A solution preserving consistency of replicated objects with hard real-time constraints*

Laurent George†
ESIGETEL
1 rue du port de Valvins
77215 Avon, France

Pascale Minet†
INRIA
Rocquencourt
78153 Le Chesnay cedex, France

Keywords replication, real-time scheduling, one-copy serializability, active redundancy, worst case response time.

Abstract

In this paper, we are interested in designing a solution for a hard real-time problem with consistency constraints in a distributed system, where fault-tolerance is achieved by active redundancy. Hard real-time constraints are expressed by tasks termination deadlines. Each task must preserve the consistency of persistent objects stored on distributed processors. For availability reasons, objects are replicated. Our solution is based on the concept of classes. Intuitively, the notion of class captures the set of tasks having, direct or indirect, conflicting accesses to objects. Tasks are scheduled with Earliest Class Deadline First (ECDF), an adaptation of Earliest Deadline First scheduling, where a task has for inherited deadline its release time plus its class deadline. We establish feasibility conditions, under which tasks meet their deadlines. The feasibility conditions complexity is pseudo-polynomial, when processor utilization is strictly less than one.

1 Introduction

Replication has been introduced to improve availability. However the design of a fault-tolerant algorithm that (i) preserves the consistency of replicated persistent objects and (ii) meets hard real-time constraints, is still an open issue. The consistency of replicated objects is enforced by one-copy serializable executions of tasks. Hence the only executions of tasks provided by our algorithm are one copy serializable [3], even in presence of processor crashes. Hard real-time constraints are expressed by tasks termination deadlines. We determine the conditions under which the tasks meet their deadline. These conditions are called feasibility conditions. They are the result of a worst case response time analysis.

There are several approaches to update all the copies of a replica. One approach consists in running a distributed task [3], in charge of updating all the copies of that replica. The drawback is that the distributed task requires a synchronization between all the processors in charge of running it, to decide whether the task is committed or not. Another approach is the lazy master replication approach [15], which is easier to deploy. Updates are made only on the primary copy, secondary copies being refreshed later on. The main concern is then the freshness of the copies.

Like in the distributed task approach, we do not distinguish between primary copy and secondary copies. A task is run in several copies, one copy per processor having a replica to be updated. Like in the lazy master replication approach, each copy of the task runs locally. We take advantage of the analysis of conflicts between tasks to define classes of tasks. The execution order of two tasks belonging to the same class determines the order of the equivalent serial execution. Two tasks of different classes can be executed in any order without impacting the order of the equivalent serial execution. Hence the only requirement enforced by one copy serializability, is the existence of a global total order per class. The global total order per class is built on top of a reliable multicast algorithm ensuring an upper bounded response time, without exchanging additional messages. In our approach, tasks are scheduled with Earliest Class Deadline First (ECDF), an adaptation of Earliest Deadline First scheduling. In this paper, we are especially interested in applications such as online auctions or stock market transactions, where it is required to schedule tasks in the same class according to their release order, to achieve fairness among tasks initiators.

To minimize the number of processors, a proces-

*Published in ISCA 12th int. conf. on Parallel and Distributed Computing Systems PDCS'99, Fort Lauderdale, Florida, USA, August 1999

†{laurent.george, pascale.minet}@inria.fr

sor can run tasks belonging to different classes[1]. We then have to solve a scheduling problem between tasks belonging to different classes. On the one hand, the execution order of two tasks belonging to different classes does not compromise objects consistency. On the other hand, tasks are constrained by termination deadlines, we have chosen a scheduling that accounts for the different deadlines between classes, i.e ECDF.

The paper is organized as follows. Section 2 is concerned with the problem specification in terms of assumptions and required properties. Section 3 briefly discusses related work. Section 4 presents our solution based on the concept of tasks classes. In Section 5, a uniprocessor analysis is given. It enables to compute the worst case response times of a sporadic task set scheduled according to ECDF. This analysis is then extended to the distributed case. The complexity of the feasibility conditions is given.

2 Problem specification

2.1 Assumptions

We investigate the problem of scheduling a set $\tau = \{\tau_1, ..., \tau_n\}$ of n sporadic tasks. Each task being replicated on different distributed processors. Each copy of a task is run locally. Any task τ_i is defined by:
• T_i, the minimum interarrival time between two successive requests of task τ_i. In the following, T_i is called the period.
• D_i, the relative deadline of task τ_i. A task τ_i requested at time t has for absolute deadline $t + D_i$, ($t + D_i$ is the latest completion time of that task's instance).
• C_i^j, the maximum processing time of the copy of task τ_i run on processor j. Notice that C_i^j is a function of processor j, as processors can be heterogeneous.
• The objects accessed by τ_i, with their access mode.
• f_i, the maximum number of crashed processors, that the execution of task τ_i must tolerate.
Moreover, we assume that:
• The copy of task τ_i, run on processor j, accesses the local copy of persistent objects. Objects are bound by consistency constraints. Initially, these consistency constraints are met. Each task, taken alone and run until completion, preserves the consistency constraints.
• The processing of a task is assumed to be non-preemptive. The times when tasks are requested, are not known a priori.
• Processors can fail by crashing. In this paper, we do not study processor recovery.
• A bounded number of network omissions can be tolerated, assuming a uniform reliable multicast protocol [12] ensuring a transmission delay bounded by Max [8]. We assume that the number of processor crashes tolerated by that protocol is never exceeded.
• Each processor has its local clock. Clocks have a bounded drift and are ε-synchronized, where ε is the clock precision. Time is assumed to be monotonically increasing (see [16]).

[1] As the replication degree depends on the class, all the processors do not necessarily run the same classes.

2.2 Properties

The critical real-time system we consider, must ensure the following properties:
• *(P1) For each task τ_i whose initiator does not crash, at least one copy runs until completion, even in case of f_i processor crashes.*
• *(P2) For each task, the values read and the values written by each copy that completes, are consistent.*
• *(P3) For each task, each copy which runs until completion, completes at the latest at the tasks's deadline.*

3 Related work

This paper addresses three topics: real-time scheduling, concurrency control in presence of replicated objects and fault-tolerance based on active redundancy. These three topics have been first studied separately. Real-time uniprocessor scheduling has been extensively studied:
• Fixed priority scheduling: e.g., Deadline Monotonic /Highest Priority First and optimal priority assignment [2, 18];
• Dynamic priority scheduling: e.g. Earliest Deadline First ([13, 4, 17]), FIFO [7], Shortest Slack Time [14].

Concurrency control in the presence of replicated data has been largely investigated [3, 15]. These algorithms differ (i) by the consistency achieved (e.g., one copy serializability in [3]), (ii) by the way they update copies (e.g. the Read One/Write All approach, the primary copy approach, the quorum based approach [11]), (iii) by their behavior w.r.t. system partitioning [1].

When fault-tolerance is achieved by active redundancy [10], every server handles the requests received from the clients, and provides a reply. To be correct,

this technique must ensure that all the non-crashed servers receive the same requests and in the same order. For that purpose, an atomic multicast is often used. A lot of atomic multicast algorithms exist (e.g. [12, 19, 5]). Here, we only use a uniform reliable multicast ensuring an upper-bounded response time [8] and built on top of it, a global total order per class (see [12] for a definition).

4 Our algorithmic solution

In the absence of failures, a sufficient condition, called one copy serialisability, to preserve the consistency of replicated objects is given in [3]. In this section, we formally define the concept of class of tasks. We then show how the one copy serializability, instead of being analyzed at the level of the whole set of tasks, can be analyzed independently at the level of each class (subsection 4.1). In subsection 4.2, we justify the choice of ECDF (Earliest Class Deadline First). The algorithm is given in subsection 4.3. Finally, we prove in subsection 4.4, that even with crash failures, the consistency of replicated objects is not compromised.

4.1 One copy serializability and classes

One copy serializability means that the tasks read and write the same values as if there were only one copy of objects, and tasks were serially executed. Now, we show how the problem of one copy serializability at the level of the set τ of tasks, can be decomposed into independent subproblems at the level of classes of τ. Classes of tasks, are such that two tasks τ_i and τ_j belonging to two different classes can be run in any order: the values read and the values written by respectively τ_i and τ_j are the same when τ_i precedes τ_j, and when τ_j precedes τ_i. Intuitively, the notion of class captures the direct or indirect conflicting accesses to objects between tasks. More precisely, we define the relation *conflicts with* between tasks:

Definition 1 *Let τ_i and τ_j be two distinct tasks of τ, τ_i* conflicts with *τ_j iff there exists a persistent object O such that the access modes of O by τ_i and τ_j are incompatible*[2].

We now consider *conflicts* with*, the reflexive and transitive closure of the relation *conflicts with*, which is symetrical. Hence *conflicts* with* is an equivalence relation.

Definition 2 *An equivalence class of the relation* conflicts* with *is called a class.*

Let us consider a set of tasks $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$, where $\tau_1 = [readO5, readO4, writeO1]$, $\tau_2 = [readO1, writeO2]$, $\tau_3 = [readO2, writeO3]$, $\tau_4 = [readO3, writeO4]$ and $\tau_5 = [readO5]$. Hence for $i \neq 5$, $class(\tau_i) = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, and[3] $class(\tau_5) = \{\tau_5\}$.
The set τ of tasks is then partitioned into classes. The execution order of tasks belonging to the same class determines the values read and written by these tasks. It determines the equivalent serial order.

Lemma 1 *An execution E of a set τ of tasks is one copy serializable iff its restriction to any class of τ is one copy serializable.*

Proof: We first prove that if E is one copy serializable, then for any class C of τ, the restriction of E to C is one copy serializable. By contradiction, suppose that there is a class C of τ, such that the restriction of E to C is not one copy serializable. From [3], there is a cycle in the replicated object serialization graph[4] associated with the restriction of E to C. Hence, E is not one copy serializable. A contradiction.
Suppose now that for any class C of τ, the restriction of E to C is one copy serializable, and E is not one copy serializable. From [3], there exists one cycle in the replicated object serialization graph at the system level. Since serializability is ensured for each class taken separately, this cycle does not involve one single class. By the construction of the classes, two tasks that conflict (directly or indirectly) belong to the same class. Hence, a contradiction. □

Lemma 2 *The replication degree of any task in $class(\tau_i)$, also called the replication degree of $class(\tau_i)$, is equal to $max_{\tau_j \in class(\tau_i)} f_j + 1$.*

Proof: For any task τ_i, let j be any processor running the copy τ_i^j of task τ_i. As τ_i^j runs locally, a copy of all the objects accessed by τ_i^j must exist on j. By recursion, a copy of all the objects accessed by any task of $class(\tau_i)$ must exist on j. Hence, the number of copies of any task in $class(\tau_i)$. □

Lemma 3 *If for any execution E of the set τ of tasks, for any class C of τ, for any processor j involved in*

[2] In the case of read or write accesses, two accesses to a same object are incompatible iff at least one access is a write.

[3] Notice that although τ_1 and τ_3 do not directly conflict, they belong to the same class because of τ_2.

[4] Intuitively, the replicated object serialization graph associated with an execution E, models how E has ordered tasks having conflicting accesses to a same object, even if it is not the same copy of the object.

C, the restriction of E to C on processor j is serializable and the equivalent serial order is the same for all processors involved in C, then one copy serializability is ensured.

Proof: a consequence of lemma 1 and lemma 2. □

4.2 Why ECDF scheduling?

In this section we first give the principles of ECDF and explain the choice of this scheduling algorithm.

• With ECDF, the next task to be run, is the one with the smallest inherited deadline. A task τ_i requested at time t_i has the inherited deadline $t_i + D_{class(\tau_i)}$, where $D_{class(\tau_i)}$ denotes the deadline of $class(\tau_i)$. The class deadline is the smallest relative deadline of tasks in that class[5].

We now show that in our context, the execution of tasks in the same class according to non-preemptive EDF based on the tasks deadlines does not meet one copy serializability.

Lemma 4 *For any class C of the set τ of tasks, if tasks in C have different relative deadlines and the workload is not the same on all the processors involved in C, then the execution of tasks in C according to non-preemptive EDF (Earliest Deadline First), does not ensure one copy serializability.*

Proof: Let us consider any two processors, p_1 and p_2 involved in two tasks τ_i and τ_j accessing the same object in incompatible modes, and such that the activation request of τ_i is received before the activation request of τ_j by both p_1 and p_2. Moreover, τ_j has a smaller absolute deadline[6] than τ_i. Let us suppose that p_2 is idle[7], when it receives the activation request of τ_i. Hence, it executes τ_i^2. Then it receives the activation request of τ_j, τ_j^2 is executed after τ_i^2 (non-preemptive effect). p_1, while executing a copy of task τ_k, receives the activation requests of τ_i and τ_j. According to EDF, τ_j^1 is executed before τ_i^1. Hence, a cycle in the replicated object serialization graph. □

The scheduling of tasks of different classes is not constrained by objects consistency, but only by tasks termination deadlines. That is why, we have chosen ECDF that selects for execution the task with the smallest inherited deadline.

[5]Within a class, the order of inherited deadlines is the release order.

[6]Such tasks exist, because tasks in C do not have all the same relative deadline.

[7]As processors can be heterogeneous and as they do not necessarily execute the same classes, their workload can differ. That is why, their pending queue can differ when they insert a new task, even if it is at the same time.

4.3 The algorithm

Informally, the solution is based on a uniform reliable real-time multicast ensuring an upper bound on the response times, denoted Max (e.g., [8]). A total order per class is then built. For a given class, this order reflects ECDF order. The scheduler selects then, among the heads of ECDF queues, the task with the smallest inherited deadline.

• A processor maintains one pending queue and as many ECDF queues as classes, the processor is involved in.

• Upon occurrence of an external/internal event triggering task τ_i, the initiator multicasts the activation request of τ_i to the f_c+1 processors that execute a copy of τ_i, where f_c is the replication degree of $class(\tau_i)$. This activation request is timestamped by its release time t_i.

• When a processor receives the activation request of task τ_i, released at time t_i, it inserts it into the pending queue, sorted by increasing release times. At local time $t_i + Max + \varepsilon$, it moves the activation request of τ_i from the pending queue to the ECDF queue associated with $class(\tau_i)$, sorted by increasing inherited deadlines.

• The next task to be run on a processor is the task with the smallest inherited deadline[8].

4.4 Proofs of properties

Let us prove that this algorithm meets the required properties P1, P2 and P3.

Lemma 5 *In the absence of failures, our solution achieves one copy serializability.*

Proof: Let E be any execution of the set τ of tasks. From lemma 1, we have only to prove that for any class C of τ, the restriction of E to C is one copy serializable. Let us consider any update task τ_i in C, released at time t_i. At local time $t_i + Max + \varepsilon$, any processor will no more receive tasks released at a time smaller than t_i, otherwise the assumptions w.r.t. Max and ε would be violated. Hence at that time, the activation request of task τ_i is inserted in the ECDF queue of $class(\tau_i)$. The insertion order in that queue is the release order of the tasks in $class(\tau_i)$. We then have a global total order per class that ensures the absence of cycle in the associated replicated object serialization graph. Hence, the restriction of E to C is one copy serializable. □

Notice that within a class, the serial equivalent order

[8]Such a task is at the head of one ECDF queue.

is the release order of the tasks in the system.

Proof of property P1: As (i) the initiator of task τ_i released at time t_i does not crash, (ii) the activation request of task τ_i is reliably multicast to $max_{\tau_j \in class(\tau_i)} f_j + 1$ processors, (iii) τ_i belongs to $class(\tau_i)$, f_i crashed processors do not prevent at least one processor j that does not crash, to receive the activation request of τ_i. At time $t_i + Max + \varepsilon$, processor j inserts τ_i^j in the ECDF queue associated with $class(\tau_i)$. From this time, up to the time when τ_i^j becomes the task with the smallest inherited deadline, processor j has to run a bounded number of tasks (an evaluation of this number is given in the computation of $L_i(a)$ in subsection 5.1.2). As (i) the execution of all these tasks is strictly local (no synchronization with other processors), (ii) their processing times by j are bounded, and (iii) processor j does not crash, j will complete the execution of τ_i^j. Hence, property P1. □

Lemma 6 *For any class C of τ, let j and k two processors involved in C such that j crashes and k never crashes. Then the sequence of tasks of C run by j is a prefix of the sequence of tasks of C run by k.*

Proof: We consider only tasks of C. We first prove that any task run by j, is run by k. By contradiction, let us suppose that task τ_i released at time t_i, is run by j and is not run by k. As the reliable multicast is uniform and upper bounded by Max, then processor k should have received the activation request of task τ_i at the latest, at time $t_i + Max + \varepsilon$. Hence, from P1, k should have run τ_i. A contradiction.
We now prove that if k has run τ_i released at time t_i before τ_m released at time t_m, and j has run τ_m, then j has run τ_i before τ_m. By contradiction, let us suppose that j has not run τ_i before τ_m. As $t_i < t_m$, and as a processor that crashes, does not recover, at time $t_m + Max + \varepsilon$, processor j should have received the activation request of τ_i, otherwise the assumptions w.r.t. Max and ε were violated. According to the algorithm, j should run τ_i before τ_m. A contradiction. □

Proof of property P2: In the absence of failure, one copy serializability is ensured (lemma 5). Hence, the objects are consistent. Now, suppose that failures occur. First, we consider only processors that do not crash. From P1, and as each copy of a task runs entirely on one processor, replicated objects consistency is maintained. Second, we consider processors that will crash. From lemma 6, it follows that a processor that crashes does not compromise replicated objects consistency. Consequently task copies that run until completion, provide consistent results. Hence property P2. □

Proof of property P3: See the next Section, that establishes the feasibility conditions under which any copy of τ_i that runs until completion, meets its deadline. □

Moreover, we can notice that the solution meets the following property:

Lemma 7 *Tasks belonging to the same class and whose release order differ more than the clock synchronization ε, are scheduled chronologically (i.e according to their release order).*

Proof: Let us consider two tasks τ_i and τ_j belonging to the same class and such that an external observer first sees the release of τ_i and d time units after, the release of τ_j with $d > \varepsilon$. Even if the clock of τ_j's initiator is ε-late with regard to the clock of τ_i's initiator, τ_j will get a timestamp higher than τ_i. Hence the scheduling order of τ_i and τ_j will reflect their release order[9]. □

5 Worst case response time analysis

5.1 Uniprocessor case

5.1.1 Concepts and notations

We consider non-preemptive ECDF. We first define the notations used for the uniprocessor scheduling analysis. Time is assumed to be discrete (task arrivals occur and tasks executions begin and terminate at clock ticks; the parameters used are expressed as multiples of the clock tick); in [4], it is shown that there is no loss of generality w.r.t. feasibility results by restricting the schedules to be discrete. We now introduce the busy period notion that identifies periods of activity of the processor. The aim of the analysis given in 5.1.2 is to identify the worst case busy periods (where the maximum response time of any task can be obtained). In the uniprocessor case, the notation is simplified: we use C_i instead of C_i^j.

- An idle time t is defined on a processor as a time such that there are no tasks requested before time t pending at time t. An interval of successive idle times is called an idle period.
- A busy period is defined as a time interval $[a, b)$ such that there is no idle time in the interval (a, b) (the processor is fully busy) and such that both a and b are idle times.
- $U = \sum_{i=1}^{n} C_i/T_i$ is the processor utilization factor, i.e., the fraction of processor time spent in the execution of tasks [13].

[9] If now $d \leq \varepsilon$, then the timestamps of τ_i and τ_j can be in any order, the scheduling order will reflect the timestamp order.

• Let time 0 be the beginning of a busy period bp. $\forall \tau_i$ run in that busy period and $\forall a \geq 0$, its activation request time, we define the sets: $hp_i(a) = \{\tau_k, k \neq i, D_{class(\tau_k)} \leq a + D_{class(\tau_i)}\}$ and $\overline{hp}_i(a) = \{\tau_k, class(\tau_k) \neq class(\tau_i), D_{class(\tau_k)} > a + D_{class(\tau_i)}\}$. From ECDF and by construction, if $\tau_k \in hp_i(a)$, at least the occurrence of task τ_k whose activation is requested at time 0 has a higher priority than task τ_i whose activation is requested at time a. All occurrences of tasks in $\overline{hp}_i(a)$ (if any) have a priority lower than task τ_i whose activation is requested at time a whatever their release time in bp.

5.1.2 Worst case response time computation

We now consider the problem of scheduling n sporadic tasks on a uniprocessor by ECDF. The worst case response time is reached for any task τ_i in a scenario described in lemma 8.

Lemma 8 *Let $\tau_i, i \in [1,n]$, be a task requested at time a and executed according to ECDF in a busy period bp. The worst case response time of τ_i requested at time a, $a \geq 0$, is obtained in the first busy period of a scenario where:*
• *any task $\tau_k \in hp_i(a)$ executed in bp is requested for the first time at time 0, and is then periodic.*
• *all the previous occurrences of τ_i executed in bp are periodic from $t_i^0 = a - \lfloor a/T_i \rfloor T_i$ to a.*
• *a task in $\overline{hp}_i(a)$ whose duration is maximum over $\overline{hp}_i(a)$ (if any) is released at time -1.*

Proof: Let τ_i be a task requested at time a and executed in a busy period bp beginning at time 0. The response time of τ_i is maximized, when the number of tasks of higher priority than τ_i that are executed before τ_i, is maximum. This leads to the scenario where tasks of higher priority are requested as soon as possible in bp (i.e at time 0 and then periodically). This corresponds to tasks $\tau_k \in hp_i(a)$ having by construction a higher priority than τ_i. Due to non-preemption, a task with a smaller priority than τ_i can still postpone task τ_i, if its execution begins when no higher priority task is pending. The non-preemptive effect is maximized, when such a task with a maximum duration is requested at time -1. □

Lemma 9 *The longest busy period L is obtained in the first busy period of the scenario where all tasks τ_k, $k \in [1,n]$, are requested for the first time at time 0 and then at their maximum rate.*

Proof: see [9]. □
L is the first solution of the equation:
$L = \sum_{i=1}^{n} \lceil L/T_i \rceil C_i$.
It can be easily shown that L always exists and is bounded by $lcm_{i=1..n}(T_i)$.

Computation of $L_i(a)$
• Let $L_i(a)$ denotes the time when τ_i's instance requested at time $a \geq 0$ can start its execution in a scenario of lemma 8.
• The response time of the instance of τ_i requested at time a is: $r_i(a) = max\{C_i, L_i(a) + C_i - a\}$. The value of $L_i(a)$ can be determined by finding the smallest solution of the equation
$t = max_{\tau_j \in \overline{hp}_i(a)}\{C_j - 1\} + \overline{W}_i(a,t) + \lfloor a/T_i \rfloor C_i$ (Eq.1).
• The first term on the right side accounts for the worst-case priority inversion w.r.t. the inherited deadline $a + D_{class(\tau_i)}$ for tasks in $\overline{hp}_i(a)$.
• The second term $\overline{W}_i(a,t)$ is the time needed to execute the instances of tasks other than τ_i with higher priority than τ_i. Its value is detailed later on.
• Finally, the third term is the time needed to execute the τ_i's instances requested before time a.
The rationale of Eq.1 is to compute the time b when τ_i's instance released at time a gets the processor. Every other higher priority instance released before b will be executed earlier, thus its execution time must be accounted for. For the same reason, the function $\overline{W}_i(a,t)$ must account for all higher priority instances of task $\tau_k \in hp_i(a)$.
• For any task $\tau_k \in hp_i(a)$, the maximum number of instances requested in $[0,t]$ is $1 + \lfloor t/T_k \rfloor$. However, among them at most $1 + \lfloor \frac{min(t, a + D_{class(\tau_i)} - D_{class(\tau_k)})}{T_k} \rfloor$ can have an inherited deadline less than or equal to $a + D_{class(\tau_i)}$. It follows that: $\overline{W}_i(a,t) = \sum_{\tau_k \in hp_i(a)} (1 + \lfloor \frac{min(t, a + D_{class(\tau_i)} - D_{class(\tau_k)})}{T_k} \rfloor) C_k$.
$\overline{W}_i(a,t)$ being a monotonic non-decreasing step function in t, the smallest solution of Eq.1 can be found using the usual fixed point computation:
$L_i^{(0)}(a) = 0$.
$L_i^{(p+1)}(a) = max_{\tau_j \in \overline{hp}_i(a)}(C_j - 1) + \overline{W}_i(a, L_i^{(p)}(a)) + \lfloor a/T_i \rfloor C_i$.

It can be shown [6] that for any task τ_i released at time a in a busy period of a scenario of lemma 8, we can restrict the computation of $r_i(a)$ to values of a smaller than $B_i(t_i^0) \leq L$, $B_i(t_i^0)$ being the maximum length of the first busy period of the scenario where all the tasks τ_k, $k \neq i$ are released at time 0 and task τ_i is first released at time $t_i^0 = a - \lfloor a/T_i \rfloor T_i$. $B_i(t_i^0)$ is the first solution of:
$B_i(t_i^0) = \sum_{k \neq i} \lceil B_i(t_i^0)/T_k \rceil C_k + \lceil \frac{B_i(t_i^0) - t_i^0}{T_i} \rceil C_i$.
$B_i(t_i^0)$ can be computed recursively. See [7] for a de-

tailed computation.
That is, the worst-case response time of τ_i is finally $r_i = max\{r_i(a),\ with\ 0 \leq a < B_i(t_i^0)\}$.

Notice that:
• when there is exactly one task per class, ECDF becomes EDF. The formula giving $\overline{W}_i(a,t)$ becomes: $\sum_{\tau_k \in hp_i(a)}(1 + \lfloor min(t, a + D_i - D_k)/T_k \rfloor)C_k$, which is exactly the result established in [9]. Nevertheless, non-preemptive EDF is not a solution for our problem (see section 4.2).
• when all the tasks belong to the same class, ECDF becomes FIFO. The formula giving $\overline{W}_i(a,t)$ becomes:$\sum_{\tau_k \in class(\tau_i), k \neq i}(1 + \lfloor min(t, a)/T_k \rfloor)C_k$, and the analysis can be simplified as follows:
$L_i(a) = \sum_{k=1, k\neq i}^{n}(1 + \lfloor a/T_k \rfloor)C_k + \lfloor a/T_i \rfloor C_i$ which leads to $r_i(a) = \sum_{k=1}^{n}(1 + \lfloor a/T_k \rfloor)C_k - a$ (see [7]).
We finally have:

Theorem 1 *A set $\tau = \{\tau_1, .., \tau_n\}$ of n sporadic tasks is feasible on a uniprocessor, using ECDF iff: $\forall i \in [1, n]$, $max_{0 \leq a < B_i(t_i^0)} r_i(a) \leq D_i$.*

5.1.3 Complexity analysis

This scheduling policy can be implemented very easily. Let us consider any processor j involved in n_j tasks and in N_j classes. It has one pending queue and N_j ECDF queues. When processor j receives a new task, it inserts it in its pending queue ordered by increasing release time. This costs $O(n_j log(n_j))$. The move from the pending queue to the ECDF queue of the task costs $O(1)$, because the task is always inserted at the tail of its ECDF queue. To select the next task to run, j compares the tasks head of the ECDF queues and selects the one with the smallest inherited deadline. It follows that the number of comparisons is equal to N_j. **The run time complexity is then in $O(n_j log(n_j))$, where n_j is the number of tasks involving processor j.**
We now consider the complexity of the feasibility conditions that is expressed by the length of the interval where $r_i(a)$ must be computed. From lemma 8, the worst case response time of a task τ_i is found in a busy period whose length is bounded by the length of the first synchronous busy period L. It can be shown that L always exists when $U \leq 1$ ([9]). Hence the feasibility conditions complexity depends on the duration of L. From lemma 9, L is the solution of $L = \sum_{k=1}^{n} \lceil L/T_k \rceil C_k$. As $\forall x \in \Re$, $\lceil x \rceil \leq 1 + x$. When $\exists (0 \leq \alpha < 1)$ such that $U \leq \alpha$ (which is the case we consider) then we have: $L = \sum_{k=1}^{n} \lceil L/T_k \rceil C_k \leq \sum_{k=1}^{n}(1 + L/T_k)C_k \leq \sum_{k=1}^{n} C_k + U.L.$

Hence, $L \leq \frac{\sum_{k=1}^{n} C_k}{1-\alpha}$. From lemma 9, L is computed by an iterative equation. Each step of the iterative formula takes $O(n)$ time, thus the whole computation takes $O(n(\sum_{k=1}^{n})/C_{min})$ time, with $C_{min} = min_{i=1\cdots n} C_i$. Hence the complexity of the feasibility conditions is pseudo-polynomial, when $\exists (0 \leq \alpha < 1)$ such that $U \leq \alpha$.

5.2 Distributed case

In this section, we extend the worst case response time analysis developed in the uniprocessor case to the distributed case. For any processor j, we have:
• C_i^j, the execution time of the copy of τ_i run by processor j.
• U^j the utilization factor of processor j.
• $\overline{W}_i^j(a,t)$, the time needed by processor j to execute the instances of tasks other than τ_i with an higher priority than τ_i.
• $L_i^j(a)$, the time where τ_i's instance requested at time $a \geq 0$, can start its execution on processor j.
• $r_i^j(a)$, the worst case response time on processor j, of τ_i's instance requested at time a.

Effect on the processors running a task's copy: Network delay and clock precision.
When distribution is considered, a task τ_i released at time a by its initiator, is received by a processor j that runs a copy of τ_i, at a variable time $a + network\_delay$ such that $network\_delay \leq Max$. And as clocks are ε-synchronized, this task has to wait until time $a + Max + \varepsilon$ before being inserted in the ECDF queue associated with $class(\tau_i)$. The worst-case analysis given in 5.1.2 must be adapted. The response time of any task τ_i released at time a by its initiator, must account for the network delay and the clock precision, $r_i^j(a) = Max + \varepsilon + max(C_i^j, L_i^j(a) + C_i^j - a)$.

Effect on the initiators: clock precision.
The clock precision on the initiators can be treated with the release jitter approach [18]. Indeed as the initiators timestamp the tasks with their release time, for two tasks released on two initiators p_1 and p_2 at the same global time, there can be a difference in values $\leq \varepsilon$. A task τ_i released at local time a by p_1 can be delayed by another task τ_k released by any $p_2 \neq p_1$, at global time t, (at time $t - \varepsilon$ at p_2) with $t \leq a + \varepsilon$ if:
• there is a processor involved both in τ_i and in τ_k execution.
• p_2's clock is ϵ-late compared to p_1's clock.

The clock precision has three impacts:

• First, the formula giving $\overline{W}_i^j(a,t)$ for processor j and task τ_i released at time a becomes: $\overline{W}_i(a,t) = \sum_{\tau_k \in hp_i(a)}(1 + \lfloor \frac{min(t, a+D_{class(\tau_i)}-D_{class(\tau_k))}+\varepsilon_k}{T_k} \rfloor)C_k$, where $\epsilon_k = \epsilon$ if processor j runs a copy of τ_k and the initiator releasing τ_k is not the one releasing τ_i, $\varepsilon = 0$ otherwise. The computation of $L_i^j(a)$ must use the new formula of $\overline{W}_i^j(a,t)$ and a is such that $0 \leq a < B_i^j(t_i^0)$.
• Second, $B_i^j(t_i^0)$ becomes:
$B_i^j(t_i^0) = \sum_{k \neq i} \lceil \frac{B_i^j(t_i^0)+\epsilon_k}{T_k} \rceil C_k^j + \lceil \frac{B_i^j(t_i^0)-t_i^0}{T_i} \rceil C_i^j$.
• Third, the transmission delay is taken into account in the formula giving $r_i^j(a)$:
$r_i^j(a) = Max + \varepsilon + max(C_i^j, L_i^j(a) + C_i^j - a)$. The worst case response time of τ_i, released at time a, is the maximum of the response times get on each processor j running a copy of τ_i. We have: $r_i(a) = max_j(r_i^j(a))$.

Due to the jitter effect, the existence of $B_i^j(t_i^0)$ can only be ensured when $U^j < 1$ [6]. This side effect is accounted for in the feasibility condition:

Theorem 2 *If for any processor j, $U^j < 1$, then a set $\tau = \tau_1, \ldots, \tau_n$ of n sporadic tasks is feasible in a distributed system using active redundancy and ECDF scheduling iff: $\forall i \in [1,n]$, for any processor k running a copy of τ_i, we have $r_i^k \leq D_i$.*

In the uniprocessor case, we have shown that the complexity of the feasibility condition is pseudo-polynomial when $U^j < 1$. For any task τ_i, the number of processors running a copy of τ_i is $f_c + 1$ where f_c is the replication degree of $class(\tau_i)$. Hence, the general feasibility condition is still pseudo-polynomial.

6 Conclusion

We have considered hard real-time constraints and consistency constraints in distributed systems, where fault-tolerance is achieved by active redundancy. We have first partitioned the set of tasks into classes. Intuitively a class is a set of tasks having conflicting accesses. Our solution can be used by applications where it is required to schedule tasks in the same class according to their release order, to achieve fairness among tasks initiators. For cost reasons, a processor runs tasks belonging to different classes. Our solution is based on a global total order per class, achieved by a combination of uniform real-time reliable multicast and ECDF scheduling. We have established feasibility conditions, under which any sporadic task meets its deadline. Each task is executed in $f_c + 1$ copies, where f_c is the maximum number of crashed processors tolerated by class C. We have shown that the complexity of these feasibility conditions is pseudo-polynomial when the processor utilization is strictly bounded by 1. On any processor j, the run time complexity is in $O(n_j log(n_j))$, where n_j is the number of tasks involving processor j.

References

[1] D. Agrawal, A. El Abbadi, *The tree quorum protocol: an efficient approach for managing replicated data*, Proc. 16th Int. Conf. on Very Large Databases, pp. 243-254, Brisbane, Australia, 1990.

[2] N. C. Audsley, *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*, Dept. Comp. Science, YCS 164, Univ. York, 1991.

[3] P.A. Bernstein, V. Hadzilacos, N. Goodman, *Concurrency control and recovery in distributed database systems*, Addison Wesley, 1987.

[4] S. K. Baruah, R. R. Howell, L. E. Rosier, *Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor*, Real-Time Systems, 2,pp 301-324, 1990.

[5] D. Dolev, D. Malki, *The Transis approach to high availability cluster communication*, Communications of the ACM, 39(4), pp. 64-69, April 1996.

[6] L. George, *Ordonnancement en ligne temps réel critique dans les systèmes distribués*, PhD Thesis, Univ. Versailles Saint-Quentin, Jan. 1998.

[7] L. George, P. Minet, *A FIFO worst case analysis for a hard real-time distributed problem with consistency constraints*, ICDCS'97, Baltimore, Maryland, May 1997.

[8] L. George, P. Minet, *A uniform reliable multicast protocol with guaranteed response times*, LCTES'98, Montreal, Canada, June 1998.

[9] L. George, N. Rivierre, M. Spuri, *Preemptive and non-preemptive real-time uniprocessor scheduling*, INRIA Rocquencourt, France, RR 2966, Sept. 1996.

[10] R. Guerraoui, A. Schiper, *Software-based replication for fault-tolerance*, Computer, pp68-74, April 1997.

[11] M. Herlihy, *A quorum consensus replication method for abstract data types* ACM trans. on Computer Systems, 4(1), pp32-53, February 1986.

[12] V. Hadzilacos, S. Toueg, *A modular approach to fault-tolerant broadcasts and related problems*, TR94-1425, Cornell University, Ithaca, NY14853, May 1994.

[13] C. L Liu, J. W. Layland, *Scheduling algorithms for multiprogramming in a hard real time environment*, Journal of the ACM, 20(1), Jan. 1973.

[14] A. K. Mok, *Fundamental design problems for the hard real-time environments*, MIT Ph.D. Dissertation, May 1983.

[15] E. Pacitti, E. Simon, R. de Melo, *Improving data freshness in lazy master schemes*, ICDCS'98, Amsterdam, May 1998.

[16] F. Schmuck, F. Cristian, *Continuous clock amortization need not affect the precision of a clock synchronization algorithm*, RJ7290, IBM Almaden, Jan. 1990.

[17] M. Spuri, *Analysis of deadline scheduled real-time systems*, RR 2772, INRIA, France, Jan. 1996.

[18] K. Tindell, A. Burns, A.J. Wellings, *Analysis of hard real-time communication*, Real-Time Systems, 9, pp147-171, 1995.

[19] R. van Renesse, K. Birman, S. Maffeis, *Horus:a flexible group communication system*, Communications of the ACM, 39(4), pp. 76-83, April 1996.