IPP HURRAY!

www.hurray.isep.ipp.pt

Technical Report

Combining RTSJ with Fork/Join: A Priority-based Model

Cláudio Maia

Luís Nogueira

Luís Miguel Pinho

HURRAY-TR-110907

Version:

Date: 09-27-2011

Combining RTSJ with Fork/Join: A Priority-based Model

Cláudio Maia, Luís Nogueira, Luís Miguel Pinho

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

This paper discusses the increased need to support dynamic task-level parallelism in embedded real-time systems and proposes a Java framework that combines the Real-Time Specification for Java (RTSJ) with the Fork/Join (FJ) model, following a fixed priority-based scheduling scheme.

Our work intends to support parallel runtimes that will coexist with a wide range of other complex independently developed applications, without any previous knowledge about their real execution requirements, number of parallel sub-tasks, and when those sub-tasks will be generated.

Combining RTSJ with Fork/Join: A Priority-based Model

Cláudio Maia
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
crrm@isep.ipp.pt

Luís Nogueira
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
lmn@isep.ipp.pt

Luís Miguel Pinho
CISTER Research Centre
School of Engineering of the Polytechnic Institute of Porto
Porto, Portugal
lmp@isep.ipp.pt

ABSTRACT

This paper discusses the increased need to support dynamic task-level parallelism in embedded real-time systems and proposes a Java framework that combines the Real-Time Specification for Java (RTSJ) with the Fork/Join (FJ) model, following a fixed priority-based scheduling scheme.

Our work intends to support parallel runtimes that will coexist with a wide range of other complex independently developed applications, without any previous knowledge about their real execution requirements, number of parallel sub-tasks, and when those sub-tasks will be generated.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming — Parallel programming; D.2.11 [**Software Engineering**]: Software architectures; J.7 [**Computers in Other Systems**]: Real-Time

Keywords

Fork/Join, RTSJ, Parallel Systems, Real-Time Systems

1. INTRODUCTION

Nowadays, embedded systems are starting to incorporate multiple processors of the same architecture. The reason behind this paradigm shift lies on the reduced production costs and improved energy efficiency. This trend will become increasingly noticeable in the future as the tendency is to reduce even more the production/selling costs of this type of systems. Furthermore, embedded systems increasingly generate heavy workloads and it is rapidly becoming unreasonable to implement them as single core systems.

Besides the incorporation of multiple processors, embedded systems are also known by their stringent operation requirements, such as low memory footprint, low power consumption, and most of them inherently present timing constraints where a single deadline miss may pose consequences for the system under control.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JTRES 2011 September 26-28, 2011, York, UK
Copyright 2011 ACM 978-1-4503-0731-4I/11/09 ...$10.00.

Over the past few years, the real-time systems community has been addressing Java, through the adoption of the Real-Time Specification for Java (RTSJ) [4]. RTSJ is a real-time extension, in the form of an Application Programming Interface (API), to the Java standard platform which provides predictable execution to applications when executed in a real-time virtual machine (VM) environment. Generally speaking, Java is considered a successful programming language that is used to develop many applications nowadays. Much of the success is due to its advantages, from a software engineering perspective, over other programming languages and models, but also due to its open-source nature, platform-independence, applications' portability, and marketing hype.

Modern Operating Systems (OS) and Java VMs running on uniprocessor systems are multiprogrammed environments where applications execute concurrently in order to maximise the utilisation of system resources. With the current evolution of uniprocessor systems to multiprocessor systems, it is not sufficient to migrate or even adapt current sequential programming models or tools to these new multiprocessor systems. The penalty of doing so lies on the underutilisation of system resources. Furthermore, moving to multiprocessor systems is likely to fail if one does not consider a natural evolution of sequential programming models to new models of hardware and software that are naturally parallel [1].

Some of the scientific challenges associated with this evolution were already identified in [1] and [7], from where we highlight the following:

- Creation of new parallel programming models that efficiently take advantage of parallel platforms and architectures;
- Development of specific parallel data structures, algorithms and, code generation tools;
- Development of programming models that should be independent of the number of processors.

Although only focused on parallel computing, these challenges provide an important research direction that allows us to explore new programming models in order to combine parallel systems with embedded real-time systems. With this powerful combination, not only it is possible to create models that take advantage of the new parallel architectures in embedded real-time systems, but also it is possible to address several issues specifically arising from the joint relation between these two distinct scientific research domains.

We consider that current embedded real-time OSes and VM environments do not offer the necessary programming models and tools to handle the parallel execution of applications. Simply running sequential applications in these relatively new parallel architectures will not be advantageous as system resources will become underutilised. Moreover, the problem of maximising resource utilisation in these parallel embedded systems becomes even more serious when considering the increasing number of resources (i.e. processors). Thus, there exists an implicit requirement of efficiently managing system resources in order to increase the performance throughput of the applications.

In this work, we propose the efficient execution of multiprogrammed or parallel applications in parallel real-time embedded systems. For that, we propose the creation of a novel framework that handles the parallel execution of dynamic real-time applications with the objective of optimising resource utilisation. Furthermore, the applications may have timing requirements and are composed by a set of complex tasks that can be divided into smaller units of execution that are executed by the underlying architecture abstraction. The framework integrates RTSJ with the Fork/Join model and runs on top of a real-time Java virtual machine considering all interactions with the underlying OS.

Although the model presented in this paper is strongly focused on the well known fixed priority-based real-time scheduling scheme, the scheduling of parallel dynamic real-time applications using work-stealing as its basis is a still an open challenge. The main focus of the framework is to assure that resource utilisation is maximised while assuring the timeliness of the applications. For that, memory sharing between different worker threads needs also to be addressed.

The remainder of this paper is organised as follows. Section II presents the related work. Section III briefly describes the Fork/Join model. Section IV puts forward the system model considering a fixed-priority scheduling scheme based on the Java Fork/Join model for parallel embedded real-time systems. Finally, Section V concludes this paper.

2. RELATED WORK

This research work considers Java real-time VMs as the platform that is underneath the proposed framework, therefore it is useful to present some of the features that RTSJ adds to standard Java concerning real-time systems.

The approach followed by the RTSJ is to incorporate the notion of a schedulable object generalizing the concept of threads (i.e. RealtimeThread or AsyncEventHandler). In order to characterise a real-time task properly, these objects have as attributes the period or deadline, release time and priority, among others. As for the scheduling policy, RTSJ specifies a pre-emptive priority-based scheduler, with 28 priority levels supported, that guarantees that the schedulable object with the highest priority is the one that is executing. This scheduler can handle periodic tasks and sporadic tasks, i.e. asynchronous event handlers.

Memory management may be considered as one of the most important aspects that need attention in embedded systems, where the amount of resources is very limited. Thus, how memory is allocated is of extreme importance in embedded systems. Standard Java uses dynamic memory allocation to allocate objects in the heap and garbage collection to deallocate objects that are not in use by any of the running programs. Furthermore, memory management in standard Java is the cause of many application delays.

Real-time systems require all the operations to be bounded in time, including memory allocation and deallocation operations. Of major importance are the deallocation operations performed by the garbage collector which may impact the timing requirements imposed by real-time applications. In order to solve this problem, RTSJ handles memory management using a model based on the concept of memory regions. The following memory regions are considered by the specification:

- HeapMemory - Allows objects to allocate memory in the heap, as it happens in standard Java;
- ImmortalMemory - This memory region allows that all the memory allocated within it is never deallocated throughout the program execution. It is shared among all the program threads and it is deallocated when the program terminates, thus it is never subject to garbage collection;
- ScopedMemory - Is a memory region with a limited lifetime and specific assignment rules. Real-time threads may allocate objects in a memory scope and the objects will stay allocated until the scoped memory area is no longer active. At that time, all the objects in the memory scope are finalized and collected.

Besides the memory regions, RTSJ provides extensions that allow programs to directly access memory by using special classes. Also, it provides an object named NoHeapRealtimeThread, that is not allowed to allocate objects on the heap and therefore, is independent of garbage collection. Furthermore, the RTSJ garbage collector has the ability of being preemptable, meaning that when a higher priority thread arrives for execution, the garbage collector can be pre-empted leaving the system in a safe state.

Regarding synchronisation, standard Java suffers from the priority inversion problem as all mutual exclusion mechanisms. RTSJ requires the implementations to support the priority inheritance or priority ceiling protocols. An alternative mechanism to the implementation of such protocols is to use nonblocking communication, and for this purpose RTSJ specifies several classes.

Although RTSJ adds several new features to standard Java in order to allow for the required predictability and determinism of real-time systems, RTSJ has limitations in what it concerns to multiprocessor support. Andy Wellings discusses the limitations of RTSJ with regards to SMP systems in [12]. The objective of that work is to identify RTSJ limitations in order to include native support for multiprocessors in future versions of the specification. Some of the limitations that need to be addressed are stated in that work and include:

- Direct mapping of schedulable objects to processors. Currently, the Java Virtual Machine does not support this feature;
- RTSJ assumes a fixed priority scheduler with a single run queue per priority level. This needs to be adapted to scheduling schemes for multiprocessors: global, partitioned and hybrid scheduling;
- Temporal protection of groups of schedulable objects. RTSJ does not account for simultaneous execution of groups of schedulable objects;

- Interrupt handling per processor. This feature is advantageous because it allows managing critical tasks in a proper manner, avoiding task migration and therefore cache issues;

- Dynamic changes in the processor set. This is specifically important for parallel systems as it enables the proposed framework to adapt the algorithms to specific changes in the processor set.

Nevertheless, these limitations are taken from the point of view of applications running on a small number of cores, typically far less than the application threads. It is important to note that as the number of cores increases, threads do not increase accordingly, as threads are given by application requirements. Therefore, it is necessary to provide new runtime models that ease the parallelisation of applications.

Many of the current applications can be parallelised as a way to take advantage of the system resources and therefore maximise system performance. As parallel computing is regaining the attention of the scientific community due to the recent switch to parallel microprocessors, a group of Berkeley researchers gathered together to discuss the future of parallel systems across different disciplines. The results of the study are presented in [1]. This study presents seven questions that should be considered in order to find new solutions for parallel hardware and software. Among the important questions and recommendations presented in that work, we would like to highlight the need for new programming models, data structures and new methods for code generation.

Regarding the programming models and tools, there are already available a few of them, that specifically target parallel computing. Frameworks such as Cilk [3], Intel's Parallel Building Blocks [5], Java Fork/Join [6], OpenMP [10], [7] and Microsoft's Task Parallel Library [8] encourage programmers to divide their applications into parallel blocks. These parallel blocks will then be assigned to CPUs either by the frameworks or by the OS in a balanced manner.

To deal with the load balancing of parallel tasks, Blumofe and Leiserson proposed the work-stealing algorithm in [2]. This algorithm was proven to be optimal for scheduling fully-strict computations, therefore it is used in frameworks such as Cilk and Java Fork/Join.

3. FORK/JOIN MODEL

Frameworks such as Cilk, OpenMP and Java Fork/Join are based on the divide and conquer principle to solve many classes of problems. Basically, an application is broken up into subtasks that can be executed in a concurrent and independent manner in a set of processors. Applications start with a single thread that is *forked* into subtasks which will then be recursively split into new subtasks until they are small enough to be executed sequentially in each of the available processors. The *join* operation causes the current task to wait until the subtasks have completed [6].

Each of the above frameworks implements the Fork/Join model in their own way. OpenMP enables parallelism by extending serial code using compiler directives which do not affect the logical behaviour of serial program. All the parallelisation is handled by the compiler which knows which code to parallelise due to the directives added by the programmer [7]. On the other hand, Cilk extends the semantics of the C/C++ languages with extra keywords which will then be processed by the Cilk runtime that is responsible for managing all the aspects of parallelism. Java Fork/Join is a library that includes a set of useful methods that can be used to add parallelism to Java applications. Although it started as an independent library under JSR166 [11], most of the relevant features were included in the Java Development Kit version 7.0.

Java Fork/Join is a variant of the design used in Cilk, and according to its author [6], the advantages of a Fork/Join framework developed in Java over the common concurrency mechanism (java.lang.Thread or POSIX threads in which java threads are often based) rely on the synchronisation mechanisms and task granularities. Regarding synchronisation, subtasks only block when they finish their work and are waiting for other subtasks. As for granularities, regular threads are heavier to construct and allocate, and sometimes the cost of memory allocation surpasses the amount of work that they need to do.

Java Fork/Join, as well as Cilk, relies on work-stealing to schedule sub-tasks. As work-stealing has its own set of rules and because it is strongly present in our work, we present an overview of the rules for the sake of completeness. The work-stealing rules are the following [6]:

- There exists a worker thread with its own scheduling double-ended queue per CPU;

- Double-ended queues support LIFO and FIFO operations;

- Subtasks generated by tasks that belong to a given worker thread are pushed into that worker thread's double-ended queue;

- Worker threads process their own double-ended queues in a LIFO order;

- When a worker thread has no tasks to execute, it steals tasks from other worker threads' double-ended queues;

- When a join operation is issued, a worker thread processes other tasks until the target task is completed;

- When there is no work to do, a worker thread remains idle until new work arrives.

Work-stealing has the advantage of reducing task contention due to the support for LIFO - worker threads processing own tasks - and FIFO - other worker threads stealing from the opposite side of the queue. Furthermore, initial tasks generate more work than later generated tasks which affects the amount of stealing operations performed and task decompositions. Whenever a worker thread steals work from other worker threads, it steals work generated earlier (from other worker threads' double-ended queues following a FIFO policy) which allows initial generated tasks to be decomposed by means of parallelisation.

4. PROPOSED MODEL

This section presents a preliminary scheduling scheme that combines the Java Fork/Join model with RTSJ. The objective of creating such scheduling scheme is to take advantage of the new parallel architectures and allow for the computation of real-time tasks in a parallel manner in such a way that the system resources' utilisation is maximised.

We consider the scheduling of sporadic independent and dynamic real-time tasks on m identical processors, $p_1, p_2, ..., p_m$. The tasks belong to a task set τ. Each task τ_i of the task set releases a job j at sporadic time intervals and the exact execution requirements as well as its time arrival is only known at runtime. During the execution of the j^{th} job, the job may spawn a set of parallel jobs $p_{i,1}, p_{i,2}, ..., p_{i,n}$ at any time. Figure 1 illustrates this nomenclature through a set of geometric figures that may help in understanding the remaining parts of the model.

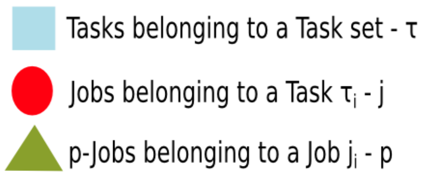
Tasks belonging to a Task set - τ
Jobs belonging to a Task τ_i - j
p-Jobs belonging to a Job j_i - p

Figure 1: Model's Legend

One should note that the Fork/Join model denotes parallel jobs as tasks. However, this definition clashes with the term task commonly used in the real-time systems domain to denote a scheduling entity, therefore, the term parallel jobs [9] is used instead in order to differentiate both entities.

Figure 2 presents a visual representation of the jobs belonging to a task set, and the spawning of the parallel jobs by each one of the jobs that is released for a specific task.

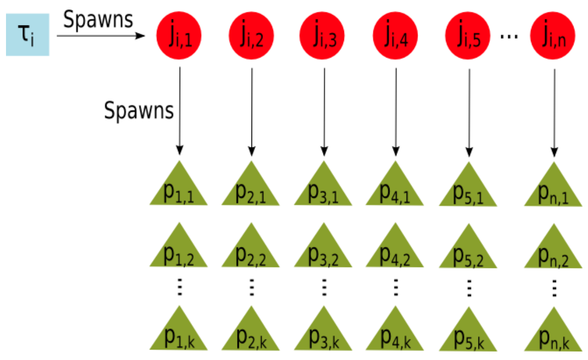

Figure 2: Parallel Jobs

The parallel jobs are work units that can be executed in different processors at the same time instant. This system model considers that each parallel job is totally independent from the remaining ones, meaning that with the exception of the processors, there are no other shared resources, critical sections or precedence constraints.

Each job that is ready to execute is scheduled according to its priority and therefore it is placed in a global submission queue, ordered by increasing priority as presented in Figure 3. It is important to mention that each parallel job will inherit the timing characteristics of the job that spawned it, i.e. period and priority. This can easily be seen in the figure.

A pool of worker threads is created at the beginning of application execution and each processor has an associated worker thread. The current model accounts for a 1-to-1 mapping, however more worker threads can be assigned to a processor. Worker threads are extended from RealTimeThread,

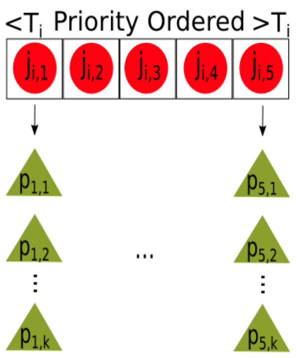

Figure 3: Scheduling of jobs using priorities

however it would be also potentially possible to inherit from NoHeapRealtimeThread instead, in order to avoid the garbage collection effects. Each worker thread is responsible for handling a double-ended queue and executing the parallel jobs that are generated by each job, as presented in Figure 4. It is important to mention that each worker thread will respect and apply the timing constraints of each job when executing its parallel jobs in order to assure that no deadline is missed.

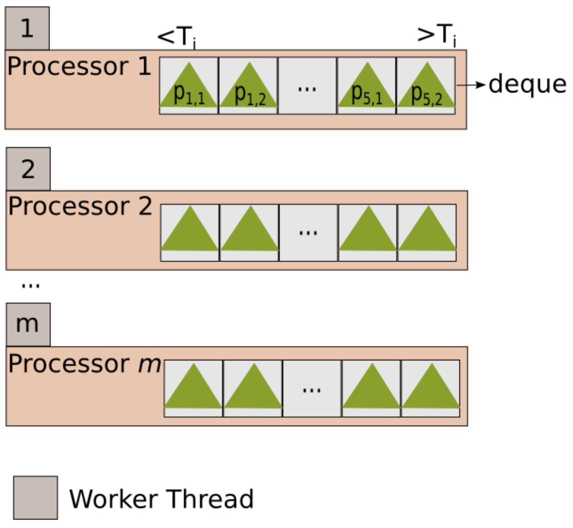

Figure 4: Mapping of parallel jobs to processors

In the beginning, the tasks that are already in the global submission queue will be directly mapped to the available processors in order to be executed. As time elapses and parallel jobs are generated, these are pushed into the double-ended queue belonging to that particular processor and not in the global submission queue. By following this approach, task contention in the global submission queue is avoided.

Another important characteristic associated with worker threads within the scope of Fork/Join model is the fact that they use a work-stealing policy to schedule parallel jobs. Each worker thread checks its own queue for parallel jobs until there are no more parallel jobs to execute. When there are no parallel jobs to execute, these worker threads steal the older parallel jobs in other threads' queues. If none is available, worker threads suspend.

In this standard approach, worker threads steal parallel jobs from other random threads, which means that priorities are not considered. Note that if this is changed and the stealing is performed from the thread where the parallel jobs have higher priorities, priority inversion may occur. A simple example is the following: assume that two cores execute two threads: t_m in core 1, of medium priority, and t_h in core 2 of high priority. Further assume that t_m generates parallel jobs which are obviously placed in core 1 double-ended queue.

If at some instant a new thread of high priority t_{h2} is ready, it will preempt t_m. The parallel jobs generated by this new thread will be also eventually placed in core 1's double-ended queue, but, according to the standard work-stealing algorithm, in the head of the queue, pushing older parallel jobs (of medium priority) to the end of the queue.

Now, if core 2 has no work to do, it may start stealing parallel jobs from core 1 queue. Since work-stealing works by stealing older parallel jobs, it will steal and execute those of medium priority. Note that this is equivalent to task t_{h2} being executed in core 1, and task t_m in core 2 (which is common in regular global scheduling systems), but in principle work-stealing would allow for task t_{h2} to be executed first in both processors in parallel. This is an example of priority inversion, but note that if no stealing would occur, core 1 would be idle.

In order to purely integrate the Fork/Join model with RTSJ, one needs to assure that the properties of work-stealing and the properties of real-time tasks are respected. This aspect should be handled carefully not only to assure the timeliness of the real-time tasks, through feasibility analysis, but also with respect to the impact of task migration and its effects on the real-time predictability of the system. This initial work considers a simple priority-based model, which can lead to priority inversion, only when a core would be otherwise idle, but that has the advantage of requiring only few adaptations to both the RTSJ and the work-stealing approach. However, more complex schemes are also being considered, which can lead to higher utilisation and flexibility [9].

From the memory management point of view, threads need to share their own queues in order to make it possible for other threads to steal tasks from them. In order to avoid the effects of garbage collection, it seems logic that each worker thread should have its own memory region. However, this cannot easily be done due to the imposed memory scope assignment rules. To solve this problem, memory portals may be used to share parallel jobs between different worker threads.

Besides all the above integration issues, it is necessary to consider the ones that were already covered in [12], among other methods, it is necessary to include methods that allows the schedulable objects to be pinned to processors (i.e. setAffinity()).

5. CONCLUSION

For fully utilising the parallel abilities of multicore platforms, real-time systems should be able to support tasks that may be executed on different cores at the same time instant. In this paper, we have presented a novel framework to tackle this problem in embedded real-time systems. This framework combines RTSJ with the Fork/Join model to schedule independent and dynamic parallel real-time tasks in Java-based systems. The proposed system model takes advantage of the existing processors in the system so that real-time jobs are divided into smaller units of execution, the parallel jobs, which are executed by a set of worker threads. Worker threads are mapped to processors and respect the timing constraints of the parallel jobs. Furthermore, by applying work-stealing, the framework is taking advantage of the spare resources existing in the system.

Concerning the present work, two important aspects are being taken into account: the definition and specification of scheduling algorithms based on work-stealing, that consider real-time parallel jobs' constraints; and the definition and specification of the memory model for handling and sharing parallel jobs among the work-stealing double-ended queues.

Acknowledgements

This work was partially supported by FCT through project RECOMP (Artemis 100202) and by the European Commission through the ArtistDesign NoE (IST-FP7-214373).

6. REFERENCES

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, May 1998.

[4] J. Gosling and G. Bollella. *The Real-Time Specification for Java.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[5] Intel Corporation. Parallel building blocks. `http://software.intel.com/en-us/articles/intel-parallel-building-blocks/`, June 2011.

[6] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[7] A. Marowka. Parallel computing on any desktop. *Commun. ACM*, 50:74–78, September 2007.

[8] Microsoft Corporation. Task parallel library. `http://msdn.microsoft.com/en-us/library/dd460717.aspx`, June 2011.

[9] L. Nogueira and L. M. Pinho. Supporting parallelism in server-based multiprocessor systems. In *The 31st IEEE Real-Time Systems Symposium (RTSS) Work-in-progress session*, San Diego, CA, USA, November 2010.

[10] OpenMP. Openmp. `http://openmp.org/`, June 2011.

[11] Oracle. Jsr 166: Concurrency utilities. `http://www.jcp.org/en/jsr/detail?id=166`, June 2011.

[12] A. Wellings. Multiprocessors and the real-time specification for java. In *Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*, pages 255–261, Washington, DC, USA, 2008. IEEE Computer Society.