideally, such advanced software

environments should take charge of many functions traditionally operated by hardware or by low level hardware-dependent software.
The natural solution for such adaptive systems would be to encapsulate capabilities in autonomous modules that could be dynamically loaded into the system. This feature could be offered by object-oriented technology. However, object oriented technology has not been intensively experimented in the field of distributed hard real-time spatial embedded systems. Therefore, this paper studies and shows how object technology can be used to respect embedded spatial communication systems hard real-time constraints.
The paper is organized as follows: section 2 identifies the problems related to real-time communication systems which are embedded within satellites. Section 3 describes how the Java language can be used in such a context. Java's strengths and shortfalls are carefully analyzed within the general framework of embedded software constellations. Section 4 involves the discussion of experiments. The Java Virtual Machine used in our experiments, is located on top of a VxWorks Real-Time Operating System. The latter is expected to schedule Java tasks conforming to RMA or EDF policies. To decide whether this objective is met or not is made possible by analysis tools available with the RTOS. In practice, major problems were identified at the JVM garbage collector level. Section 5 discusses whether the necessary memory management adaptations should be implemented at the application level or within the underlying layers.
Finally, conclusions and future work directions are presented in Section 6.

How Object Technology Meets Spatial System Requirements

As the L.E.O. satellite visibility period is short and because of the low latency applications that aim to use the constellations' services, high level real-time networking services will have to be introduced in satellite constellations. These services will have to evolve according to ground needs, and so they should be dynamically configurable without any service interruption. Moreover, to respond shorter time delays, the dynamic system will also have hard real-time constraints: real-time configuration, real-time switching, real-time filtering, real-time routing and classical real-time satellite functionalities (antenna positioning, etc.). All these embedded real-time features will contribute to efficiently support interactive distributed multimedia applications.
Such real-time constraints have caused satellite manufacturers to use classical solutions to design telecommunication satellites. In order to resist to cosmic rays, embedded hardware systems are specific, thus are very expensive and perform poorly. Hence, only very basic functionalities are realized by the software, still using hardware components rather than software components.
A usual satellite application is composed of about 10 tasks: periodic tasks which periods are close to 1 second and asynchronous tasks with very short reaction time (less than 100 µs in response to a hardware interruption). Consequently, manufacturers have chosen to fully control their code and use a simple proprietary kernel or even a sequencer. The software part of the application is developed in assembler language for low level hardware access and in C, Ada 83 and 95.
Such specific choices are possible for a single GEO satellite but not for a whole constellation of satellites; new functionalities are hard to implement using embedded systems based on specific and hardware oriented technology. Moreover, very few development and test tools exist for such hardware platforms, reducing the reliability of the developed system. Using specific hardware entails the use of proprietary kernels. All the above mentioned problems deeply impact the methodology design of satellite constellations where cost and reliability problems are tenfold.
Designers of communication satellites therefore consider using standard technology. Standard hardware coupled with advanced fault-tolerant systems could efficiently support spatial constraints. These standard architectures allow commercial runtime kernels to be used. However, by using on-shelf technology, manufacturers have to cope with a rapidly evolving market, really not compatible with constellation software development and maintenance. A very pertinent response to this problem comes from an environment based on virtual software platforms which insure a good isolation between software and hardware. This technique potentially allows new satellites to benefit from hardware progress without impacting software. The dynamic extension of satellite's functionalities could be obtained using highly modular software environments that allow

software components to be dynamically downloaded. This consideration potentially makes object oriented technology a good candidate for the design of the new generation of communication satellites embedded software.
The combination of a hardware-independent platform and an object oriented environment leads us to define and experiment a Java framework. Indeed, the use of an independent hardware and object oriented software makes it possible to dynamically add new Java objects that replace old functionalities (a routing algorithm for example) or ensure new services. Moreover, in this new general framework, a global constellation can be considered as a real-time distributed system [1] on which mobile codes [9] (objects) can migrate from satellite to satellite for constellations supporting ISL or from satellite to ground control station.

Java's Strengths and Shortfalls for Embedded and Real-Time Applications

Java, a New Language

Java [10] is a brand new (1995) object-oriented language. Its syntax very close to C/C++ makes it easy of access for many programmers. But the main Java characteristic is its JVM (Java Virtual Machine) a software platform.
Java is a platform-independent language, first used mainly by applet developers, and originally not designed with real-time applications in mind. Nevertheless, many Java native features could be particularly useful for embedded distributed systems.

Java in Embedded Systems

First, Java is portable from one platform to another with a proprietary JVM without any Java bytecode (the Java assembler code) recompilation. Portability facilitates embedded system integration into an already developed heterogeneous system, just like an evolving constellation.
Second, Java is free of pointers. This dramatically increases the system's robustness. Moreover, a garbage collector periodically runs to erase unused objects from memory. The garbage collector also fights against memory fragmentation.
Third, Java permits C code integration (JNI mechanism) mainly for low-level functions, drivers and BSP (Board Support Package), as described in [4]. Other language code just like assembler code can also be called from the C code.
Finally, Java code can dynamically be loaded. This can serve fairly well the need for adaptive applications [9]. As the old Java code is garbaged, this ensures there is enough memory for new downloaded code as long as its memory footprint is not higher than the one of the previous code.

Java for Real-Time Applications

The Java language has also many interesting features for real-time applications.
First, the Java platform integrates its own scheduler. Thus, the Java API Thread class allows to program tasks that are scheduled by the Java runtime system. The internal scheduler works with fixed priority tasks. Each executing task runs until either a higher priority task is available for running (on preemptive systems), or until it yields or stops, or its time allotment expires (on time-slicing systems). For two tasks of the same priority, a round-robin system is applied.
The Java language also has thread objects synchronization capabilities.

Main Java Limitations for Spatial Real-Time Systems

Though Java has many great capabilities for real-time embedded applications and more specially for spatial real-time embedded applications, on the other side, it also has important limitations [7] now to be described.
First, the interpreted Java bytecode generally poorly performs compared to C. Hopefully, JIT (Just In Time) compilers can drastically increase Java's speed, getting it close to non optimized C++ compiled code. Many research groups including Sun [18] are currently working on Java performance improvements.
Secondly, it is not clear if the scheduling functionalities can handle hard real-time scheduling and short response time found in spatial applications. It should also be noticed that the Java runtime scheduler remains quite platform dependant.
Spatial applications need a perfect deterministic memory management. So usually, real-time spatial applications avoid dynamic memory allocation. With the fully object oriented JRE (Java Runtime Environment), such a politic

seems unsuitable. Moreover, the whole software platform (JRE + librairies) increases the system's memory footprint.
The last main problem in embedding Java is its poor hardware control API. Hence, Java spatial applications requires C code which impedes system portability [6] and full object applications.

Java on Top of a Real-Time Kernel

Java, as described above has real-time embedded advantages. But both scheduling and memory management could be obstacles for the use of standard Java in spatial application. These two limitations are not Java specific for such problems have already been mentioned in the general framework of object oriented technology [5]. Nevertheless, the object oriented technology and dynamic features of the language make it a potentially excellent solution for satellite constellations. Finding solutions to transform its shortfalls (real-time memory management, memory footprint, tasks scheduling) into strength is a task full of stakes.
Sun proposes two solutions for embedded systems: Personal Java [19] and Embedded Java [17]. Both are mainly composed of specific APIs and tools meant to reduce the software platform memory footprint. Both solutions require a Real-Time Operating System (RTOS). Another solution also proposed by Sun consists in executing Java code on a specific processor [12]. This solution has few chance of being used for spatial applications as it has never been proved to be cosmic rays resistant.
A commercial kernel associated with object technology has already been proved to be efficient for multimedia application systems support in distributed systems [3]. A perfect real-time scheduler could bring real-time scheduling guarantees to Java tasks. This should be done if the JRE, instead of being a layer between the Java application and the operating system was a module of the kernel, changing Java threads into first class citizens as other kernel threads (see Figure 1).

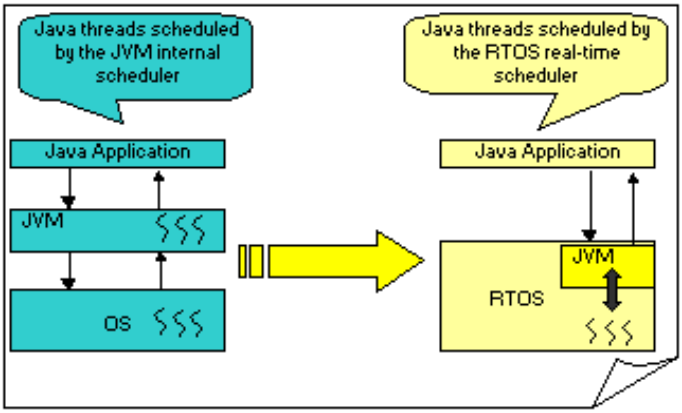

Figure 1 : a Java Platform Layer vs a Java Platform Module

The chosen RTOS should propose a fully modular Java compliant on-shelf solution. Among the widely used and supported real time kernels (LynxOS, pSOS, Chorus, VxWorks, …) and less classical real-time Java solutions (PERC [13] and jBED [8]), we have chosen to experiment with VxWorks because of its 100% Java certificate. Moreover, a whole Java solution that comes with this kernel, called Tornado For Java [21], gives functionalities close to those of Embedded Java. All the other VxWorks features (analysis tools etc.) also actively participate in fast development cycles.

A Java-based Experimentation Platform

The so settled spatial experimentation platform is composed of:

- a well-known hardware component (PowerPC) because of its potential good resistance to cosmic rays – with 8 Mb RAM.
- A trusty commercial real-time kernel (VxWorks).
- A high level object-oriented programming environment (Java).
- Development tools (crossed-development) and test tools made by third parties.

The whole solution refers to on-the-shelf technology (see Figure 2), from the hardware layer to the application layer which reduces system's cost and greatly increases system's reliability.
All the potential limitations previously described now have to be tested. First, very short response time and real-time task scheduling in the payload software is essential for the new generation of spatial broadband communication systems. Therefore, we have to prove first that it is possible with a real-time kernel to schedule Java tasks in order to respect hard real-time broadband communication constraints.

Experimenting Java Tasks Scheduling on VxWorks

Experimentation Methods

A two-step approach has been adopted for studying real-time Java tasks scheduling on top of a real-time kernel. The first step consists in using well known models to obtain results based on an analytical model. That model gives the ideal and optimum results that should be obtained (theoretical scheduling politics, theoretical utilization factor and so on). The second step is an experimental validation of the first one based on a realistic satellite simulation platform. The first test is composed of tasks compliant with the theoretical model, whereas the second test uses a set of tasks whose scheduling profile is similar to the one found in communication satellites.

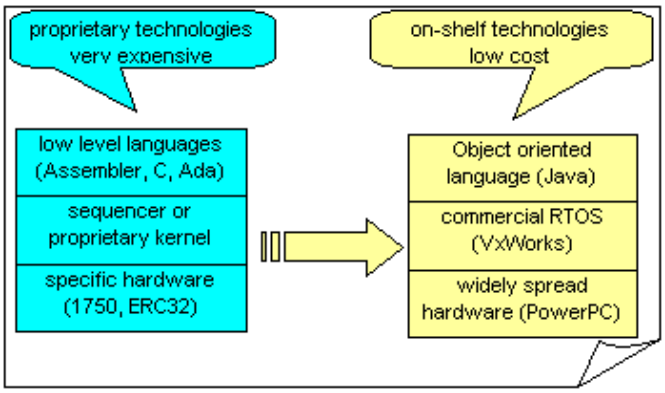

Figure 2 : Towards On-shelf Technologies

Testing Java tasks scheduling implies to use advanced test tools dealing with quite low time resolution. First a Tornado tool, WindView [20] performs graphical systems events analysis. Secondly, our own programmed tool allows fast scheduling politics analysis. Both tools use kernel instrumentation and Java task instrumentation. Third, an advanced instrumentation has been written in assembler language, using the timer register of the PowerPC. This method is specially designed for accurate context switching time measurement. All these tools used in a low priority mode were found to be minimally intrusive.

Experimentation of Analytical Models

Evaluating Java tasks scheduling on VxWorks first implies to prove that it is possible to define a scheduling model which Java tasks will always follow. Consequently, the test-bed should first demonstrate that every kind of disturbing elements in the system (for instance a fault happening in a task) would not modify the system so that critical deadlines could be missed. Secondly, some events answer time should be bounded. This issue entails that system's overhead and task switching time must be as low as possible and in any case be bounded. This leads us to consider the utilization factor as a mean for analyzing system overhead.

Two well-known and representative classes of real-time scheduling have been used: RMA (Rate Monotonic Algorithm), EDF (Earliest Deadline First) algorithm. As these scheduling algorithms are simple, they generate a very light overhead. With the chosen RMA implementation (static priorities are applied to tasks), the scheduling policy overhead is null, as once a priority is applied to a task, nothing else has to be done on system events (switching or else). Concerning the EDF algorithm implementation (dynamic priorities), the overhead is quite low: the test shows that on our tasks scheduling, the time used for reevaluating priorities is not really significant compared to the evaluated elements. In fact it is only significant when testing task switching time. So better accurate switching time are calculated only with the RM algorithm.

The utilization factor and scheduling policies are tested with the two algorithms.

Different kind of tasks are used for the test. The system is composed of 2 to 8 tasks, that perform basic calculation. Thus, they are independent from each other according to what the two algorithms assert. Two-tasks applications allowed accurate tests whereas 8-tasks applications are more representative of classical communication satellite applications. All these Java tasks are first modeled using the UML language [2]: they all inherit from the PeriodicTask class which itself extends the standard Thread class. Then, the tasks are programmed using the Java language. The scheduler is also considered as an application task and therefore extends the Java Thread class. The RMAScheduler class and EDFScheduler class inherit from the Scheduler class. So both schedulers and tasks are Java task objects. They are all scheduled by the real-time VxWorks scheduler. Thus, both RMA and EDF algorithms are Java programmed.

Three application elements must be evaluated: correct scheduling of tasks according to the chosen model, the switching time and the utilization factor.
The two scheduling policies have been tested using multiple scenarios. For each one, we check if the tasks are scheduled according to the theoretical model. As evoked previously, the system's overhead analysis is based on the utilization factor, denoted μ, which is given by the following formula (equation 1):

$$m = \sum_{i=1}^{i=n} \frac{c_i}{p_i} \leq 1 \qquad (1)$$

where :

- c_i is the maximal execution time of task T_i
- p_i is the period of task T_i
- n is the number of tasks

By minimizing the period p_i of each task, it is possible to increase the utilization factor. This can be done progressively until the system is no longer scheduled correctly. We assert that the maximum system utilization factor for this set of tasks according to the chosen model is reached when the decrease of any period, p_i, entails that the considered set of tasks cannot be correctly scheduled. Tools tell us whether the system is correctly scheduled. Therefore, by analyzing the system each time a period task is changed, generally lowered, the maximum utilization factor can be found.
We compared the performances of these 2 scheduling policies using the Solaris Kernel and VxWorks real-time kernel. This kind of comparison is hard to realize on two systems which are composed of different microprocessors, different JRE and of course different kernels. These differences imply that the same tasks don't last the same time on each system. However, by using the utilization factor experiment, the evaluation is easier to perform because the system's overhead measurement is not related to the global performance of the considered platform.
As expected, better results were obtained with VxWorks system than with Solaris.
The tests on VxWorks show that as long as a set of tasks can be scheduled according to the analytical model, the effective scheduling done by the real-time kernel will always work as

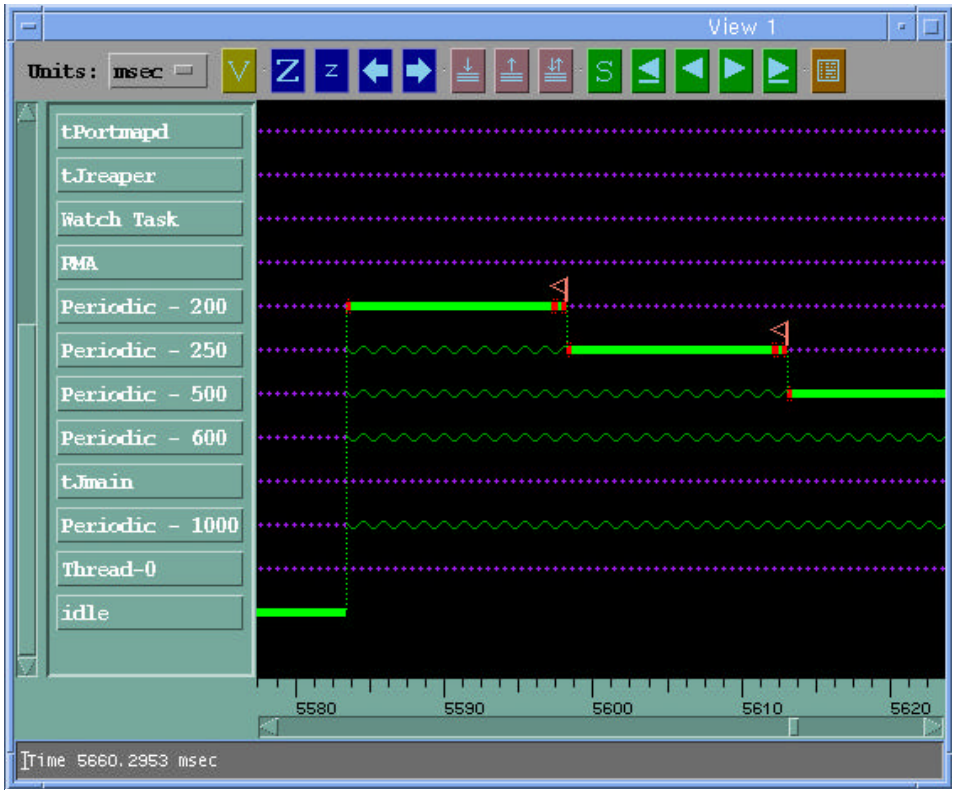

Figure 3 : Java Tasks Scheduled by the VxWorks Kernel

expected. Figure 3 shows the perfect Java task scheduling on the VxWorks system . Conversely, on the Solaris kernel, even when working with a minimal set of formally schedulable processes in a privileged mode, scheduling errors appear. These scheduling errors demonstrate the negative impact of system overhead on the scheduling algorithm.

Another test evaluated the switching time. On the Solaris system, the results were quite bad whereas on the VxWorks system the results obtained are compatible with hard real-time i.e. less than 100 µs.

The three curves (see Figure 4) depict the utilization factor experiment results.

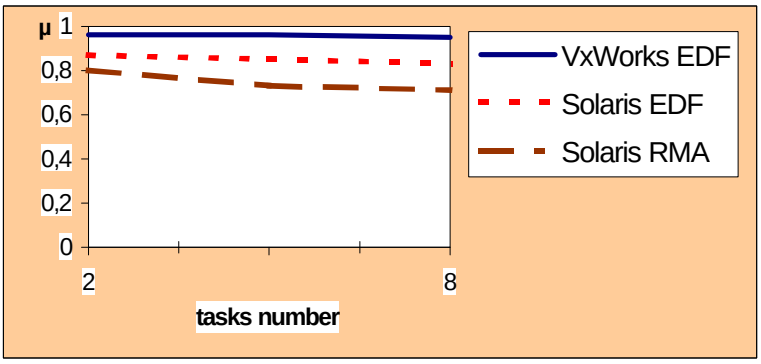

Figure 4 : Evolution of the Utilization Factor in Function of the Number of Tasks

- RMA utilization factor results under Solaris are identical to those of RMA VxWorks. They are in-fact close to the theoretical maximum, as a necessary condition of the RM algorithm is that the system is certain to be scheduled if (equation 2):

$$m = \sum_{i=1}^{i=n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1) \qquad (2)$$

  During the test, this assertion was almost always verified on both systems (Solaris, VxWorks). The obtained curve is, in fact closer to a line than the theoretical curve. Of course, on both systems, the set of tasks with a utilization factor really higher than $n(2^{1/n} - 1)$ are scheduled correctly (sufficient condition). But the curve shows the worst situation obtained.
- Secondly, we notice that the system performs better with the EDF algorithm on the VxWorks system than on the Solaris system. Moreover, it is more resistant to high number of tasks than the Solaris system, due partially to the fact that context switching time is higher on Solaris systems and so the overhead increases faster than on VxWorks systems, when the tasks number increases.

The results are, in fact, very good using the EDF model on VxWorks because the graph shows the worst case. But in most situations (other set of tasks), the utilization factor was higher than the two EDF curves show.

So, with the help of a formal approach, the tests permit us to conclude that the system is very efficient in scheduling Java tasks. Thus, the results demonstrate that scheduling Java tasks on top of a real-time kernel like VxWorks is as efficient as scheduling other classical tasks: scheduling Java tasks on a real-time kernel is compatible with standard hard real-time scheduling.

Experimentation First Results on a Large Scale Spatial Application

The first results need to be validated on a large scale application that could certify that the real-time scheduling is compatible not only with periodic tasks, but also with spatial tasks.

The test environment is composed of a Solaris workstation simulating a satellite ground station and the target system simulating a L.E.O satellite which communicates with several antennas precisely located on earth. The host and target communicate using an Ethernet link.

Our system is composed of two on-going tasks, one periodic task, and five sporadic tasks which are very close in job and time specification to other actual projects tasks. Another periodic task realizes fault injection and a sporadic one simulates the antennas by generating data

packets. As mentioned before, most of the tasks were either periodic or sporadic. Each task is assigned a static priority to reduce context switching time overheads.
Simulating this described system is very interesting as our first tests are in an abstract formal framework. Thus, it was quite outstanding to visualize hard real-time spatial tasks, programmed in Java, being correctly scheduled and answering in time to urgent injected events. This simulation platform enable us to confirm the first results. Moreover most of the Java synchronization capabilities are tested and demonstrate their high performance.
But an important noticeable limit to all these tests must be mentioned: the garbage collector. During all the tests it is disabled. Is its unpredictable behavior compatible with the perfect obtained real-time scheduling ?

Dealing with the Garbage Collector

As demonstrated before, scheduling real-time Java tasks is very efficient when the garbage collector is disabled. But deterministic scheduling cannot be obtained without deterministic memory management. How could we deal with both a perfect memory management and the JRE garbage collector ?
The Java classical "mark and sweep" garbage collector works in two different phases. First, it scans the memory finding unreferenced objects and marking them. Then it frees memory related to these objects and compacts it.
The Java garbage collector has two functioning modes. In the asynchronous mode (the default one) the garbage collector will start each time the system is idle, or whenever the programmer calls it explicitly, or whenever there is a request for memory and none is available. In the synchronous mode, the garbage will only start on explicit call or when there is a memory need.
The first mode is the most efficient one for classical applications as the programmer really oughtn't worry about freeing memory. This mode is mainly compatible with graphical user interfaces as the garbage collector has enough time to run between two inputs. But on real-time embedded applications with hard constraints, is there a better mode to use and are there garbage collector options or programming methods to achieve a deterministic dynamic memory management ?

First, tests are made to fully analyze how the collector works. Experimental results are obtained using the simulation platform previously described. A thread simulating communication functions enables us to experiment with an environment that dynamically creates and frees objects.
Better results are obtained on the VxWorks kernel than on Solaris: the VxWorks garbage collector starts every second according to the system's idleness percentage. Each time it runs for about the same time (60ms), scanning the memory during approximately 30 ms, and then sweeping and compacting the memory during approximately 30 ms. On the Solaris platform, we got non deterministic results.
Using the synchronous mode, the garbage collector doesn't start on both kernel without an explicit call to the System.gc() method. With explicit call, results obtained are the same as mentioned with the asynchronous mode.
Hence, those tests exhibit another important problem: the garbage collector cannot be preempted by other Java tasks during its work. More precisely, it can be preempted during the first phase (initialization and memory scanning) but not during the last 30 ms (sweeping and compacting) which correspond to a very critical job. Moreover, the garbage collector's run time is not theoretically bounded. Non Java tasks with a higher priority than the garbage collector can preempt it at any moment. But as portable spatial applications should be programmed only with Java tasks, solutions must be found to cope with this issue.
Two ideas could provide potential solutions to the memory management problem. The first one is a programming solution which consists in allocating memory only during an initialization phase. The second is to integrate the garbage collector's scheduling in the design of the real-time application.

In the first solution proposed, the main idea is to authorize dynamic allocations during a single phase, then run the collector and just after enter a mode in which garbage collection is prohibited. The different application states introduced by such an approach are described in Figure 5. In the initial state, the system has not been booted yet. From state 1 to state 2, the kernel boots and then starts the Java environment from state 2 to 3. Then the spatial application is ready to start. The first application task, called ReinitMemTask must allocate every object used further: tasks

objects, of course, and other data objects. Once

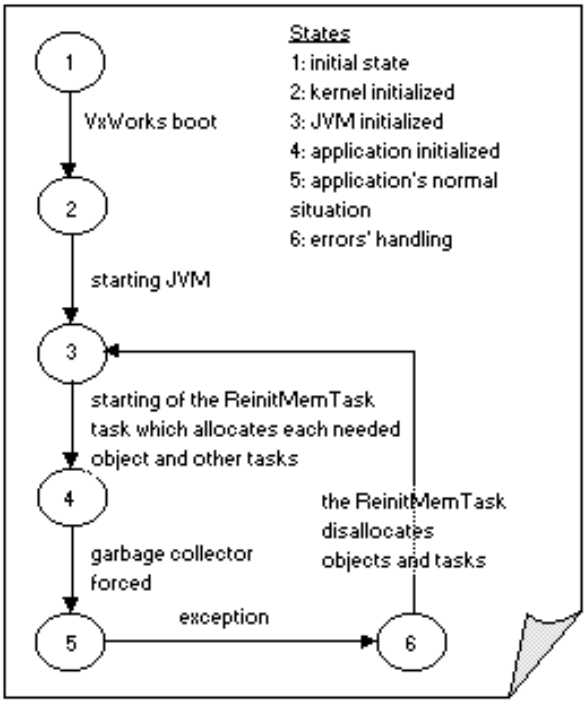

Figure 5 : Enforced Application States

the allocations are complete, the garbage collector is forced to run. Then the application can begin its normal work.

In state 5, memory allocation must not be made. In fact, more memory can be allocated but as the collector will not run, the application will only be able to work until memory is full. Obviously the most efficient way to avoid memory problem in this state is just not to use at all object allocations (notably the '*new*' operator).
If any exception occurs (memory management exception, memory fault or other), all the allocations are deleted (state 6) and the application is reinitialized in state 3. This service interruption lasts about 4 seconds (non optimized). Because of satellite visibility periods, actual manual reboots may last more than an hour ! Note that in both case all on board data are lost.
This method implies a perfect knowledge of the application code for programmers must be able to a priori know the allocated objects. Moreover, as the Java exception mechanism uses memory allocation (a Java exception is a object), its use must be severely limited to critical cases.
A first version of our embedded satellite environment has been programmed with all constraints mentioned above. Though they may appear very restrictive, they turned out to be quite easy to respect.
About data storage - our application simulates a communication routing node -, linked lists are programmed: objects are allocated in the first phase, just like arrays that represent all the needed memory. So, messages are stored in free objects (we mark our objects whether they are free or not using marked arrays) and added to free blocks in arrays. The linked list package programmed is quite easy to use and very efficient.
Naturally, this kind of programming method has limitations: it is not possible to program blindly (but has it already been possible ?). Thus, the programmer must pay particular attention to object allocations and to the use of the exception mechanism when dealing with pre-existent Java code (Java standard API for example).

The other main idea experimented to deal with the garbage collector, is to consider it as an application task. Indeed, during the first test:

- the garbage collector works as a real task of the VxWorks system with a low priority, except that it cannot be preempted during its critical part.
- Its periodicity and starting time can be finely tuned.
- Even if its duration cannot be theoretically bounded, it has never ran more than 70 ms and during half that time, it can be preempted. So on a one second periodic spatial application, urgent event could not be treated by Java tasks during 3.5% of the time. Of course, C tasks could deal with those urgent events.

Accepting those three assertions, the second solution allows dynamic allocations to be used at any time, as long as the collector considered as an application thread is periodically activated. Thus, from the delays obtained by performance measures on the garbage collector and by limiting urgent events to precise time intervals, classic Java applications using dynamic allocations could be programmed on top of a VxWorks kernel.

Both described solutions induce some limitations. The first one restricts the Java programming whereas the second one forbids the processing of urgent tasks during the activation periods of the garbage collector. Moreover, if some programming policy is adopted in the Java code, it seems difficult to define experimentally

the maximum bound for the duration of garbage collector. Therefore, it seems interesting to take the better of the two solutions: most allocations have to be made during an initialization phase and then the use of the garbage collector should be severely restricted during the second phase. Thus, the exception mechanism could be used.
Last, let's remark that both solutions were proved to be efficient and reliable enough to be used, which does not mean they cannot be improved ! Better solutions would probably be a mixture of the two, using a real-time garbage collector - garbage specific memory areas for example - which would permit to preempt it. Much work has already been made on real-time garbage collectors. Currently, no perfect solutions have been proposed. The one proposed in this paper has the advantage to have been specifically tested for spatial embedded system.

Conclusion

This study demonstrates that object technology and the Java environment can be used to ensure the real-time constraints needed for the new generation of broadband communication satellites are met. Indeed by analyzing the real-time scheduling behavior and finding better deterministic memory management solutions, we have shown that the general robustness and reliability of Java real-time embedded applications can be greatly improved. However, one must not forget that as the hardware used is not a-priori fully cosmic-ray resistant, advanced fault-tolerant mechanisms must be added to the system.
Thus, very simple fault-tolerance mechanisms have been tested on the simulation platform. A basic checksum mechanism which consists of using the Java exception mechanism to repair mistakes has been added to the memory management. Hence, in case of major errors, the application is unavailable for a maximum of 4 seconds only.
The general kernel stability and the application stability has also been tested by running it for several days and testing its capacity to the limit (sending wrong communication packets,...). No unexplained crashes occurred.
All these studies demonstrates that object technology is mature enough for real-time embedded systems. Of course, this simulation was only the first step of an approach which must now be validated in space. Future work will be dedicated to the study of dynamic configuration capabilities offered by this environment and its impact on the real-time scheduling and memory management. Such an approach would permit us to dynamically extend an embedded application while insuring a fully deterministic real-time behavior. Finally, we aim to combine our previous work in the area of real-time system specifications [15] [16], with Object Technology in order to provide software engineering methodology and related tools for designing object based distributed real-time systems.

References

1. J. Bates. CaberNet Report: "the state of the art in distributed and dependable computing", University of Cambridge, UK, June 1998
2. K. Bergner, A. Rausch, M. Sihling. "Using UML for modeling a distributed Java application". Technical report. Technische Universität München, Institut für informatik. TUM-I9735, July 1997
3. G. Blair et al. "A programming model and system infrastructure for real-time synchronization in distributed multimedia systems". IEEE journal on selected areas in communications, vol. 14, n°1, January 1996
4. M. Bunnell, "Mixing Java and C in Embedded Systems". Real-time Magazine, Internet Embedded – 98/1
5. C. Creel, B. Meyer. "Is object technology ready for the embedded world ?". JOOP, the Journal of Object-Oriented Programming, March/April 1998
6. P.C. Dibble. "Thoughts on the Java subset APIs". Real-time Magazine, Internet Embedded – 98/1
7. P.C. Dibble. "The reality of real-time Java". Computer design, August, 1998
8. Esmertec inc., Jbed Whitepaper, http://www.jbed.com/pages/products/rtos/whitepaper/index.html
9. A. Fuggetta, G. P. Picco, G. Vigna. "Understanding code mobility". IEEE transactions on software engineering, vol. 24, N°5, May 1998
10. J. Gosling, H. McGilton, "The Java Language Environment: a White Paper", Sun

Microsystems Inc., 1995. http://java.sun.com/whitePaper/java-whitepaper-1.html

11. R. Henriksson, “Predictable Automatic Memory Management for Embedded Systems”, OOPSLA’97 Workshop on Garbage Collection and Memory Management, Atlanta, Georgia, October, 1997

12. H. McGhan, M. O’Connor. “PicoJava: a direct execution engine for Java bytecode”. IEEE computer, 0018-9162/98

13. NewMonics Inc., PERC Product Information, http://www.newmonics.com/WebRoot/perc.info.html

14. T. Roussel, J.P. Taisant. “Optimizing space constellations for mobile satellite systems”. In JPL, proceedings of the third international mobile satellite conference, IMSC’93, p163-168.

15. P. Sénac, F. Fabre, E. Chaput, M. Diaz, “A Model and Toolkit for the Formal Specification of Weakly Synchronous Systems”, Proceedings of the IEEE Conference on Systems, Man and Cybernetics, Vancouver, Canada, October 1995.

16. P. Sénac, M. Diaz, E. Chaput, F. Fabre, P. de Saqui-Sannes - "Un modèle formel unificateur pour les systèmes temps réel" - Actes de RTS'96 (Real-Time Systems) - TEKNEA - Paris - Janvier 1996.

17. Sun Microsystems Inc., EmbeddedJava application environment, http://java.sun.com/products/embeddedjava/

18. Sun Microsystems Inc., HotSpot: The Java HotSpot virtual machine architecture, http://java.sun.com/products/hotspot/whitepaper.html

19. Sun Microsystems Inc., PersonalJava application environment, http://java.sun.com/products/personaljava/

20. WindRiver systems, WindView 2.0, http://www.windriver.com/products/html/windview2.html

21. WindRiver systems, Tornado for Java, http://www.windriver.com/embedweb/html/torn-java.html