Running Real Time Distributed Simulations under Linux and CERTI

Bruno d'Ausbourg
Pierre Siron
Eric Noulard
ONERA/DTIM
2 avenue E. Belin
31055 Toulouse Cedex
France
bruno.d.ausbourg@onera.fr, eric.noulard@onera.fr, pierre.siron@onera.fr

Pierre Siron
Université de Toulouse, ISAE
10 avenue E. Belin
31055 Toulouse Cedex
France
pierre.siron@isae.fr

Keywords:
Distributed simulation, real time, RTI, satellite.

ABSTRACT: *This paper presents some experiments and some results to enforce real time distributed simulations in accordance with the High Level Architecture (HLA). Simulations were run by using CERTI, an open source middleware, as the Run Time Infrastructure (RTI). Models were distributed over computers under various available versions of the 2.6 Linux kernel. Studies and experiments relied on a real case study. The chosen case study was the simulation of an "in formation" flight of observation satellites. This case study brings up some real applicative needs in real time distributed simulations and real configurations of simulators and models. Two simulations of "in formation" flight of satellites were studied. The study consisted in modeling the behaviour of the simulators and in running these models by using various kernel or middleware operating mechanisms and services. Time measurements were performed at each test giving some results on the ability of the simulation to meet its real time requirements.*

1. Introduction

New systems become more and more complex. They are made of numerous components. These components often make use of new technologies and interact between themselves. These systems can be embedded inside aircrafts (manned or unmanned), satellites or defense systems. Mastering and managing the development and the evaluation of such systems become a really difficult task. Simulation is more and more required to bring some help in these processes. Because many scientific and technical problems are addressed and because numerous models are needed to treat these problems, simulations are actually generally distributed. Models and simulations must interoperate in order to build some relevant results. So simulations must rely on some basic mechanisms and services to properly interoperate. Moreover, some hardware equipments or some real subsystems can be integrated inside the loop of these simulations. In that case, hard real time constraints must be taken into account when running these distributed simulations.

There is clearly a difficulty in satisfying hard real time constraints in interoperable distributed simulations. Satisfying these constraints may depend on some basic mechanisms and services that are implemented inside operating systems, inside kernels or inside dedicated middleware. Then, the question may be the following: is it possible to enforce real time distributed simulations by using common operating systems and run time infrastructures ?

This paper presents some experiments and some results to enforce real time distributed simulations in accordance with the High Level Architecture (HLA) (see [1], [2]). Simulations were run by using CERTI (see [3], [4]) as the Run Time Infrastructure (RTI). Models were distributed over computers that were running under various available versions of the 2.6 Linux kernel.

2. Case studies

The simulations that were used as case studies have their own history. And this history can explain some choices that were done with respect to their

architecture. At the beginning, old computers were used. These computers could only run the simulator of the satellite on-board computer because of performance and memory space requirements. Therefore a second computer was needed to run a model of the dynamics of the satellite and to run a model of its environment. So, in fact, a distributed simulation architecture was yet extensively used to design and to devise new satellites. To study the flight in formation of satellites, it seemed quite natural to duplicate these existing components in order to ensure some cost reduction. And the original simulation architecture was reused and incorporated in the design of the new simulation architecture.

2.1 A simple simulation

The first case study is a simulation (see [5]) that is made up by four components that are four simulators as depicted by figure 1:

- A simulator of the board computer on satellite 1.
- A simulator of the board computer on satellite 2.
- A simulator of the dynamics of the satellite 1.
- A simulator of the dynamics of the satellite 2.

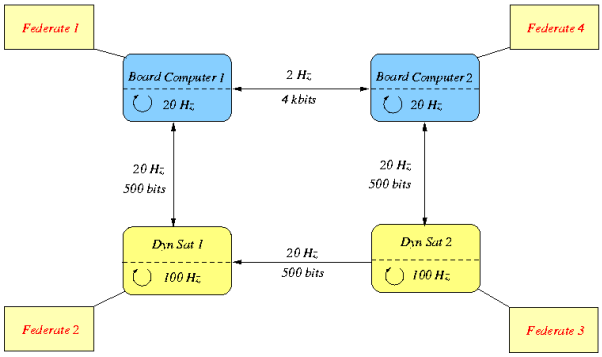

Figure 1: Structure of the first application case study

Each simulator is encoded by a federate in the final HLA simulation. Each of them runs a simulation loop at a frequency that is given on the figure. The data exchanges are also depicted on the figure by their size and their frequency.

A classical structure of exchanges is used between the simulators (see figure 2): the sender emits data at the end of a simulation cycle, at a logical time δ and the receiver receives data at the beginning of a new cycle, at a logical time δ'>δ. Exchanges between federates 3 and 2 on the figure 1 (between the dynamics simulators of satellite 2 and 1) have a different structure: data are emitted by federate 3 at the end of a simulation cycle, at a logical time δ, and are received by federate 2 in the middle of a simulation cycle at the same logical time δ. These exchanges are very useful to compute the relative positions of the two satellites in the real world. Modeling this structure of exchanges, under HLA, requires to invoke particular asynchronous delivery services of the RTI. And the experiments show that this particularity has a great influence on the performance results and on the ability of the whole simulation to meet its objectives in time.

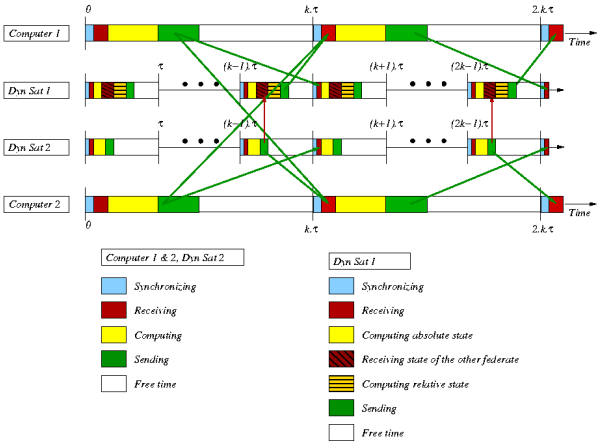

Figure 2: Exchanges of data between federates

2.2 A more complex simulation

The second simulation application incorporates the components of the first one and is a bit more complex because two other federates are added to the first simple simulation as depicted by figure 3:

- Federate 5 implements a simulation model of the payload computer at 200 Hz.
- Federate 6 implements a simulation model of the payload components at 200 Hz.

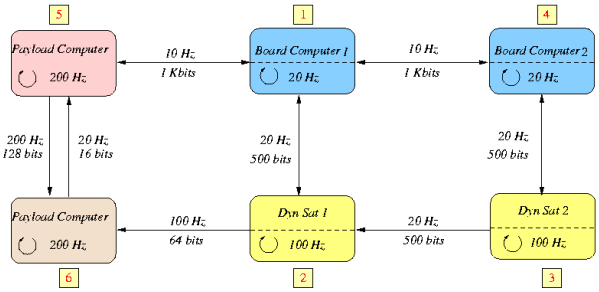

Figure 3: Structure of the second simulation case study

3. Modeling

Two kinds of federate models were developed. Each of them used a different mechanism to ensure time progression and synchronization inside the simulation. The first kind of modeling does not make use of any time management mechanism and

produces so-called HLA Real Time federates. The second kind of modeling makes use of the HLA Time Management services (see [6] and [7] for example). The programming of these two different models leads to two kinds of logical structure of their internal simulation loops. A federation includes only federates of the same type.

3.1 First type of simulation loop

Figure 4 depicts the logical structure of the HLA Real Time simulation loop. The structure of this loop is mapped on the sequence of steps that are performed by each simulator on each cycle. Every federate but federate 2 runs, inside the loop, the five following steps:

1. Synchronization phase.
2. Receipt of measured or command data from other federates.
3. Computation (25 % of a cycle).
4. Update of measures, commands or state data to other federates.
5. Free time.

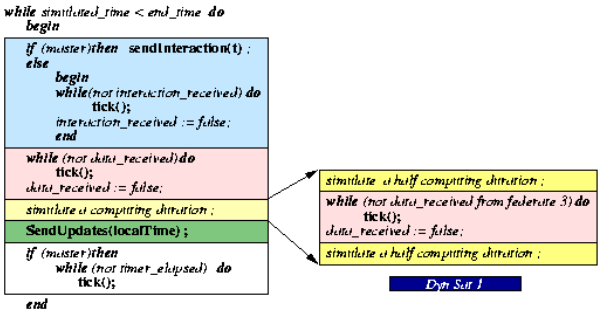

Figure 4: Real time simulation loop

The computation step is a bit particular when processed by federate 2. This step incorporates the receipt of state data from federate 3. This is depicted and detailed by the right side of figure 4.

The federate that beats at the highest frequency (with the smallest cycle duration) is considered as the master of the synchronization process : for example, that is the case of federate 2 in figure 1. At each beginning of a cycle, the federate emits an HLA interaction that acts as a global synchronization signal. This signal gives to other federates, that have subscribed this interaction, the information that the master federate is starting a new cycle. The other federates wait for these signals to make their own progress. There is no more synchronization constraint. So, some jitters may be observed when a federate remains idle for a while. This federate may gather and accumulate interaction signals from the master federate. And when becoming busy, the federate may speed up by consuming all these signals one after another. So it can be observed that some federates slow down and then speed up when running their simulation loops.

3.2 Second type of simulation loop

Figure 5 depicts the structure of the simulation loop when time management services are used by the federates. In this case, the synchronization does not need explicit receipt of interaction messages.

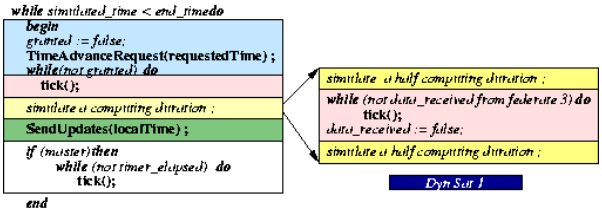

Figure 5: Coordinated time simulation loop

Time stamped messages are automatically delivered by the RTI while the federate is waiting to be time granted (when it is running the *while (not granted)* loop on figure 5). So, in order to receive an updated value while the federate is in an other state, as this is required by federate 2, these values must be carried by messages that are not time stamped (HLA Receive Order). These RO messages must have been previously authorized to be delivered in an asynchronous way.

3.3 Using a timer

A timer is used by the most time constrained federate (the master federate). This timer is programmed to be triggered at the end the theoretical cycle duration. The master federate, at the end of each cycle, simulates a free time by running padding instructions. So, when the timer triggers, two cases are possible:

- The cycle is achieved and the federate is in a free time state, waiting the cycle time has elapsed. In that case all is right and a new cycle can start.
- The cycle is not terminated and is yet in progress. In that case, the cycle duration is too long and this fact is registered in statistics that keep some information about the simulation behaviour.

4. Basic experiments

4.1 Hardware and software configuration

Our experiments were performed on a hardware and software reference configuration that was made up by COTS components. The hardware configuration is very simple and very common. It is based on two dual-core processors in *PowerEdge 860* machines.

Processors are *Dual Core Xeon 340* processors at 1,86 GHz with 2 MB of cache memory. Each of them can make use of 2 GB RAM memory. These two machines are linked with a dedicated 1 Gb link.

All experiments were based on the use of a *Linux Fedora Core 6* system in a 32 bits version. Different CERTI versions were also used.

4.2 First results on a basic single machine

Table 1 gives ten results by the simple simulation case study and by the first type of simulation loop. These results are with respect to federate 2. In this simulation, the targeted rate of federate 2 is 100 Hz and federate 2 is the master federate of the federation. In other words, simulation cycles must keep a duration of 10 ms. And simulation cycles may not be less than 10 ms because the programmed timer triggers after 10 ms.

Min Cycle Duration	Max Cycle Duration	Mean Cycle Duration	Standard deviation	Median
10.820	7243	100.67	129.65	12.004
10.869	19678.55	105.782	129.924	12.005
10.874	8922.58	83.659	111.068	12.002
10.879	3244.7	66.9	91.61	11.990
10.831	3071.89	82.97	103.85	12.001
10.392	3643.56	86.161	96.281	12.002
10.733	3738.055	82.182	1002.524	12.002
10.878	12228.395	92.118	125.461	12.001
10.790	13276.98	102.748	137.04	11.991
10.404	12733.412	103.624	135.689	11.996

Table 1: Simple case study, results of federate 2

The column on the left of the table shows that minimum durations of cycles are close to 10 ms. This fact demonstrates that the time objective can be reached. The problem is with maxima. Maximum cycle durations are too long : the worst case reaches 20 seconds (in red on the table). Moreover, these maximum duration values are fully distributed around mean values that are themselves always too long. So standard deviations are very important and this fact indicates a big jitter in cycles. This simulation does never meet its objectives in performances and is very irregular. A surprising fact is in medians. In every experiments, median values are in the interval [11.99..12]. This seems to indicate that performance is not really the problem: the problem is rather in allocating processors at the right time. The results for the other federates are not better and very irregular.

The complex simulation described in section 2.2 can not be run on the basic hardware and software configuration. In fact more than 2 hours were necessary to run 4000 cycles of 5 ms. In other words, what would have been done in 20 seconds was not achieved in two hours.

5. New real time mechanisms and results

The conclusion of the previous section is that a lot of work was necessary to improve the real time performances of these distributed simulations. The following presents and describe some different aspects of this incremental work. These aspects concern all the parts of the distributed simulations and even the applications, they are in fact very interleaved.

5.1 A more real time operating system

Setting a Real Time scheduling. Section 4 showed that a main difficulty, when attempting to master real time constraints, is to allocate processors to processes in a correct way. New experiments were performed by using two allocating mechanisms : a *static* mechanism and a *dynamic* one. These mechanisms are founded on real time resources that were added to Linux kernels.

The *static* mechanism uses affinity masks and the *taskset* command. This command is used to launch a new command with a given CPU affinity. CPU affinity is a scheduler property that *bonds* a process to a given set of CPUs on the system. The Linux scheduler will honor the given CPU affinity and the process will not run any other CPUs.

The *dynamic* mechanism is founded on the choice of real time scheduling algorithms for POSIX/Linux. Two real time algorithms, SCHED_FIFO and SCHED_RR, are intended for time-critical applications that need precise control over the way in which runnable processes are selected for execution. Only processes with superuser privileges can get a static priority higher than 0 and can therefore be scheduled under SCHED_FIFO or SCHED_RR. All scheduling is preemptive: if a process with a higher static priority gets ready to run, the current process will be preempted and returned into its waiting list. The scheduling policy determines only the ordering within the list of runnable processes with equal static priority. With SCHED_RR (Round Robin) each process is only allowed to run for a maximum time quantum. If a SCHED_RR process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority. A SCHED_RR process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum.

Locking memory pages. The *mlockall* system call disables paging for all pages mapped into the address space of the calling process. This includes the pages of the code, data and stack segment, as well as shared libraries, user space and kernel data, shared memory

and memory mapped files. All mapped pages are guaranteed to be resident in RAM when the *mlockall* system call returns successfully and they are guaranteed to stay in RAM until the pages are unlocked again or until the process terminates or starts another program. Real time applications require deterministic timing, and, like scheduling, paging is one major cause of unexpected program execution delays.

Real time timers in a new preemptible kernel. There is also a problem with timers. In standard versions of Linux, timers have a resolution that is given by the kernel basic frequency: 1000 Hz. In other words, a timer in Linux has a resolution of 1 ms. So time measurements on cycles of 5ms are not really accurate. In order to get more accurate timers, high resolution timers were introduced in Linux kernels (see [8]). These new timers permit a resolution of 1 µs.

Moreover, kernel is made preemptible. In that case, a process in kernel mode may be interrupted. In our first experiments, the kernel was not preemptible. In that case, when a federate executes a system call, the execution of the kernel code can not be interrupted. With a preemptible kernel (see [9]) the execution of the kernel code may be interrupted if something more important needs to run. So, the kernel is more reactive.

The process scheduler has been rewritten in the 2.6 kernel to eliminate the slow algorithms of previous versions. Formerly, in order to decide which task should run next, the scheduler had to look at each ready task and make a computation to determine that task's relative importance. After that all computations were made, the task with the highest score would be chosen. Because the time required for this algorithm varied with the number of tasks, complex multitasking applications suffered from slow scheduling. The scheduler in Linux 2.6 no longer scans all tasks every time. Instead, when a task becomes ready to run, it is sorted into position on a queue, called the current queue. Then, when the scheduler runs, it chooses the task at the most favorable position in the queue. When the task is running, it is given a time slice, or a period of time in which it may use the processor, before it has to give way to another thread. When its time slice has expired, the task is moved to another queue, called the expired queue. The task is sorted into this expired queue according to its priority. As a result, scheduling is done in a constant amount of time. This new procedure is substantially faster than the old one, and it works equally as well whether there are many tasks or only a few in queue. This new scheduler , due to I. Molnar, is called the O(1) scheduler (see [10]).

5.2 New execution configuration

The new experiments that will run the complex simulation use the configuration that is depicted in figure 6. By using mechanisms described in the previous section, federates that have the highest rate (federates 5 and 6 at 200 Hz) have respectively cores 0 and 1 at their own disposal on the second machine. The central kernel part of CERTI (RTIG) resides on core 1 of the first machine while federates 1, 2, 3 and 4 share the core 1 of the first machine. Due to a lack of available resources, we cannot study the impact of a greater number of processors.

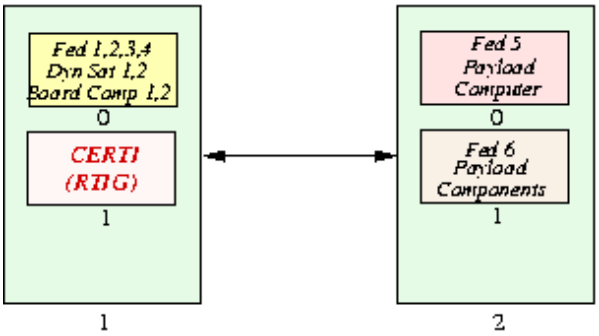

Figure 6: New execution configuration

5.3 New CERTI mechanisms

New tick function in the CERTI middleware. When running, federates can dynamically give the processor resource to the Run Time Infrastructure (CERTI) by using the *tick* CERTI call. So the RTI can launch, in response to received messages, the callback functions that are defined inside the federate program and that are associated to these messages. The problem is that, in early versions of CERTI, function *tick* was not blocking. It immediately returned when the RTI could not launch any callback in return, or it returned after having launched a particular callback. Using this function was done, in federate programs, by writing such blocks of instructions :

```
granted = false;
//require time advance
timeAdvanceRequest (timerequested);
//busy waiting of grant by RTI
while (! granted) do tick();
```

The callback function, launched by the RTI when, in this example, time advanced can be granted to the federate, assigns a value *true* to *granted*. This is done while the federate enters a busy waiting loop. This loop generates, on each *tick* call, exchanges of

messages between the federate and the RTI. It generates also useless context switches between these two processes. So the processor resource may be only used by these only two processes : this may seriously disrupt other processes and other federates.

To avoid such a lock of the processor, the function *tick* was reimplemented in a blocking mode. In other words, this function now returns only after a callback function has been launched by the RTI. Structure of programming is syntactically the same, but semantically, things are very different because only a few messages are generated and only two context switches are involved. This makes the processor free to be used by many other processes as long as it is not possible to return from *tick*.

Asynchronous message delivery. Moreover, a new mechanism to treat asynchronous message delivery was also introduced in CERTI. By this way, when a federate makes use of time management services of the RTI, messages that are not time stamped can be delivered by the RTI even if the federate is not in a time advancing state. By this way, asynchronous delivery of messages between federates 2 and 3 (Dyn Sat 1 and Dyn Sat 2) can be modeled and treated in a simulation that makes use of time management services. This work is a part of a more general working plan, the goal of which is to make CERTI more complete with regards to the 1.3 specifications of HLA in a first phase and to the new IEEE specifications in a second phase.

5.4 New programming of the federate user code

To improve the execution of simulations loops, I/O system calls were systematically eliminated in the user code of these loops inside the federate programs. The only calls that were kept are calls to the RTI services. These services may perform such system calls (by using sockets API for instance), but the federate programs do not.

Time measurements are also performed without doing any system calls. A direct access to the Time Stamp Counter of the processor is performed by making use of a **RDTSC** instruction in programs. Time computations on measurements were also optimized inside loops.

By this way, the code of programs in simulation loops is, essentially and only, devoted to the simulation processing.

5.5 Final results

Under these assumptions, many experiments were performed. In particular, HLA real time scheme and HLA Time management scheme were experimented. An other parameter seems to play a significant role in the global mastering of jitters: the priority that is assigned to federates. Experiments were performed by using the two simulation case studies. In this paper, only the most significant results are presented. They are related to the most complex and to the most constrained federate of this federation. So figures 7 and 8 present results of the master federate of the simulation (federate 5, Payload Computer). They give the maximum and median values for simulation cycles when simulations are run in low or high priority and by making use of time management (HLA TM) services or not (HLA RT).

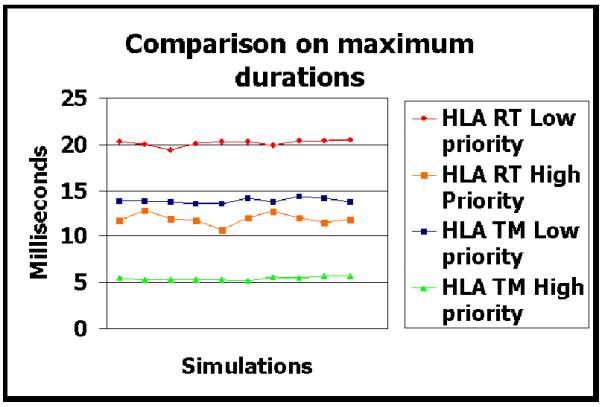

Figure 7: Federate 5 (200 Hz)

Figure 7 shows that maximum durations become lower when increasing priority and when using time management services. In particular when using time management services with a high priority, maximum values of cycles durations remain close to the expected duration of 5ms.

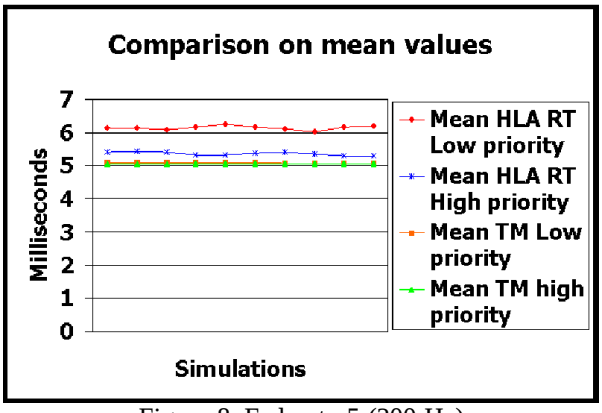

Figure 8: Federate 5 (200 Hz)

Figure 8 shows that the time objectives were finally met when the original results, at the beginning of experiments, were so bad. Here the maximal deviation is now less than 50 µs. With respect to mean values, priority is not the main determining

factor but rather the use of time management services that improves the global performance of the whole simulation.

Experiments that were done by making use of HLA Time Management services are very instructive. Making use of time management services implies that a distributed algorithm is enforced by each RTI part of federates in order to decide when advance time requests can be granted. A computational overhead is so generated when using these services. But experiments show that, globally, performances are in fact always improved.

6. Conclusions and learned lessons

Several lessons were learned while running these various experiments. First of all, it must be considered that all results were gained after modeling ``worst cases''. Assumptions that were done and basic mechanisms that were used to implement models were particularly pessimistic:

- The computation load in simulation cycles is heavy: 25% of the cycle duration is devoted to simulate the computational activity of the real simulator cycle.
- Computational and free times are hard encoded inside models: so these steps necessitate to have processors at one's disposal in order to progress.
- Item distribution of the simulation is minimal due to the very simple and very poor simulation architecture: two COTS dual-core machines.
- The synchronization scheme, when federates do not use time management services and are when they run under the HLA Real Time scheme, is very light. This lack of synchronization strengthens jitter possibilities.

Reducing some of these constraints would permit to gain better results and to meet more properly the real time objectives in simulations. But because the experiments were done on worst cases, learned lessons are so very instructive.

Don't confuse real time and performances. All experiments were done on the same hardware platform and by using the same machines and processors. This architecture is a very common one and not especially known as a particularly high-performance one. Spectacular results have been obtained with regard to runs of federates that make up the complex simulation. These federates did not properly run in the first experiments. They ran by being close to their real time objectives in the final experiments. In both cases the same architecture with the same level of performance, the same machines and the same processors were used. So what did change?

- Various and different kernel functions were used in order to better allocate processors resources to processes when needed, in the right time, and in order to provide more accurate mechanisms in time measurement.
- The communication protocol between federates and the RTI is made more efficient by limiting the busy waiting of processes.
- Some good programming practices were introduced in model programming. They ensure that computing resources are mainly devoted to the simulation process and they permit a better progress of the federates.

All seems to be in a good kernel scheduling. Two mechanisms are particularly determining to take into account real time objectives. Scheduling policies and scheduling algorithms are the heart of the problem and of the solutions. It seems very important to make sure that preemptions can be done when needed and that scheduling algorithms are efficient without generating any heavy cost because context switches may be numerous. The experiments that were described in this paper show that enforcing a proper scheduling of tasks permit to gain in global performance and to reduce jitters in the simulation runs.

All seems to be in a good real time programming. The experiments performed in this study showed that programming practices are also determining to take real time policies into account. For example programming changes in the protocol of exchanges between federates and RTI permitted to eliminate busy waiting loops in programs. By this way processors can be allocated to other federates that need them in order to progress. In other words, programmed mechanisms inside middleware have been devised to introduce some asynchronism in it. This asynchronism can permit to release processors and to allocate them to tasks that really need them. Moreover, experiments showed that system calls are costly. The cost comes from the overhead due to the system call itself. But while running a system call, a federate can reach a preemption point. At this point the kernel scheduler is invoked and the federate may lose its processor. So system calls must be used only when there is no mean to do otherwise, and in particular I/O system calls must be prohibited. The lesson here is that the algorithmic structure of programs has a great influence on the real time behaviour of federates.

HLA Time Management is good for real time. A primarily surprise of these experiments is that time management by the RTI seems good for real time and

that the HLA Real Time scheme does not seem to be a very good thing to take real time constraints into account. The best results are obtained by requiring time management services. These services generate some overhead. But, in fact, this overhead is compensated by the better synchronization that these services enforce between federates. This better synchronization between federates reduces latencies in data exchanges, reduces the cycles durations and makes the global behaviour more regular because jitters are also very reduced. An other explicit programming of a stronger synchronization could be enforced in models in the real time scheme. In particular the use of interactions could be generalized, but this practice would generate more data transfers between federates, and probably results would be still less convincing. In fact the time advancing algorithm of the RTI enforces a very good synchronizing of federates that seems to be the best efficient approach.

Some new objectives. Some new research and development directions seem to be opened to follow these experiments. Scheduling policies are a very determining factor in permitting the simulations to satisfy their real time requirements. A possible way would be to experiment other scheduling policies (for example *Completely fair Scheduler*) or to implement some new scheduling algorithms that would be devoted to satisfy properly real time needs of federates. Some new works could be performed in the CERTI middleware in order to introduce more asynchronism: in particular the communication stack in CERTI could be made a parallel treatment and asynchronous mechanisms for notifications to the federates could be also introduced. These new developments will contribute to make hard real time distributed simulations more feasible.

7. References

[1] DMSO: “High Level Architecture Specifications” Version 1.3. 1998.

[2] IEEE: “Standard for Modeling and Simulation (M&S) High Level Architecture (HLA). Std 1516. 2001.

[3] B. Bréholée, P. Siron: “CERTI: Evolutions of the ONERA RTI Prototype” Fall Simulation Interoperability Workshop. September 2002.

[4] P. Siron, E. Noulard, J.-Y. Rousselot: “CERTI” www.cert.fr/CERTI. 2008.

[5] P. Y. Guidotti: “Entrées CNES sur les besoins de démonstration HLA” DVF-CR-BV-AI-NS-144-CN. February 2005.

[6] R. M. Fujimoto: “Time Management in the High Level Architecture” Simulation, 71, pp 388-400. December 1998.

[7] R. M. Fujimoto, R. M. Weatherly: “Time Management in the DoD High Level Architecture” Workshop on Parallel and Distributed Simulations. May 1996.

[8] T. Gleixner: “High Resolution Timers” www.tglx.de/hrtimers.html 2007.

[9] R. Love: “Lowering Latency in Linux: Introducing a Preemptive Kernel” www.linuxjournal.com/article/5600 2002.

[10] J. Aas: “Understanding the Linux 2.6.8.1 Scheduler” josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf 2006.

Author Biographies

BRUNO D’AUSBOURG received his PhD in computer science at Toulouse University in 1983. He is currently a Research Engineer at ONERA and he works in real time and distributed systems.

PIERRE SIRON was graduated from a French High School for Engineers in Computer Science (ENSEEIHT) in 1980, and received his doctorate in 1984. He is currently a Research Engineer at ONERA and he works in parallel and distributed systems. He is leader of the CERTI Project. He is also Professor at the University of Toulouse, ISAE, and the head of the computer science program of the SUPAERO formation (French High School for Engineers in Aerospace Sciences).

ERIC NOULARD graduated from a French High School for Engineers in Computer Science (ENSEEIHT) in 1995 and received his PhD in computer science from Versailles University in 2000. After 7 years working in the Aerospace & Telecom domain for BT C&SI mostly for building high performance tests & validation systems he joined ONERA research center in Toulouse as Research Scientist. He works on distributed and real-time systems and his actively involved in the development of the CERTI and TSP Open Source projects.