ETFA'2007 - 12th IEEE Int. Conf. on Emerging Technologies and Factory Automation

Prediction of End-to-End Deadline Missing in Distributed Threads Systems

Patricia Della Mea Plentz, Carlos Montez, Rômulo Silva de Oliveira
Programa de Pós-Graduação em Eng. Elétrica – Federal University of Santa Catarina (UFSC)
88.040-900 – Florianópolis – SC – Brazil -- {plentz, montez, romulo}@das.ufsc.br

Abstract

Distributed real-time threads are schedulable entities with an end-to-end deadline that traverse nodes, carrying their scheduling context. In each node, the thread will be locally scheduled and predictions about deadline missing allow that actions are carried out to improve system performance. This paper presents a task model and deadline partitioning algorithms that consider the possibility of a distributed thread to follow different paths. The future execution flow of the distributed thread is only probabilistic known before its execution. End-to-end deadline missing prediction mechanisms can be carried out through definition of estimated local deadlines. Simulations show that the proposed prediction mechanism presents good results in overloaded systems.

1. Introduction

Along the years, the academic community has researched several aspects of the end-to-end scheduling [10, 9]. The correct operation of distributed real-time systems that have tasks with timing constraints depends on adequate scheduling policies [7,1].

A Distributed Thread (DT) is a schedulable entity that can traverse nodes, carrying its scheduling context (timing constraints) among the entities responsible for the scheduling in those nodes [1]. This concept can be used as a central abstraction for distributed real-time systems in control and supervision applications, for example. Control tasks are executed 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 are created at pre-programmed instants, or when some supervision level alarm is activated and it can be modeled as a DT. In [3] and [4] other examples of this abstraction are described.

Usually, an end-to-end deadline of a distributed task is defined as part of the system requirements. It is necessary to define a way for partitioning an end-to-end timing constraint among local tasks.

In this work we assume a system architecture that supports the DT abstraction with timing constraints. This kind of thread represents firm real-time aperiodic tasks. With this architecture we intend to meet deadlines of hard periodic local tasks, while meeting firm end-to-end deadlines of aperiodic tasks.

The objective of this paper is to propose different strategies to estimate the probability of end-to-end deadlines to be met. It is assumed that DTs are aperiodic and they have firm end-to-end deadlines. The end-to-end deadline missing prediction can be done generating estimated local deadlines, which are defined from end-to-end deadline partitioning methods. Good deadline missing predictions allow that remedial actions can be taken in time to improve the system performance.

The remaining of the paper is organized as follows: Sections 2 and 3 describe briefly some related works and the main concepts related to DTs. In section 4 are presented end-to-end deadline partitioning methods found in the literature. The proposed model is described in section 5 and the strategies proposed in this paper for deadline partitioning considering a probabilistic knowledge of the future itinerary that the DT will carry out are presented in section 6. The mechanisms for deadline missing prediction are presented in section 7. The results obtained through simulation shows in section 8 the applicability and performance of end-to-end deadline missing prediction mechanisms proposed. Section 9 contains the final remarks.

2. Related Work

Real-time DT implementations are described by some authors [4, 10, 9, 11, 8]. In [4] an end-to-end scheduling framework at the application level is proposed. This framework is based on the CORBA 2.0 middleware and supports the DT notion as programming abstraction for distributed real-time systems. The goal of the Distributed Scheduling Service (DSS) framework [10] is to achieve globally sound end-to-end scheduling and overload management using the local enforcement capabilities of the local end systems. This framework also considers CORBA 2.0 distributed threads and uses the Effective Deadline [3] partition method. Furthermore, all subtasks that compose a distributed thread are known, including their corresponding execution times and nodes where it will be executed. In [9] the end-to-end deadline is the main requirement of the system and the author suggests an integrated end-to-end scheduling framework. In [11] a performance comparison between DTs and event channels is presented. The work also proposes the integration of *Release Guard* synchronization protocol [9] with DTs to improve its schedulability. A system architecture that supports end-to-end scheduling of the real-time DT is proposed in [8]. The Real-Time

Specification for Java (RTSJ) is used for the implementation of distributed real-time threads so that the proposed architecture is flexible enough to accommodate different scheduling algorithms.

Many works have addressed the end-to-end deadline partition problem [3, 7, 2, 6]. In [3] the authors study the subtask deadline assignment problem and suggest guidelines for deriving subtask deadlines from a global task's end-to-end deadline. Among the proposed deadline partition methods in that work are: Ultimate, Effective Deadline, Equal Slack and Equal Flexibility. It compares the deadline miss rates among the proposed methods and it is observed that the EQF method significantly improves the performance of global tasks, while local tasks have a good chance to meet their deadlines. The *Slicing* technique is proposed in [7], where time slices are assigned to tasks using the critical-path concept. According to the authors of that work, the critical-path is that one that minimizes the overall laxity of the task graph. They use two laxity metrics: *Fair Assignment* and *Assignment Proportional to the Workload*. The first one assigns a task deadline based on the number of tasks in the critical path, and the second one assigns deadlines based on their execution times. In [2] an improvement of *Slicing* technique is proposed, called *Adaptive Slicing Technique* (AST), which encompasses new metrics that are capable of adapting themselves to changes in workload and system size. Also, the degree of task graph parallelism that can be exploited in the system is taken into account. In [6], the proposed algorithms check the acceptability of a new flow. In the case of the flow being acceptable, they must distribute its end-to-end deadline on the visited nodes between the source and the destination. The authors suggest two new methods of laxity distribution, *Fair Laxity Distribution* (FLD) and *Unfair Laxity Distribution* (ULD). The performance of FLD and ULD methods are compared with two methods proposed in [7]. ULD shows better results when *Earliest Deadline First* (EDF) [5] non-preemptive scheduling is used.

Some few works propose mechanisms for response time or deadline missing predictions [12, 13]. In [12], deadline prediction algorithms for embedded systems are proposed. They are based on response time of tasks systems. In [13], exceptions triggering predictions in workflow systems are studied. One of the algorithms proposed in that work considers the costs associated with exception triggers and uses available slack for deadline adjustment. None of these works address the problem of deadline missing prediction from local deadlines of DTs with probabilistic knowledge of its future itinerary.

3. Distributed Threads

A Distributed Thread (DT) is an abstract entity that can span physical nodes carrying its scheduling parameters. Usually, a DT is implemented through a sequencing of executions of local threads (Figure 1) [1] and each one acts as a local thread at a particular node of the distributed system on which it is executed.

Thus, it is possible to visualize a DT being composed by local segments of execution. In each node that a DT executes, a local segment of this DT is created, being implemented by a local (operating system) thread. Although it executes on several nodes, each instance of a DT is a unique entity, with a well-known identifier valid in the whole system.

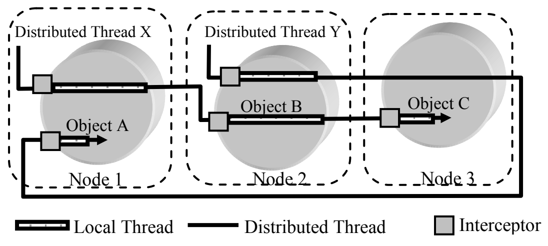


Figure 1. Distributed threads and local segments.

A source node is where a DT is created. Segment nodes are all nodes that host part of the execution of a DT. A DT should be eligible for execution (or suspended) at only one node in the distributed system, at any instant of time. This segment node receives the name of head node. Whenever a DT arrives at a node it means that it will execute a method in this node. When a DT makes a remote invocation it means that it departs from a node.

The DT carries parameters and other attributes of the computation task when it traverses a node. A DT executes operations inside distributed objects whose attributes can be modified and accumulated, in a nested way [1]. Local scheduling policies are used to schedule a DT when it begins executing in a system's node. Coherence should be maintained by the end-to-end scheduling model among all scheduling policies in each segment node of the DT. Real-time DT can share physical resources (e.g. processor, disk, I/O) and logical resources (e.g. locks), which can be subject of mutual exclusion constraints [4]. A program consists of multiple DTs executing concurrently and asynchronously.

The execution of a DT may terminate in the same node where it began, or in another node of the system. A DT that is executing can create a new real-time DT (e.g. through one-way invocations). This new real-time DT may begin executing in another node of the system. However, this kind of invocation is not considered in this work.

4. End-to-End Deadline Partitioning

Usually, an end-to-end DT deadline is defined as part of the system requirements. This time constraint and other information, as the sequence of nodes traversed in a given activation, can be stored in a history of each DT. This history is carried with the DT, as it traverses the nodes of the system, and it is updated at each new activation.

Different approaches for end-to-end deadline partitioning are presented in literature. In this work we consider only deadline partitioning methods that do not use information about the system load. Although the use of information on system load may improve the deadline partitioning, it also requires a more sophisticated infrastructure and additional overhead. Ultimate Deadline (UD) is a very simple method that assigns the end-to-end deadline to each local DT segment. Considering local deadline (dl), end-to-end deadline (d) and local segment (si), UD is defined as:

$$\mathrm{dl(si)} = d(DT)$$

The Effective Deadline (ED) method [3] is an improvement in relation of the previous method. The estimated computation times of local segments are used in the definition of local deadlines. Considering predicted execution time (*pex*) and DT's local segments (*m*), ED is defined as:

$$\mathrm{dl(si)} = d(DT) - \sum_{j=i+1}^{m} pex(sj)$$

A disadvantage of this technique is that the first DT local segment receives all the available slack. It makes the other local segments to not receive enough slack to execute, causing the possible missing of their deadlines. Equal Slack (EQS) [3] is a method where the slack is equally divided among all DT local segments. Considering arrival time (*ar*), EQS is defined as:

$$\mathrm{dl(si)} = ar(si) + pex(si) + [dl(TD) - ar(si) - \sum_{j=i}^{m} pex(sj)]/(m-i+1)$$

Another method is called *Equal Flexibility* (EQF) [3] which proposes that the total remaining DT slack should be divided among its local segments in the proportion of their estimated execution times. EQF is defined as:

$$\mathrm{dl(si)} = ar(si) + pex(si) + \left(d(TD) - ar(si) - \sum_{j=i}^{m} pex(sj) \right) * \left(pex(si) \Big/ \sum_{j=i}^{m} pex(sj) \right)$$

In the EQF method, each local segment receives enough slack to execute because the slack definition considers the estimated execution time of this segment in relation to others in a proportional way. According to [3], DTs with end-to-end deadlines partitioned by EQF have a better chance of meeting their deadlines when compared with other methods. This occurs because EQF considers the slack of the DT and divides the slack proportionally to the execution time of each local segment. That study considered only non-preemptive scheduling of pipeline-like DTs.

5. Proposed Model

In this work a distributed real-time system like those in factory automation is considered. At a given moment only one application executes in all nodes of the system and real-time DTs are used in the implementation of this application. The consequences of a timing fault of the DT are of the same order of magnitude of the benefits of the system when in normal operation.

There are periodic local tasks with hard deadlines and aperiodic distributed tasks with firm deadlines in each node of the system. The scheduling of the aperiodic distributed tasks must not jeopardy the scheduling of the periodic local tasks, because local tasks are considered critical for the application. These aperiodic distributed tasks are recurrent, that is, they may execute several times and they have end-to-end timing constraints. Since they can be created at runtime, this kind of system has a dynamic load and there may be overload situations.

In this work, whenever a real-time DT is created, an end-to-end deadline and an estimated average execution time (ACET) are defined. Each real-time DT carries these time constraints when it traverses system nodes. There is a scheduler in each system's node. These schedulers do not collaborate with each other in an explicit way and they are considered independent. The scheduling is influenced only by the timing attributes associated to each DT.

Aperiodic servers are used in each node of the distributed real-time system. In each node of the system the corresponding architecture is shown in Figure 2.

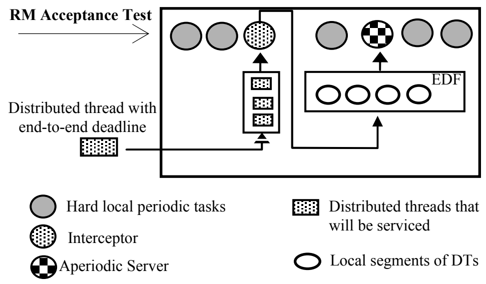

RM Acceptance Test
Distributed thread with end-to-end deadline
EDF
Hard local periodic tasks
Interceptor
Aperiodic Server
Distributed threads that will be serviced
Local segments of DTs

Figure 2 - System architecture for scheduling DT.

The interceptors are responsible for servicing the aperiodic real-time DTs, and they are local threads with execution times previously assigned, that is, they are considered as one additional periodic task of the system. Aperiodic servers carry out the execution of the local segments of DTs.

In this system it will be used the Rate Monotonic (RM) [5] algorithm which will preemptively schedule the periodic local tasks with hard deadlines, as well as the interceptor and the aperiodic server. When an aperiodic real-time DT arrives at a node (head node), the interceptor services it. A list of local segments of the real-time DTs that execute (or executed) in the node is maintained by each interceptor. For each aperiodic real-time DT 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 real-time DT that arrived. Otherwise, the interceptor creates a local segment with the function of executing on behalf of its parent DT.

In these two situations, all the properties of the aperiodic real-time DT (such as an identifier and timing constraints) are inherited by the local segment of the DT. These properties are stored in a history maintained by each local segment of a real-time DT. The last activations of the DT are also stored in this history.

The interceptor puts this local segment in the aperiodic server's queue for that node. The Earliest Deadline First (EDF) algorithm [5] is used to preemptively schedule the aperiodic server's queue.

A DT is autonomous in the sense that its future execution flow is determined at runtime as soon as it traverses system nodes executing remote calls. For example, node 1 has method A, node 2 has method B and node 3 has method C (Figure 3). If a DT begins executing in node 1, it can go to node 2 (method B) or node 3 (method C), according to system behavior.

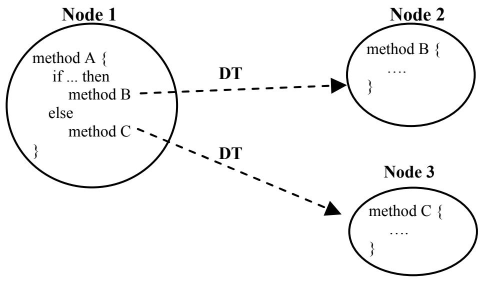


Figure 3 – DT execution flow alternatives.

Based on the DT history, the itinerary of the next activation of the DT, in spite of not being totally known, it is not completely random. The end-to-end deadline partitioning of the DT can be made considering this probabilistic knowledge of the future itinerary.

There are several scheduling approaches for DTs in the literature [1][4]. The scheduling approach used in this work is composed by two stages: partitioning of the end-to-end deadline and local scheduling. This is not an uncommon procedure in the real-time literature [9]. 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.

6. End-to-End Deadline Partitioning for Probabilistic Known Itineraries

In this work, the history of previous activations of each DT is kept stored. We can consider that the future itinerary of DTs is probably known. The end-to-end deadline partitioning of the DT can be made considering this probabilistic knowledge of the future itinerary that it will carry out.

6.1 Longest

This strategy searches in the DT history the longest itinerary that it traversed and the end-to-end deadline partition is carried out based on it (Figure 4.a). A problem that appears when this strategy is used is that the end-to-end deadline will always be partitioned considering the largest itinerary executed by the DT. Obviously, it will not always execute such itinerary. It is the application that chooses which way to follow. The itinerary that the DT executes will vary in accordance with the system dynamics.

6.2 Shortest

This strategy searches in the DT history the shortest itinerary that it traversed and the end-to-end deadline partition is carry out based on it. It is very similar to the previous approach.

6.3 Most Likely

This strategy searches in the DT history the most executed itinerary, and the end-to-end deadline partitioning is carried out considering the nodes related to this itinerary. A probabilistic knowledge about this itinerary is used, as Figure 4.b shows. When the DT is in Node 1, it has 80% of probability to go to Node 3 and 20% of probability to go to Node 2. With this strategy, the end-to-end deadline partitioning will be carried out as most likely itinerary, that is, Node1, Node 3 and Node 4. The main motivation for the use of this strategy comes from the fact that, in this work, the DTs represent supervision tasks that collect information for analyses and diagnoses and, therefore, they will execute the same itinerary most of the time.

6.4 Average

This strategy searches in the DT history the many itineraries executed by this DT. The end-to-end deadline partitioning is carried out considering an average of the task past behavior. The different probabilities of the DT to go to one or to the other path are taking into account.

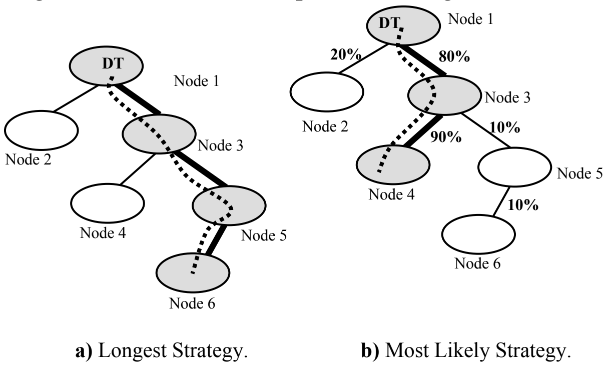


Figure 4 – Deadline Partitioning Strategies.

Many deadline partitioning methods can be applied in conjunction with these four strategies. All the partitioning calculations are carried out at the arrival of each DT activation, before it begins to execute. Longest strategy can be used in conjunction with EQF end-to-end deadline partitioning method, for example. In this way, the path with greater number of nodes is chosen and EQF end-to-end deadline partitioning is carried out.

One of the four strategies is defined for deadline partitioning, but because it is autonomous and traverses the nodes according to application dynamics, maybe it doesn't follow this pre-determined itinerary. In this situation, a new partitioning needs to be computed. When the system observes that the DT is in a node that don't belong to the original itinerary defined by the deadline partitioning strategy, a new deadline definition is made through the same strategy last used starting from the present node. An important aspect to be considered is when a local segment misses its deadline.

7. Proposed Deadline Missing Prediction Mechanisms

For each arrival of a DT in the system, a deadline missing prediction can be used to determine the probability of its end-to-end deadline to be met. This kind of prediction is important in distributed real-time systems because it allows the possibility of taking remedial actions to improve the system performance. Deadline adjustment is an example of remedial action that can be done.

In this work, the deadline missing prediction is carried out with the definition of estimated local deadlines, called milestones. To the end of the DT execution, the accurate of the prediction is analyzed. Some well known deadline partitioning methods can be used to create estimated local deadlines. In this work, we use the following methods as end-to-end deadline missing prediction mechanisms: Ultimate Deadline (UD), Effective Deadline (ED), Equal Slack (EQS) and Equal Flexibility (EQF) [3]. Our goal is to analyze if one of them presents reliable results enough to permit remedial actions to improve the system performance.

The deadline partitioning methods are used in this work for two purposes: local scheduling and deadline missing prediction. EQF method is used for local scheduling purpose. Thus, the DT end-to-end deadline is partitioned by this method, creating local deadlines that will be used by local schedulers in each node of the system. UD, ED, EQS and EQF methods are used for deadline missing prediction purpose. Through these methods, milestones will be created and used to compare the predictions with what actually happens.

It is necessary to define a metric to be used to compare the quality of the end-to-end deadline missing predictions done by several mechanisms in a given system. In this work we use the Relative Error Rate E(z) [12] observed for each considered mechanism.

At each arrival of a distributed thread k (DTk) in the system, each end-to-end deadline missing prediction mechanism z under evaluation calculates the probability Pk(z) of this DT to meet its end-to-end deadline, $0 \leq Pk(z) \leq 1$. DTk is executed and its effective response time Rk is measured.

Let be Ek(z) the error associated with the prediction of the response time done by mechanism z for the DTk, and dlk the local deadline defined through end-to-end deadline partitioning methods. Ek(z) is defined as:

$Ek(z) = 1 – Pk(z)$ case $Rk \leq dlk$;

$Ek(z) = Pk(z)$ case $Rk > dlk$.

The value of Ek is necessarily between 0 and 1. The Pk(z) variable is defined as:

$Slack = (Mk – Rk) / Mk$;

$Pk(z) = Slack + 0.5$;

If $Pk(z) > 1$ then $Pk(z) = 1$;

If $Pk(z) < 0$ then $Pk(z) = 0$;

The *Slack* variable represents the distance of the local prediction (Mk) in relation to local response time (Rk). If the value of *Slack* is zero, that is, the values of Mk and Rk are equal, then Pk(z) = 0.5. This means that DTk still has 50% of probability to meet its end-to-end deadline. The situations below illustrate the behavior of this metric:

- The mechanism z is sure that DTk will miss its deadline, Pk(z)=0, and it really misses, we have the relative error Ek(z) = Pk(z) = 0;
- The mechanism z determines that there is a chance of 80% for DTk to meet its deadline, Pk(z) = 0.8, and it meets its deadline, we have the relative error Ek(z) = 1 - 0.8 = 0.2;
- The mechanism z determines that there is a chance of 80% of DTk to meet its deadline, Pk(z) = 0.8, but the task misses its deadline, we have the relative error Ek(z) = 0.8;
- The mechanism z determines that there is a chance of 50% of the DTk to meet its deadline, Pk(z) = 0.5, and in this case we will have Ek(z) = 0.5 independently of the task meeting (1-0.5) or not (0.5) its deadline.

Let be nk the number of DTs in the system. The relative error rate of a given mechanism z is defined as:

$E(z) = \sum$ all k $Ek(z) / nk$.

It is important to observe that this metric considers the confidence of a mechanism on the meeting or not of a given deadline. For example, suppose that for DTk we have the predictions of two mechanisms y and z, being them: Pk(y) = 0.6 and Pk(z) = 0.8. In case DTk meets the deadline, we will have Ek(y) = 0.4 and Ek(z) = 0.2. As nk increases, we have a measure of the capacity of each mechanism in doing correct predictions about the response times. A clairvoyant mechanism would generate a relative error rate equal to zero along its execution.

8. Simulations

The simulation objective is to compare the use of the end-to-end partitioning methods, Ultimate (UD), Effective Deadline (ED), Equal Slack (EQS) and Equal Flexibility (EQF) in order to evaluate them as end-to-end deadline missing prediction mechanisms. In the case of pipeline-like DTs, method EQF is used for local scheduling, defining local deadlines from the original end-to-end deadline. For deadline missing prediction purpose, the UD, ED, EQS and EQF mechanisms are used to the milestones definition.

In the case of DTs with probabilistic knowledge of their future itinerary, we use the strategies defined in this work (Longest, Shortest, Average and Most Likely) in conjunction with UD, ED, EQS and EQF methods for the purpose of end-to-end deadline missing prediction. In this way we propose the following combination of deadline missing predictions:

- Ultimate;
- ED-Longest (ED-L), ED-Shortest (ED-S), ED-Average (ED-A), ED-Most Likely (ED-M);
- EQS-Longest (EQS-L), EQS-Shortest (EQS-S), EQS-Average (EQS-A), EQS-Most Likely (EQS-M);
- EQF-Longest (EQF-L), EQF-Shortest (EQF-S), EQF-Average (EQF-A), EQF-Most Likely (EQF-M).

As Ultimate deadline partitioning is a method that doesn't divide the end-to-end deadline, that is, it only assigns the end-to-end deadline for each local DT segment, it was not combined with any strategy. EQF-Longest is used for local scheduling purpose.

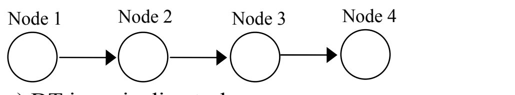
In this work we are proposing the use of the Sporadic Server [5] for the scheduling the local segments of the aperiodic DT. The aperiodic server queue will be preemptively scheduled by the EDF [5] algorithm. Three different load scenarios will be simulated and analyzed in the sequence of this text. The conditions of the simulation are described bellow.

8.1. Simulation Conditions

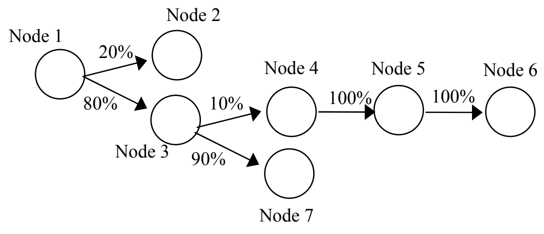
The system is composed of seven interconnected nodes. For each system's node, the use of the processor is 50% for periodic local tasks with hard deadlines and 50% for the aperiodic server. There are four hard periodic local tasks in each node, whose periods are distributed uniformly between 10 and 100. Their utilizations are of 20% for the task with the smallest period and 10% for each one of the other three. Those tasks have relative deadlines equal to their periods.

There are eight aperiodic real-time DTs composed by several local segments previously known. These aperiodic real-time DTs are of two types (Figure 5). The first one is a pipeline-like task graph (Figure 5.a) and the second is a Probabilistic-OR task graph - POR (Figure 5.b). We have four DTs of the pipeline type and four of the POR task graph type.

The aperiodic real-time DTs always execute a method in the node where it was created. These DTs traverse some system nodes to collect information. The inter-arrival time of these DTs follows an exponential distribution of 1000 units of time - u.t. and the WCET of each segment is different to each other, varying from 5 u.t. to 60 u.t. It is assumed that there is no network partition and the communication network delay has uniform distribution between 1 u.t. and 2 u.t. A Sporadic Server is used, with capacity of 5 u.t. and period equal to 10 u.t. 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.



a) DT in a pipeline task.



b) DT in a POR task graph.

Figure 5 – Distributed Threads types of the system.

All DTs of the pipeline type have 7 local segments, where segments 1 to 4 represent execution path on the nodes 1 to 4. Segments 5 to 7 represent execution path on the nodes 3 to 1, that is, return of DT to the source node (Figure 6).

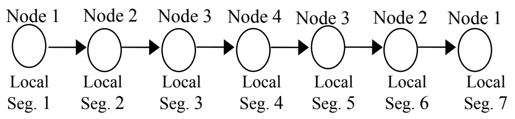


Figure 6 – Local Segments of the DTs of the system.

In the simulations with pipeline-like DTs, we focus our attention on the execution results of local segment 4 because it represents the middle of the pipeline. If an accurate prediction is carried out at this point, remedial actions can be made in the sense of improving system performance.

On the other hand, in the simulations with DTs of POR task graph type, we focus our attention on the DT execution results when it reaches half of its end-to-end deadline. For example, if DT's end-to-end deadline is 300 u.t., we observe the predictions on the node that this value is 150 u.t.

8.2. Simulation Results

Tables 1, 2 and 3 present the simulation results for DTs of the pipeline type. Three types of load distribution were used. In the first case, almost all end-to-end deadlines are met. In the second case there is a balanced load distribution, where the number of met and missed end-to-end deadlines is very close. In the last case, almost all end-to-end deadlines are missed. For each type of load distribution we have 3 measurements: *Right*, *Wrong* and *Error*. Correct predictions done by the mechanisms proposed are represented through *Right* measurement. Mispredictions done by mechanisms proposed are represented through *Wrong* measurement. Error represents the error rate Ek(z) and is defined in section 6. The deadline missing prediction mechanisms (UD, ED, EQS and EQF) were simulated for the three types of load distribution defined above. Also, we have simulated the arrival of 100 DTs activations approximately. The arrival pattern of the many DTs is the same for all algorithms tested.

Tables 1, 2 and 3 show the results. When the system has a light load (Table 1), where almost all end-to-end deadlines are met, the deadline missing prediction mechanisms present results very close. None of them was shown to be consistently superior to the others. The Error rate of the UD and ED are smaller than EQS and EQF. This is because UD and ED can be considered optimistic algorithms, that is, they believe that DTs will meet their end-to-end deadlines and this really occurs in this scenario.

When the system have a balanced load (Table 2), where the number of met and missed end-to-end deadlines are very close, we observed that EQF prediction mechanism presents better results in relation to the others (UD, ED and EQS). While UD did 10 correct predictions and 13 mispredictions, EQF did 23 correct predictions and any misprediction. Moreover, the Error rate of the EQF also is better in relation to the others. The results obtained by EQS are close to EQF, but EQF indeed is better.

Table 1 – Light load, end-to-end deadline = 560

	MET	MISS	RIGHT	WRONG	ERROR
UD	28	7	28	7	0,19
ED	28	7	29	6	0,17
EQS	28	7	29	6	0,16
EQF	28	7	31	4	0,36

Table 2 – Balanced load, end-to-end deadline = 520

	MET	MISS	RIGHT	WRONG	ERROR
UD	10	13	10	13	0,52
ED	10	13	12	11	0,38
EQS	10	13	15	08	0,30
EQF	10	13	23	0	0,22

Table 3 – Heavy load, end-to-end deadline = 480

	MET	MISS	RIGHT	WRONG	ERROR
UD	4	18	6	16	0,67
ED	4	18	9	13	0,45
EQS	4	18	11	11	0,34
EQF	4	18	19	3	0,19

A good result in favor of EQF also was observed when the system has a heavy load (Table 3). This case is especially interesting because it shows that EQF is a good enough algorithm for overloaded systems, when special actions need to be executed to improve system's performance. Unlike UD that can be considered an optimistic algorithm, EQF can be seen as pessimistic algorithm, that is, it believes that almost all DTs will miss their end-to-end deadlines and this really occurs in fact in this scenario.

Simulations also were carried out with DTs of POR task graph type and the tables 4, 5 and 6 show the results. The same types of load distribution were used. The end-to-end deadline partitioning methods (ED, EQS and EQF) was combined with the strategies proposed in this work (Longest, Shortest, Average and Most Likely) to compose new end-to-end deadline missing predictions mechanisms.

Tables 4, 5 and 6 show similar results to those obtained in tables 1, 2 and 3. Again, interesting results appear when the system is overloaded (Table 6). EQF-Longest and EQF-Average present best results as end-to-end deadlines missing prediction mechanisms among all studied. In 33 DTs activations, EQF-Average makes 26 correct predictions while the other mechanisms make an average of 10 correct predictions. The Error rate of these mechanisms also show better values in relation to the other.

Table 4 – Light load, end-to-end deadline = 380

	MET	MISS	RIGHT	WRONG	ERROR
UD	23	11	23	11	0.32
ED-L	23	11	24	10	0.29
ED-S	23	11	23	11	0.32
ED-A	23	11	23	11	0.32
ED-M	23	11	23	11	0.32
EQS-L	23	11	22	12	0.36
EQS-S	23	11	24	10	0.29
EQS-A	23	11	24	10	0.31
EQS-M	23	11	24	10	0.31
EQF-L	23	11	19	15	0.56
EQF-S	23	11	22	12	0.36
EQF-A	23	11	21	13	0.38
EQF-M	23	11	21	13	0.38

Table 5 – Balanced load, end-to-end deadline = 300

	MET	MISS	RIGHT	WRONG	ERROR
UD	20	17	20	17	0.44
ED-L	20	17	21	16	0.42
ED-S	20	17	22	15	0.42
ED-A	20	17	22	15	0.41
ED-M	20	17	22	15	0.41
EQS-L	20	17	27	10	0.41
EQS-S	20	17	21	16	0.43
EQS-A	20	17	21	16	0.43
EQS-M	20	17	21	16	0.43
EQF-L	20	17	27	10	0.41
EQF-S	20	17	24	13	0.33
EQF-A	20	17	25	12	0.3
EQF-M	20	17	24	13	0.33

Table 6 – Heavy load, end-to-end deadline = 250

	MET	MISS	RIGHT	WRONG	ERROR
UD	7	26	9	24	0.73
ED-L	7	26	7	26	1.86
ED-S	7	26	11	22	0.68
ED-A	7	26	11	22	0.66
ED-M	7	26	11	22	0.67
EQS-L	7	26	7	26	34.74
EQS-S	7	26	12	21	0.64
EQS-A	7	26	12	21	0.64
EQS-M	7	26	12	21	0.64
EQF-L	7	26	26	7	0.21
EQF-S	7	26	17	16	0.33
EQF-A	7	26	27	6	0.2
EQF-M	7	26	17	16	0.33

These experiments demonstrated that using EQF as prediction mechanism in overloaded systems it is possible to take remedial actions to improve the system's performance. An example of remedial action could be to increase the end-to-end deadline of a DT for this task to meet its end-to-end deadline. Another possibility could be to discard this DT in order to reduce system load and increase the overall met rate.

9. Conclusions

Early predictions allow carrying out actions to improve system performance. In this work, it is addressed the problem of deadline missing prediction from local deadlines of DTs with probabilistic knowledge of its future itinerary. We proposed mechanisms to predict the probability of a DT deadline to be met.

It was used a system architecture that offers support to real-time DTs. The architecture goal is to provide a guarantee for hard local periodic tasks, while trying to meet the end-to-end firm deadline of each DT. Two types of DTs were used: pipeline task graph and probabilistic OR task graph. New strategies were proposed for partitioning the end-to-end deadlines of DTs considering probabilistic knowledge about its future itineraries: Longest, Shortest, Average and Most Likely.

The proposed deadline missing prediction mechanisms carry out predictions from the creation of milestones (estimated local deadlines).

Simulation experiments were done in order to analyze the behavior of these mechanisms with different load systems. The comparison among the proposed prediction mechanisms was done with the error rate metric. EQF mechanism shows interesting results in overloaded systems. Because of EQF mechanism considers the computation times of DT's local segments in the creation of milestones, they give a more realistic view of the system load and provide better predictions. EQF-Longest mechanism presents better results in relation to the other in case of DTs of the POR task graph type. This is why EQF-Longest considers the longest path in the task graph (in terms of nodes number) in the creation of milestones. Thus, independently of what itinerary the DT will traverse, the prediction is done considering the largest path.

Acknowledgement

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

References

1. Clark, R.K., Jensen, E.D., Reynolds, F.D.: *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.
2. Jonsson, J., Shin, K. G.: *A Robust Adaptive Metric for Deadline Assignment in Heterogeneous Distributed Real-Time Systems*. In: Proc. IEEE International Parallel Processing Symposium, pp. 678-687, 1999.
3. Kao, B., Molina, H.G.: *Deadline Assignment in a Distributed Soft Real-Time System*. In: IEEE Trans. Parallel and Distributed Systems, vol. 8, nb. 12, pp.1268-1274, Dec 1997.
4. Li, P., et al.: *Scheduling Distributable Real-Time Threads in Tempus Middleware*. In: Proc. ICPADS, Newport Beach, California, pp. 187, Jul. 2004.
5. Liu, J.W.S.: *Real-Time Systems*. Prentice Hall, 2000.
6. Marinca, D., Minet, P., George, L.: *Analysis of Deadline Assignment Methods in Distributed Real-Time Systems.* In: Computer Communications, Vol.27, issue 15, June 2004.
7. Natale, M. D., Stankovic, J. A.: *Dynamic End-to-End Guarantees in Distributed Real-Time Systems*. In: Proc. RTS Symposium, San Juan, Puerto Rico, 7-9 Dec, 1994.
8. Plentz, P., Oliveira, R. S., Montez, C.: *Scheduling of the Distributed Thread Abstraction with Timing Constraints using RTSJ.* In: ETFA'2005, Catania, Italy, 19-22 September 2005, pp. 23-30.
9. Sun, J.: *Fixed-Priority End-To-End Scheduling in Distributed Real-Time Systems*. Thesis, Graduate College, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1997.
10. Zhang, J., et al: *A Real-Time Distributed Scheduling Service for Middleware Systems*. In: Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, pp. 59-65, 2005.
11. Zhang, Y., et al: *A Real-Time Performance Comparison of Distributable Threads and Event Channels*. In: RTAS'2005, pp 497-506.
12. Tatibana, C. Y., Montez, C., Oliveira, R. S. *Soft Real-Time Task Response Time Prediction in Dynamic Embedded Systems*. In: Proc. 5th IFIP Workshop Software Technologies for Future Embedded & Ubiquitous Systems, 2007.
13. Panagos, E., Rabinovich, M. *Reducing Escalation-Related Costs in WFMSs*. In: A. Dogac et al., editor NATO Advanced Study Institute on Workflow Management Systems and Interoperability. Springer. Istanbul, Turkey. 1997.