# Scheduling of the Distributed Thread Abstraction with Timing Constraints using RTSJ

Patricia Della Méa Plentz, Romulo Silva de Oliveira, Carlos Montez
LCMI - Automation and Systems Dept – Federal Univers. of Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianópolis – SC - Brazil
{plentz, romulo, montez}@das.ufsc.br

## Abstract

*In this paper we propose a system architecture that supports end-to-end scheduling of the distributed real-time thread abstraction. The Real-Time Specification for Java (RTSJ) is used for the implementation of distributed real-time threads. By using simulations, we conclude that the proposed architecture is flexible enough to accommodate different scheduling algorithms.*

## 1. Introduction

An end-to-end execution model is essential for obtaining predictability in distributed real-time systems. Usually, to obtain end-to-end predictability in those systems, the timing behavior (for example, timing constraints, WCET) is used in the management of resources (scheduling) in a consistent way on each node of the distributed processing. In a distributed real-time system, the scheduling of tasks considering timing constraints is necessary for the correct operation of this system. The academic community, along the years, has researched several aspects of the end-to-end scheduling.

In this context, the Distributed Thread (DT) concept can be used as a central abstraction for those systems. Basically, a DT is a schedulable entity that can transpose nodes, carrying its scheduling context (timing constraints) among the entities responsible for the scheduling in those nodes [3].

A possible employment of this abstraction can be found in control and supervision applications. In those applications, control tasks execute locally and they have hard or firm deadlines. On the other hand, supervision tasks have soft deadlines and they are distributed - visiting several nodes to collect information for the carrying out of analyses and diagnoses. Those tasks – that can be modeled as DTs – are created at pre-programmed instants, or when some supervision level alarm is activated (for example, when a variable representative of the state of the physical plant leaves the interval of normal values) or still by solicitation of a human operator. Other examples of the use of this abstraction are described in [8] and [9].

Distributed threads also can be used to solve some problems related to RMI [19], which are: a) Java RMI does not propagate synchronization operations to remote objects; b) Java RMI does not maintain thread identity across different machines [15]. There are solutions in the literature based on DT, because that abstraction is identified in all the systems´s nodes as a unique computational entity.

In [20] it is described a Real-Time RMI (RT-RMI) framework that supports timely invocation of remote objects. Although this is an ongoing work, it doesn´t address the problems related to standard RMI listed above. Moreover, we understand that DT abstraction is of higher level than RMI. Actually, RMI is used as communication support to implement DTs.

The DT can be implemented as part of the operating system (e.g. Alpha system [3]), as part of middleware (e.g. RT-CORBA 1.2 [12]) or as part of a programming language (e.g. proposal for Java [5]).

The implementation of distributed real-time threads in Java is facilitated if a real-time extension is adopted for this language, like the RTSJ (Real-Time Specification for Java) [4]. This specification extends the Java platform with new concepts and mechanisms, creating an environment that allows the creation of real-time applications. However, it was not defined for distributed systems and the extension of RTSJ for that kind of environment raises many questions. The effort for a definition of RTSJ for distributed environments is at an initial phase. Research on several aspects of that context is still required [6].

In the literature there are only a few works related to distributed real-time threads. Besides, most of them investigate the use of that abstraction in distributed systems, but few of them study it in the context of distributed real-time systems.

The objective of this paper is to define a system architecture that supports the distributed real-time thread abstraction. In this work, this kind of thread implements soft real-time aperiodic tasks. With this architecture we intend to meet deadlines of hard periodic local tasks, while minimizing response times of soft aperiodic distributed tasks. We intend to study the end-to-end scheduling of distributed real-time threads having the RTSJ as support for local execution. The challenge is to find scheduling policies that can be applied in this context, considering that, at the end, these policies

should be mapped using the scheduling mechanisms present in RTSJ.

The remaining of the paper is organized as follows: Sections 2 and 3 describe briefly the RTSJ and the main concepts related to DTs. In section 4 the assumptions of this work are presented, and section 5 contains the proposed scheduling approach. The architecture implementation using RTSJ is presented in section 6. The results obtained through simulation are presented and discussed in section 7. Section 8 shows related work. Conclusions are presented in section 9.

## 2. RTSJ

The standard Java platform seems an unusual choice for the implementation of real-time systems. With the Real-Time Specification for Java – RTSJ it is possible to carry out the programming of real-time applications [4].

RTSJ extends the Java platform through the creation of new classes and interfaces, and also through mechanisms inserted in the Java Virtual Machine – JVM. The main items that it adds to the existing Java specification are the following:

- Real-Time Threads: These threads have scheduling attributes that allow the definition of real-time tasks (period, deadline, deadline miss handler and overrun handler).
- New Memory Allocation Domains: These domains allow programmers to write real-time code in memory areas where it will never be delayed by garbage collection.
- Events and Asynchronous Event Handler: This facility allows the association of asynchronous events with happenings outside the JVM.
- Asynchronous Transfer of Control: This mechanism lets a real-time thread to affect the control flow of another real-time thread. It is a carefully controlled way for one real-time thread to throw an exception into another thread.

This specification introduces the concept of a schedulable object, which can be a real-time thread, a no-heap real-time thread or an asynchronous event handler.

Although RTSJ has very interesting features for real-time programmers, it was not elaborated for distributed systems and the enhancement of RTSJ for this kind of system presents many questions, which are currently being discussed in the literature [16].

## 3. Distributed threads

The Distributed Thread (DT) abstraction is a schedulable entity that can transpose physical nodes. It acts as a local thread at each node of the distributed system on which it executes. Usually, a DT is implemented through a sequencing of executions of local threads (Figure 1) [3, 5, 7, 9]. Thus, it is possible to visualize a DT being composed by local segments of execution. In each node that the DT executes, a local segment of this DT is created, that is implemented by a local (operating system) thread. Although it executes on several nodes, each instance of a DT is an unique entity, having a well-known identifier valid in the whole system.

All nodes that host part of the execution of a DT are denominated segment nodes. At any point in time, a DT should be eligible for execution (or suspended) at only one node in the distributed system. That segment node receives the special name of head node. The node in which the DT is created is denominated source node.

When a DT departs from a node it means that it made a remote invocation. On the other hand, when a DT arrives at a node it means that it will execute a remote method in this node.

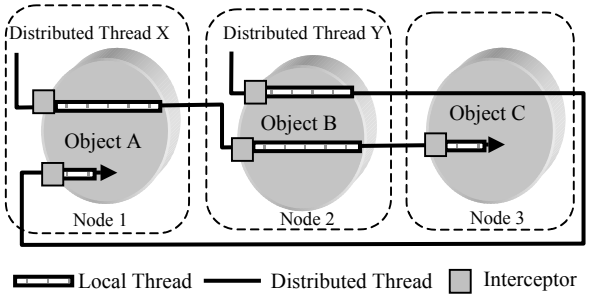


**Figure 1. Distributed thread.**

Whenever a DT transposes a node, it carries parameters and other attributes of the computation that it represents. Those attributes can be modified and accumulated, in a nested way, as it executes operations inside distributed objects [3]. When a DT begins its execution in a node of the system, this is scheduled according to the local scheduling policies. Consequently, an end-to-end scheduling model should maintain coherence among the scheduling policies in each segment node of the DT.

Distributed real-time threads can share physical resources (e.g. processor, disk, I/O) and logical resources (e.g. locks), which can be subject of mutual exclusion constraints [9]. A program consists of multiple DTs executing concurrently and asynchronously.

A distributed real-time thread may terminate its execution in the same node where it began, or in another node of the system. It can also create a new distributed real-time thread (e.g. through one-way invocations). In this case, the new distributed real-time thread may begin its execution in another node of the system. In this work, we are not considering this kind of invocation.

## 4. Problem formulation

In this work we use distributed real-time threads in the implementation of an application for a distributed

real-time system, where the consequences of a timing fault of the DT are of the same order of magnitude as the benefits of the system when in normal operation. We are considering a factory automation system, for instance, where only one application executes in all the nodes of the system, at a given moment.

The application has, in each node, periodic local tasks with hard deadlines and aperiodic distributed tasks with soft deadlines. The scheduling of the aperiodic distributed tasks must not jeopardy the scheduling of the periodic local tasks, because these last ones are considered critical for the application.

The periodic tasks are implemented as RTSJ periodic real-time threads and they execute with fixed priorities. The aperiodic distributed tasks have end-to-end timing constraints and they are recurrent, that is, they may execute several times. They can be created at runtime. Consequently, this kind of system has a dynamic load, and it may occur overload situations. Distributed real-time threads represent these distributed tasks. The local segments of these DTs are implemented, in each node, by using local RTSJ real-time threads.

An end-to-end deadline is defined at the moment of creation of each distributed real-time thread. The distributed real-time threads carry this time constraint. Each node of the system has a scheduler. These schedulers are independent and they do not collaborate with each other in an explicit way. The scheduling is just influenced by the timing attributes associated to each DT.

The future execution flow of the distributed real-time thread is known at the moment of its creation, as well as the estimated average execution time (ACET).

## 5. Proposed system architecture

In this work, the scheduling approach for DTs will be solved in two stages: partitioning of the end-to-end deadline and local scheduling. This is not an uncommon procedure in the real-time literature [14].

Figure 2 illustrates the architecture present in each node of the system. The distributed real-time system model of this work uses interceptors and aperiodic servers in each node of the system. The interceptors are responsible for servicing the aperiodic distributed real-time threads, and the aperiodic servers are responsible for executing the local segments of DTs. The interceptors are local threads with execution times previously reserved, that is, they are considered as one additional periodic task of the system. The aperiodic server is implemented through RTSJ mechanism called *Processing Group Parameters* [17]. With this mechanism it is possible to associate an object, which represents a *Processing Group Parameters,* to a schedulable object group (in this case aperiodic real-time threads that represent segments of DTs), creating then a logical server. The *Processing Group Parameters* attributes are: start time, period, cost, deadline, overrun handler and deadline miss handler.

By the time aperiodic real-time threads associated with this group should execute, the RTSJ scheduler verifies that these threads belong to a logic server. Then, the execution of aperiodic real-time threads is carried out according to server's capacity.

The real-time specification for Java doesn't provide guidelines on how to use *Processing Group Parameters*. Moreover, this specification doesn't bring any aperiodic server algorithm implemented. In this paper we use the mechanism proposed in [17] for the implementation of aperiodic servers through the *Processing Group Parameters*.

The standard RTSJ has a preemptable fixed priority scheduler. For this reason, in this system we will use the Rate Monotonic (RM) algorithm [10], which assigns priorities to the tasks with values inversely proportional to their periods. This algorithm will schedule the periodic local tasks with hard deadlines, as well as the interceptor and the aperiodic server.

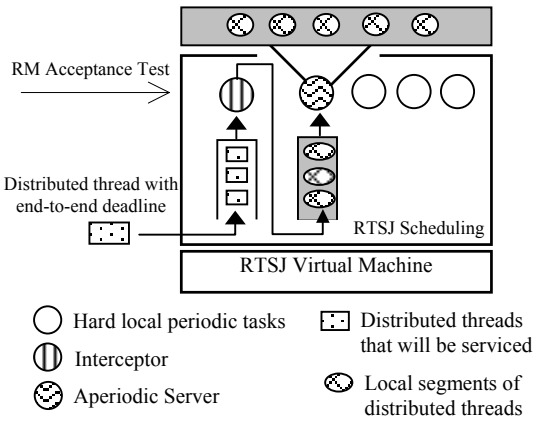


**Figure 2. System architecture.**

The interceptor services an aperiodic distributed real-time thread when it arrives at a node (head node). Each interceptor contains a list of the local segments of the distributed real-time threads that execute (or executed) in that node. For each aperiodic distributed real-time thread that arrives, the interceptor verifies if a local segment of this DT already exists. In affirmative case, this local segment is activated to execute on behalf of the aperiodic distributed real-time thread that arrived. Otherwise, the interceptor creates a local segment with the function of executing on behalf of its parent DT.

In both cases, the local segment of the DT inherits all the properties of the aperiodic distributed real-time thread, such as identifier and timing constraints.

In the sequence, the interceptor includes this local segment in the aperiodic server's queue for that node. The aperiodic server's queue is scheduled through the Earliest Deadline First (EDF) algorithm [10].

Each RTSJ local real-time thread that implements a segment of a distributed real-time thread has a data structure associated that contains information such as its identifier and scheduling parameters.

The arrival at a node and the exit for another node of a distributed real-time thread are considered scheduling events for the aperiodic server. When a distributed real-time thread arrives at a node, and a new local segment is created to execute on behalf of it, the server should be informed that exists a new local segment that will compete for resources with the other local segments already executing in that node.

The scheduling objective of the proposed architecture is to guarantee the deadlines of the hard local tasks. At the same time, it should reduce the response time of the aperiodic tasks in order to service the deadlines of the local segments of DTs and, consequently, to meet the end-to-end deadlines of the DTs.

### 5.1 Local deadline assignment

Usually, in distributed real-time systems, the end-to-end deadline of a distributed task is defined as part of the application specification. Therefore, it is necessary to define a way to partition the end-to-end deadline of the distributed task among its local segments.

The segments compose a DT, what facilitates the choice of points for partitioning the end-to-end deadline. In that way, we investigate solutions for assignment of deadlines to local segments, by taking into account the end-to-end deadlines of the DTs, and assuming the RTSJ abstractions and mechanisms as underlying system.

The literature presents different forms of deadline partitioning ([8], [14], [1]). The method called *Ultimate Deadline* is very simple because the deadline of a local segment is equal to the end-to-end deadline of its DT. In [8], the authors suggest a deadline partitioning method called *Equal Flexibility* (EQF), that proposes that the total remaining slack of the distributed task is divided among its subtasks in the proportion of its estimated execution times (the term "subtask" is used, in that work, to designate local segments of a distributed task). In that paper it is also defined the concept of flexibility of a task, which is the quotient of the slack of this task by its execution time. Thus, the more flexible a task is, less strict are its timing constraints. With the EQF partition method, subtasks of the same distributed task have the same flexibility, although they have different slack values.

The method suggested in [8] can be adapted for the context of this work. Thus, the total remaining slack of the distributed real-time thread is divided among its segments in the proportion of the estimated execution times of each segment on each node. This can be made at the moment of the activation of the DT or during its execution, as it transposes the nodes of the system.

The difference between the two strategies is that in the first the calculation for all the segments is carried out before the distributed real-time thread begins its execution. Thus, the DT, from the start, carries the calculated deadlines. In the second strategy, the interceptor carries out the calculation, as the distributed real-time threads transpose the nodes of the system.

The EQF method makes all segments of the distributed real-time thread to receive a proportional slack, according to the estimated execution time. The segment of the distributed real-time thread with smaller deadline will have higher priority at the node that it executes, with regard to the queue of the aperiodic server.

### 5.2 Local scheduling

In the literature there are several well-known algorithms for the scheduling of aperiodic tasks [11]. The Bandwidth-Preserving, the Polled Executions and the Slack Stealing Servers are proposed for priority-driven systems, among others.

Polling Server [13] is another algorithm used for scheduling aperiodic tasks. It executes as if it was a periodic task and it uses its capacity to service the aperiodic tasks. A queue of aperiodic tasks exists and, during its execution, the server verifies this queue and executes the tasks in it. The Polling Server suspends its execution or the scheduler suspends it after it consumed all its execution time, or when the aperiodic queue becomes empty.

The Background Server algorithm [13] is another solution for scheduling aperiodic tasks. It only executes aperiodic tasks when periodic tasks are not ready for execution. This algorithm produces correct scales and it is of easy implementation.

There are several algorithms that improve the Polling Server method, conserving its execution time when it finds the aperiodic queue empty, and allowing it to execute later on within its period if an aperiodic task arrives. These algorithms are called Bandwidth-Preserving Servers. SpSS [13] is a kind of Bandwidth-Preserving server. It has an execution budget, with consumption and replenishment rules that guide its execution and, therefore, the execution of the aperiodic tasks.

## 6. Implementation of the proposed architecture using RTSJ

In this work, the proposed architecture is implemented by using periodic and aperiodic RTSJ real-time threads. Both of them with deadline miss handlers. Moreover it is used the *Processing Group Parameters* mechanism to implement the aperiodic server and RTSJ methods that guarantee the periodic execution of tasks.

Listing 1 presents the creation of four periodic real-time threads that represent the hard local tasks of the system and the interceptor, as shown by Figure 2.

```
import javax.realtime.*;
public class SystemArchitecture {
   public static void main(String[] args) {
      PeriodicTask  tp1 = new PeriodicTask ();
      PeriodicTask  tp2 = new PeriodicTask ();
      PeriodicTask  tp3 = new PeriodicTask ();
      Interceptor    in =  new interceptor();

      tp1.start(); tp2.start(); tp3.start(); in.start();
      try {
         tmv.join(); tmo.join(); tmn.join();
      } catch (Exception e)
...
```

**Listing 1. Periodic real-time threads creation.**

The deadline miss handler is shown in Listing 2. Each application thread uses its own instance of this class. The *schedulePeriodic()* method guarantees that the thread with a missed deadline will be rescheduled.

```
public class MissHandler extends AsyncEventHandler {
   String thread;
   RealtimeThread me;

   public MissHandler(RealtimeThread th, String t) {
     me = th; thread = t; }

   public void handleAsyncEvent() {
     RealtimeThread rtt = RealtimeThread.currentRealtimeThread();
     Tela.println("Thread:" + me.getClass().toString() + "miss dead.");
     me.schedulePeriodic();
...
```

**Listing 2. Deadline miss handler.**

Listing 3 shows the real-time thread class constructor. It uses *ReleaseParameters* to define periodic real-time threads.

```
public class PeriodicTask extends RealtimeThread {
   public PeriodicTask () {  super();
     // Declare ReleaseParameters w/ params (periodic,period, ...)
     ReleaseParameters release = new PeriodicParameters(
       new RelativeTime(0000, 0),   // start
       new RelativeTime(0000, 0),   // period ... seconds
       new RelativeTime(0000, 0),   // cost
       new RelativeTime(0000, 0),   // deadline
       null,                        // overrun handler
       new MissHandler(this, "PeriodicTask")); //deadl miss handler
     // Declare SchedulingParameter with priority
     SchedulingParameters sched  = new
            PriorityParameters(PriorityScheduler.MIN_PRIORITY + 11);
     // Assign the parameters of the thread
     setSchedulingParameters(sched);
setReleaseParameters(release);
   }
   public void run() {
     while(true) {
          do { // execute some periodic computation
          } while(waitForNextPeriod());
...
```

**Listing 3. Periodic real-time threads constructor.**

Listing 4 shows interceptor class and interceptor queue implementations. This class represents the Interceptor present in each system node (Figure 2). The *InterceptorQueue* class is composed by *consultQueueState(), removeOfQueue()* and *insertInQueue()* methods.

```
public class Interceptor extends PeriodicTask {
  public void run() {
    while(true) {
      do {
        if (InterceptorQueue.consultQueueState != empty)
          if (LocalSegmentsList.comsultList(id_TD) == true)
            LocalSegment.activeLocalSegment;
          else{ LocalSegment.criateLocalSegment;
            LocalSegmentList.insertInList(id_TD)
       }
       LocalSegmento. inherit Properties(id_TD);
       AperiodicServerQueue.insertInQueue(id_TD);
       InterceptorQueue.removeOfQueue(id_TD);
      } while(waitForNextPeriod());
...
public class InterceptorQueue{
    public synchronized static consultQueueState() { ... }
    public synchronized static removeOfQueue() { ... }
    public synchronized static insertInQueue() { ... }
...
```

**Listing 4. Interceptor queue.**

Listing 5 shows the class that implements a DT when it starts its execution in the system (Figure 2). The end-to-end deadline of the DT, priority and release parameters are defined in this class. Also, it is defined the association of this DT with the logical server – *ProcessingGroupParameters*.

```
public class DistributedThread extends RealtimeThread {
  public DistributedThread(
    ReleaseParameters release = new AperiodicParameters ( ...)
    SchedulingParameters sched = new PriorityParameters (...)
    ProcessingGroupParametes group = new
                        DeferrableProcessingGroupParameters;)
  public void run() {
    while(true)      //execute some aperiodic computation
...
```

**Listing 5. Distributed thread creation.**

Listing 6 describes the DT local segment implementation, which is shown in Figure 2. The *activeLocalSegment()* method is used when the DT arrives at a node that already has a segment that represents it. On the other hand, when the node doesn't have a segment representing the DT, the *criateLocalSegment()* method is used. In both cases, the *inheritProperties()* method is executed so that the local segment receives the distributed thread parameters when it arrives at the node. The class that represents the local segment list has the *consultList() and insertInList()* methods.

```
Public class LocalSegment {
  public synchronized static activeLocalSegment () {...}
  public synchronized static criateLocalSegment () {...}
  public synchronized static inheritProperties() { ...
              // local segment receive the properties of the TD }
}
public class LocalSegmentsList {
  public synchronized static consultList() {...}
  public synchronized static insertInList() {...}
...
```

**Listing 6. Local Segment and local segments list.**

The aperiodic server and server queue is presented in Listing 7. We use the implementation proposed in [17],

which uses the RTSJ Processing Group Parameters mechanism.

```
public class DeferrableProcessingGroup extends
                                    ProcessingGroupParameters {
  public DeferrableProcessingGroupParameters(
    HighResolutionTime start,
    RelativeTime period,
    RelativeTime cost,
    PriorityParamters pri)

  super(start, period, cost, pri, null, null);

  // save priority, throw exceptions, update queue, etc.
...
public class AperiodicServerQueue {
  public synchronized static insertInQueue() {...}
...
```

**Listing 7. Aperiodic Server and Server Queue.**

The relationship among the main classes and interfaces described in this paper are shown by the UML diagram of Figure 3.

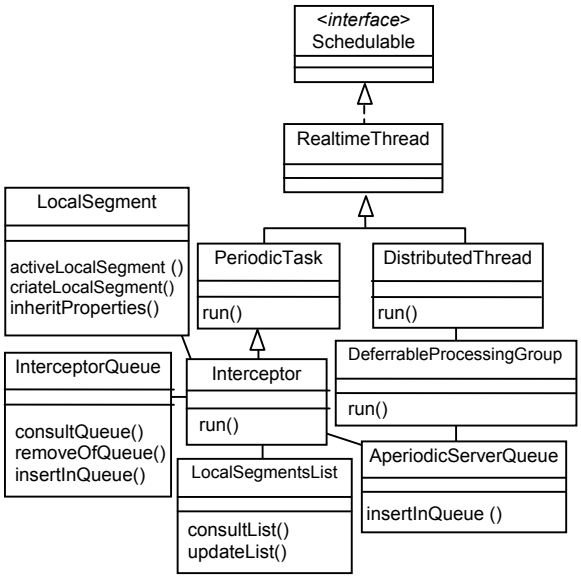


**Figure 3. Relationship among classes and interfaces.**

## 7. Simulation

The objective of the simulation is to observe the sensibility of the proposed architecture with regard to the server type used for scheduling the local segments of DTs. We also consider the policies that it uses in its queue. For this objective we considered in our experiments the rate of met *deadlines* as main metric.

For partitioning the end-to-end deadlines of the aperiodic distributed threads we will use the Ultimate and EQF methods [8]. The interceptor of each node will carry out this partitioning. The deadline partitioning using method EQF will be carried out during the execution of the DT, as it transposes the nodes of the system (dynamic partitioning). The deadline partitioning using method Ultimate will be carrying out before the DT starts its execution in the system (static partitioning).

In this work we are proposing the use of two types of aperiodic servers, Background Server and Polling Server [13], for scheduling the local segments of the aperiodic real-time distributed threads. They were chosen because both are classic algorithms of the literature and they are of easy implementation. The queue of the aperiodic server will be scheduled by the Earliest Deadline First (EDF) [10] and First-In First-Out (FIFO) algorithms. The combination of the two types of servers with the two queue ordering disciplines defines four different execution scenarios, which will be simulated and analyzed in the sequence of this text. The conditions of the simulation are described bellow.

### 7.1 Simulation conditions

The system is composed by 3 interconnected nodes. For each node of the system, the use of the processor was supposed to be 50% for periodic local tasks with hard deadlines; and 50% for the aperiodic server.

There are 4 hard periodic local tasks in each node, whose periods are distributed uniformly between 10 and 100. Their utilizations are of 20% for the task with the smallest period and 10% for each one of the others three. Those tasks have relative deadlines equal to their periods.

There are 10 aperiodic distributed real-time threads, each of them composed by 6 local segments of execution. The aperiodic distributed real-time thread always executes a method in the node where it was created. These DTs are divided in two different groups.

In the first group, the inter-arrival time of these DTs follows an exponential distribution with different average for each simulation (100, 150, 200, 250 and 300 units of time - u.t.), the end-to-end deadline also varies in each simulation (100, 150, 200, 250 and 300 u.t.) and the WCET of each segment is in the interval between 1 and 5 u.t. In the second group, the inter-arrival time of the aperiodic distributed real-time follows an exponential distribution with average of 500, 750, 1000, 1250 and 1500 u.t., in each simulation. The end-to-end deadlines are of 500, 750, 1000, 1250 and 1500 u.t., in each simulation. The WCET of each segment is in the interval between 10 and 50 u.t.

It is assumed that the clocks of the system are synchronized, there is no network partition and that the delay in the communication network has exponential distribution with known average of 6 u.t.

Two aperiodic servers, Background Server and Polling Server, are used. This last one with capacity of 5 u.t. and period equal to 10 u.t. For each server, we used the EDF and FIFO algorithms for scheduling the aperiodic queue. For each pair Aperiodic Server – Aperiodic Queue Algorithm, it was carried out 5 simulations of 1.000.000 u.t. each one. For each simulation, the aperiodic distributed real-time thread

load was changed. Also, each simulation was done twice, each time with a different deadline partitioning method.

As previously described, in this system we will use the RM algorithm, which will schedule the periodic local tasks with hard deadlines, as well as the interceptor and aperiodic server.

### 7.2 Simulation results

Figure 4 shows the results with regard to the rate of end-to-end deadlines serviced. As the aperiodic load increases (X axis), the aperiodic server is able to service a smaller number of end-to-end deadlines.

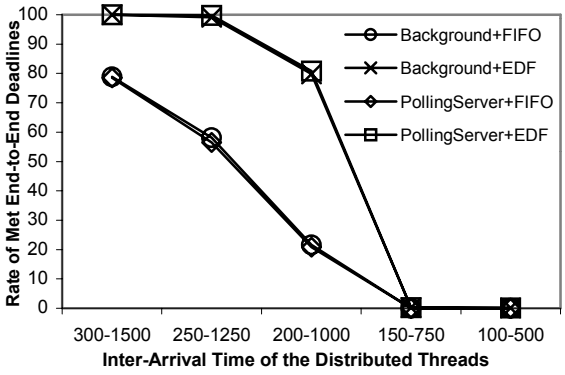


**Figure 4. Rate of met end-to-end deadlines of the distributed real-time threads using Ultimate deadline partitioning method.**

Figure 5 shows the results using EQF deadline partitioning method. When considering only the use of Polling Server for scheduling DTs local segments, it is noticed that the use of EDF in the server queue produces better results than when FIFO is used. EDF is also better than FIFO as the policy of the local segments queue when a Background Server is used. These results were observed both when EQF and Ultimate methods were used.

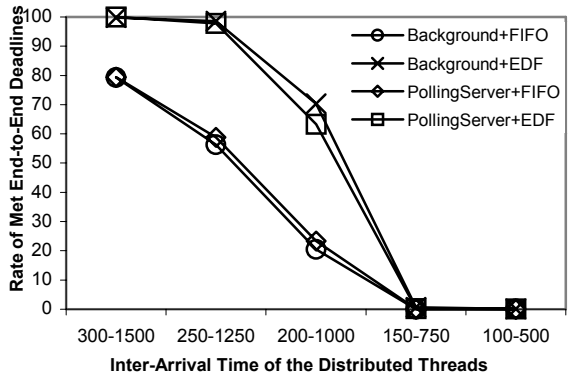


**Figure 5. Rate of met end-to-end deadlines of the aperiodic distributed threads using EQF method.**

The simulations showed that for both deadline partitioning methods, and considering only FIFO as server queue scheduling policy, Background Server and Polling Server presented very similar results. The same happens when considering only EDF as server queue scheduling policy. That happens because of the simulation conditions and because a pure Polling Server was used instead of a Polling + Backgroung Server.

In relation to the deadline partitioning methods, we observed that there was no significant difference between Ultimate and EQF methods, for the conditions simulated.

The experiences showed a great sensibility of the proposed architecture with regard to the scheduling solution employed. The scheduling solution should be carefully designed for the proposed scenario, that is, DTs with soft deadlines executing along with local periodic tasks with hard deadlines.

## 8. Related work

The literature presents several works that investigate scheduling solutions for DTs, where end-to-end timing constraints are considered. In [1] it is presented a method for scheduling a system of flow-shop tasks. In [14], the author suggests an integrated end-to-end scheduling framework, where the end-to-end deadline is the main requirement of the system. In [2], a scheduling method is proposed to service soft aperiodic requests in a hard real-time environment, where a set of periodic tasks with hard deadlines is scheduled using the EDF algorithm.

In spite of the fact that all those works investigate subjects related with end-to-end scheduling, none of them address DTs and execution support based on RTSJ.

Some works in the literature address the use of DTs out of the context of real-time systems. In [15], they propose the implementation of DTs through bytecode transformation of stub routines only, instead of the entire client application. The authors consider that this approach reduces the system overhead when compared with others techniques as in [5] and [18].

Some authors describe the implementation of real-time distributed threads. In [9] it is proposed an end-to-end scheduling framework at the application level. This framework is based on the CORBA 2.0 middleware, that supports the DTs notion as programming abstraction for distributed real-time systems.

Currently, the literature presents a work that proposes a mechanism so RTSJ provides support for DTs [16]. The authors exploit the ways in which the RTSJ can be integrated with RMI, to allow that distributed real-time threads execute remote objects invocations. They propose three integration levels, where in one of them the Java virtual machine is extended to support distributed real-time threads.

None of the above works propose the RTSJ as support for the implementation of DTs without modifying the Java Virtual Machine. Through concepts and mechanisms, this work presents a feasible

implementation proposal of DTs using the RTSJ without to alter the RTSJ virtual machine. The mechanisms described in this paper are portable to any system that has a RTSJ virtual machine installed.

## 9. Conclusion

In this paper we presented a system architecture that offers support to the abstraction distributed real-time threads. The proposed architecture services the deadlines of the hard local periodic tasks, while it tries to minimize the response time of the soft aperiodic distributed threads. The use of interceptors was shown important for the creation of the local segments of aperiodic distributed real-time threads, and the dispatch of these to the aperiodic server queue.

We described a hybrid task set for the proposed system in this work. It was composed by some periodic local tasks with hard deadlines and by aperiodic distributed threads with soft deadline. We concluded that the proposed architecture is flexible enough to accommodate this hybrid task set as well as several scheduling policies.

In [16] the authors had proposed the implementation of the DTs in the RTSJ context, suggesting new classes and interfaces, and alterations in the Java virtual machine. Our work, on the other hand, follows the approach of implementing the DT abstraction without having to modify the Java virtual machine. In this direction, we adopt the use of interceptors and aperiodic servers, through the RTSJ mechanisms, such as the *Processing Group Parameters* and *Release Parameters* for real-time threads.

In the simulation experiments we applied two end-to-end deadline-partitioning methods described in the literature (Ultimate Deadline and EQF). The first one was used in the static context, and the last one, in the dynamic context. However, it was observed that there were not significant differences in the obtained results when the adopted method was changed.

We intend to continue exploring the flexibility of this system architecture, through, for example, the use of other aperiodic server and local scheduling algorithms.

**Acknowledgement**

This work is partially supported by a research grant from CNPq - The Brazilian National Council for Scientific and Technological Development.

**References**

[1] R. Bettati, J.W.S. Liu. "End-to-end Scheduling to Meet Deadlines in Distributed Systems". In ICDCS, Yokohama, Japan, 1992, pp.452-459.

[2] G.C. Buttazzo, F. Sensini. "Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments", IEEE Trans. On Computers, vol. 48, no. 10, pp. 1035-1052, Oct. 1999.

[3] R.K. Clark, E.D. Jensen, F.D. Reynolds. "An architectural overview of the Alpha Real-Time Distributed Kernel", in Proc. USENIX Workshop on Microkernels and Other Kernel Architectures, Seatlle, 1992, p.p. 127–146.

[4] P.C. Dibble. "Real-Time Java Platform Programming",1st ed., Prentice-Hall, 2002.

[5] B. Haumacher, T. Moschny, J. Reuter, et al. "Transparent Distributed Threads for Java", in IPDPS, Nice, França, 2003, pp. 136.1.

[6] E.D. Jensen. "The Distributed Real-Time Specification for Java – An Initial Proposal", J. Computer Systems Science and Engineering. Vol. 16, Issue 2, March 2001.

[7] E.D. Jensen (March 2004), "Distributed Threads: An End-to-End Abstraction for Distributed Real-Time", http://www.real-time.org/docs/distributed_threads.pdf

[8] B. Kao, H.G. Molina. "Deadline Assignment in a Distributed Soft Real-Time System", IEEE Trans. Parallel and Distributed Systems, vol. 8, no. 12, pp.1268-1274, Dec 1997.

[9] P. Li, et al. "Scheduling Distributable Real-Time Threads in Tempus Middleware", in Proc. ICPADS, Newport Beach, California, pp. 187, Jul. 2004.

[10] C.L. Liu, J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", J. ACM, vol. 20, no. 1, pp. 40-61, 1973.

[11] J. W.S Liu. "Real-Time Systems", 1st ed., Ed. New Jersey : Prentice Hall, 2000.

[12] Object Management Group – OMG. Dynamic Scheduling Real-time CORBA 2.0 - OMG Final Adopted Specification. OMG document PTC/2001-08-34, 2001.

[13] B. Sprunt, L. Sha, J.P. Lehoczky. "Aperiodic Task Scheduling for Hard Real-Time Systems". J. Real-Time Systems, vol. 1, no. 1, pp. 27-60, 1989.

[14] J. Sun. "Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems", Thesis – Graduate College, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1997.

[15] E. Tilevich, Y. Smaragdakis. "Portable and Efficient Distributed Threads for Java". In: Middleware (Oct. 2004 : Toronto, Canada).

[16] A. Wellings, et al. "A Framework for Integrating the Real-Time Specification for Java and Java´s Remote Method Invocation". In: IEEE ISORC, Apr.2002. p.13-24.

[17] A. Wellings, A. Burns. "Processing Group Parameters in the Real-time Specification for Java". In: JTRES (Nov. 2003 : Catania, Italy).

[18] D. Weyns, E. Truyen, P. Verbaeten. "Distributed Threads in Java". In: ISPDC (Jul. 2002 : Iasi, Romania). Proceedings. p. 94-104.

[19] Sun Microsystems, Remote Method Invocation Specification, http://java.sun.com/products/jdk/rmi/, 1997.

[20] A. Borg, A. Wellings. "A Real-Time RMI Framework for the RTSJ". In: ECRTS 03 (15. : Jul. 2003 : Porto, Portugal). IEEE Computer Society, 2003. p. 238-248.