Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies*

Pascal Chevochot — INRIA / IRISA — pchevoch@irisa.fr

Isabelle Puaut — INSA / IRISA — puaut@irisa.fr

IRISA, Campus de Beaulieu — 35042 Rennes — France

Proc. of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99), Hong-Kong, China, december 1999.

Abstract

*Replication is a well-know fault-tolerance technique, and several replication strategies exist (*e.g. active, passive, *and* semi-active *replication). To be used in hard real-time systems, the presence of replication must be dealt with in scheduling algorithms, and more particularly in the feasibility tests in charge of testing whether deadlines will be met or not. So far, existing solutions to integrate replicated tasks in scheduling algorithms were specific to a given replication strategy or to its implementation on a given architecture. This paper is devoted to the description of a framework for taking into account the replicated tasks in scheduling algorithms that is largely independent of the replication technique. We show on an example that the same scheduling algorithm can be used whatever replication strategy is selected, even if several replication strategies are simultaneously used.*

1 Introduction

Hard real-time systems are systems for which tasks produce incorrect results when they miss their deadlines. In such systems, a *scheduling algorithm* is in charge of computing an order in which tasks will be executed. In hard real-time systems, a scheduling algorithm includes a *feasibility test* which decides whether all deadlines will be met or not. A scheduling algorithm is called *off-line* (resp. *on-line*) when its feasibility test is executed on the entire task set before tasks execution (resp. at run-time at task arrival or termination). A scheduling algorithm is called *static* (resp. *dynamic*) if the scheduling decisions are based on fixed parameters assigned to tasks before their activation (resp. are based on dynamic parameters that may change at run-time).

Most hard real-time systems must be highly reliable. The use of fault-tolerance techniques is one way to reach a high reliability. Three steps are required to make a system fault tolerant: *error detection* to detect erroneous system states; *error treatment* to eliminate errors before they lead to an incorrect behavior of the system; and *fault-treatment* to isolate the faulty system components and reconfigure the system to tolerate other faults. The central step of fault-tolerant techniques is error treatment. Most error treatment strategies are based on some kind of redundancy (*replication* in the fault-tolerance terminology). The mostly used task replication techniques [12] are *active*, *semi-active* and *passive* replication. When using active replication, the same computation is executed in parallel on failure-independent sites; a consensus protocol detects erroneous replicas and masks them. When using passive replication a computation is executed on a single site, called *primary* site, whereas other sites (*backup* sites) memorize the computation state *(checkpoint)*. Semi-active replication is similar to active replication except that decisions common to all replicas are taken by one site, whereas in active replication they are taken by a distributed consensus protocol.

The presence of hard timing constraints complicates the design of fault-tolerant real-time systems. Both the task replicas and the fault-tolerance building blocks used in the replication strategies (*e.g.* consensus protocol used in active replication) must be integrated into the scheduling algorithms.

Related work

Several studies have dealt with the integration of fault-tolerance mechanisms into scheduling algorithms. In some

*This work is partially supported by the French Department of Defense (DGA/DSP), #98.34.375.00.470.75.65. and by Dassault-Aviation, #98.C.304.00.31303.01.2

of them [8, 5, 13] all tasks have identical fault-tolerance attributes: they are replicated on the same sites and use the same replication strategy (active replication in the above-mentioned references). This simplifies the integration of fault-tolerance mechanisms into scheduling algorithms: if a static scheduling algorithm is used, the same scheduling plan can be used for all sites [8]; if a dynamic scheduling algorithm is used, scheduling decisions can be taken independently by the sites if they are not source of non determinism [14] (see § 3.3 for a definition of determinism).

When tasks may have different fault-tolerance attributes (replication at the task level or lower), the same scheduling decisions cannot be used for all sites anymore. In [7], scheduling of periodic fault-tolerant computations with precedence constraints is considered, but this work considers only active replication. Other works [10, 6, 1] deal with the integration of fault-tolerant tasks in scheduling algorithms for shared-memory multiprocessors. Such studies take advantage of the underlying architecture (i.e. low latency communications) to neglect the cost of the fault-tolerance building blocks like state saving or consensus. The study described in [9] treats task replication and task scheduling in an independent way (tasks are replicated before the feasibility test is applied to the fault-tolerant tasks), but only active replication is considered and the issue on non deterministic replicated computations is not tackled.

Contribution of this paper

The above-mentioned solutions to integrate replicated tasks in scheduling algorithms rely on the fault-tolerance mechanism in use: type of replication strategy [7, 9], replication level and replica location [8, 5, 13] or implementation of the replication strategy on a given architecture [10, 6, 1]. The contribution of this paper is to define a framework for integrating replicated tasks in scheduling algorithms that is largely independent of the fault-tolerance strategy. Integration of replicated tasks in a scheduling algorithm depends neither on the type of replication strategy (active, passive or semi-active), nor on its parameters (replication level, replication granularity and location of replicas).

Our approach relies on the use of a task replication tool we have developed, named hereafter Hydra. It makes tasks fault-tolerant through the replication of parts of their code before their execution (graph rewriting). Hydra provides an extensible set of replication strategies. Currently, three error treatment strategies are provided: active, passive and semi-active replication. To use Hydra, applications must conform to a generic task model that statically identifies communications and synchronizations. Scheduling fault-tolerant tasks in this context only consists in using a scheduling algorithm that meets a set of properties that do not depend on the replication strategy: support for the generic task model, and consideration of ordering constraints required to make the execution of replicated computations deterministic.

Paper organization

The remainder of the paper is organized as follows. Section 2 introduces the system model on which our work relies. An overview of the Hydra replication tool is then given in Section 3. Two of the error treatment strategies provided by Hydra (active and passive replication) are detailed, and the constraints that must be met by the scheduling algorithm to deal with the issue of determinism of replicated computations are described. Section 4 is then devoted to the issue of scheduling fault-tolerant tasks: the properties to be met by a scheduling algorithm to deal with fault-tolerant tasks are described. An example of scheduling algorithm that meets these properties is given in Section 5, and its use to schedule a fault-tolerant application is detailed. This example shows that the integration of fault-tolerant tasks into a scheduling algorithm does not depend on the error treatment strategy used (passive and active replication in the example). Lastly, we conclude in Section 6.

2 System model

Our work relies on a system model that describes the underlying architecture (§ 2.1), its fault model (§ 2.2) and a task model defining the structure and properties of tasks (§ 2.3).

2.1 Architectural model

Computations are executed on a set of sites connected by a network. Every site has access to sensors and actuators used respectively for observing and acting on the external environment. A site is made of a processor, a memory and a clock. A *resource* abstracts any hardware or software component that can be shared between computations (*i.e.* memory segment, semaphore). A resource is local to a site, and has an associated persistent memory area (*state* of the resource).

2.2 Fault model

We consider permanent and transient faults that are the results of physical phenomena (also called *physical faults)*. Sites are assumed to be *fail-silent*: every site either provides correct results in the time and value domains or remains silent forever (*i.e.* is stopped). We assume that the physical network never partitions and that a bounded number of successive omissions can occur. Actuators are assumed to be reliable. Lastly, sensors are subject to value and timing errors. The actual number of errors that are supported depends on the error treatment strategy in use (see § 3).

2.3 Task model

The task model defines the internal structure of tasks, and attributes related to their execution. The key motivations for the definition of this task model were to be able to statically identify temporal and synchronization attributes so that tasks can be integrated into off-line scheduling algorithms, and to be compatible with most task models for hard real-time systems found in the literature.

Task structure. Every task is described by a directed acyclic graph whose nodes model computations and edges model precedence constraints and data transfers between them.

A node (also called *EU* for *Elementary Unit*) models a computation local to a site. It is a sequence of code having a known worst-case execution time (obtained for example through static program analysis). An *EU* uses a set of input variables and computes a set of output variables. An *EU* has no memory area which persists after each of its executions.

An edge models a *precedence constraint* between nodes. A precedence constraint from eu_i to eu_j (noted $eu_i \rightarrow eu_j$) states that eu_j can start executing only after eu_i has finished. An output variable *out* of an *EU* eu_i can be copied into the input variable in of an *EU* which is the destination of a precedence constraint from eu_i. We note this copy $in := out$.

A graph may also include *conditional constructs*. A conditional construct serves at dynamically requesting the execution of p subgraphs among n possible ($p \leq n$). In the most pessimistic case, which is considered in the feasibility test, the n subgraphs are executed.

Task attributes. A set of synchronization, timing, distribution and fault-tolerance attributes can be specified for each task.

Synchronization attributes allow synchronizations between *EUs* through shared resources that can be requested by *EUs* in *exclusive* or *shared* mode (the *exclusive* mode must be used to modify a resource). An attribute associated with each precedence constraint $(eu_i \rightarrow eu_j)$ indicates resources which must not be released when eu_i terminates. These resources are said to be *inherited* from eu_i by eu_j.

Timing attributes express temporal properties at the task and *EU* levels (e.g. expected task arrival law, deadline, earliest and latest start time).

Distribution attributes mainly define for each *EU* the site where to execute it (the execution site of an *EU* is statically known).

Fault-tolerance attributes specify for each task or task portion (*i.e.* graph or subgraph) which replication strategy is to be applied, the requested replication level, and additional information specific to the replication strategies (*e.g.* consensus function for active replication).

3 Task replication

Tasks are made fault-tolerant through the replication of parts of their code (graph rewriting). This is achieved offline thanks to a tool we have developed, which is named Hydra. Hydra takes the following fault-tolerance attributes given by the application designer as input parameters: a set of *patterns*, and for each of them the *transformation schemes* to be used, and the corresponding *replication level* (requested number of copies). The pattern is the granularity at which the fault-tolerance mechanisms are applied. A pattern is a subgraph made of *EUs* assigned to the same site. For every pattern, Hydra applies the transformation schemes that have been requested by the application designer. A transformation scheme is an algorithm that replaces a pattern gr by n copies of gr, with n being the replication level, and connects them with *EUs* implementing basic fault-tolerance building blocks (*e.g.* consensus, state backup). Hydra provides an extensible set of replication strategies for error treatment (currently *active*, *passive* and *semi-active* replication) and for error detection (*temporal* replication). We present hereafter the transformation schemes for active replication (§ 3.1) and passive replication (§ 3.2) named hereafter T_{active} and $T_{passive}$. Then, we present the constraints that must be respected by a scheduling algorithm in order to enforce determinism of the replicated executions (§ 3.3). Further details concerning Hydra can be found in [3].

3.1 Transformation scheme for active replication (T_{active})

Figure 1 depicts T_{active} applied to a pattern gr whose output variables are named Out_i, $1 \leq i \leq p$, with a replication level n . An attribute specifies the n sites onto which the copies of gr must be placed. When gr does not use sensors, active replication masks $n - 1$ errors (stopped sites) resulting from transient or permanent faults.

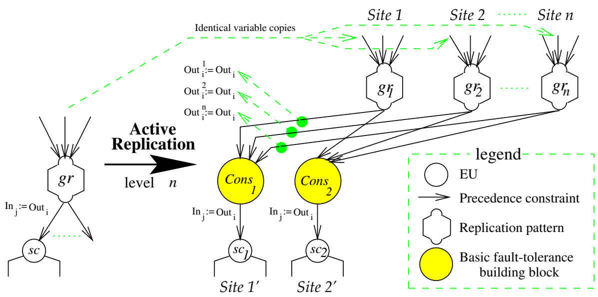


Figure 1. Transformation scheme for active replication of level n

T_{active} transforms pattern gr into n copies of gr called gr_k, $1 \leq k \leq n$, which are run on the n sites in parallel. All resources used by gr are also replicated on all these sites[1]. T_{active} connects the n copies of gr to m fault-tolerance building blocks called $Cons_k$, $1 \leq k \leq m$. The number m is equal to the number of different sites where a successor sc of gr is placed (when sc is also replicated, m is the number of sites onto which one copy of sc is placed). Each $Cons_k$ is in charge of determining a consensus value for each output variable Out_i of gr, *i.e.* must compute the same correct value for each variable Out_i from the values of Out_i of $gr_k s$. By default, $Cons_k$ implements the consensus function *any-value*, that returns any Out_i coming from a non-stopped site.

If the *EUs* of a pattern use sensors, the default function *any-value* may not be suitable anymore, since values returned by sensors copies can vary according to sites and times when they were read. The application designer can thus use an alternative consensus function. A library of commonly used consensus functions is provided: *f-midpoint function* and *f-mean function* [4], which both allow to mask f faulty sensors, with $f \leq \lfloor \frac{n-1}{2} \rfloor$.

3.2 Transformation scheme for passive replication ($T_{passive}$)

Passive replication is especially interesting in dynamic environments, because it is then possible in case of fault-free executions to reclaim the resources used by backup copies of computations. $T_{passive}$ replicates a pattern gr, whose output variables are named Out_i, $1 \leq i \leq p$, on n sites. It allows to support $n-1$ site stops resulting from transient or permanent faults, and assumes reliable sensors. For the sake of simplicity, Figure 2 depicts $T_{passive}$ for a replication level of 2. An attribute specifies the two sites on which the copies of gr must be placed. Let R_e be the resources used by gr in exclusive mode, and R_s those used in shared mode. $T_{passive}$ requires that the states of resources of R_e can be saved and restored.

$T_{passive}$ replaces gr by two copies of gr located on two different sites. One of the two sites (site 1 on the figure) is the primary site, while the other one (site 2) is the backup site unless site 1 is stopped. The resources used by gr are replicated on site 1 and site 2. They are noted respectively R_e^1 and R_s^1 on site 1, and R_e^2 and R_s^2 on site 2. On site 1, the fault-tolerance building block *Bck* is in charge of transferring the states of resources belonging to R_e^1 from site 1 to site 2 (*i.e.* copying their state in its output variable *st*). The fault-tolerance building block *Mon* is in charge of deciding whether site 2 is backup or primary, depending on the

[1] A resource that only consists of a memory area is replicated without user intervention, while for a more specific resource (*e.g.* peripheral device, server) the application designer is in charge of resource replication and indicates the identity and location of the resource replicas.

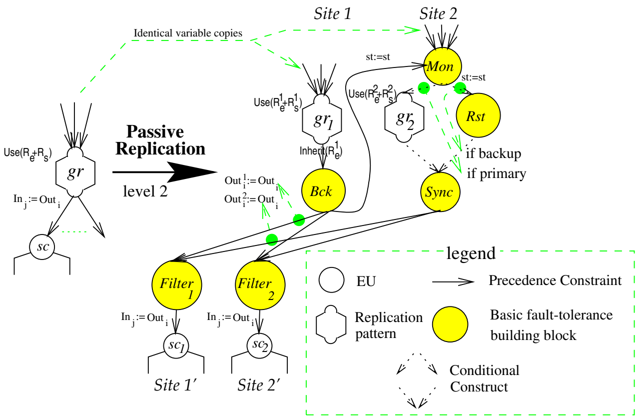


Figure 2. Transformation scheme for passive replication of level 2

validation status of the precedence constraint *Bck* $\rightarrow$ *Mon*. If site 2 is backup (*i.e.* if site 1 is not stopped), *EU Mon* selects, thanks to a conditional construct, the execution of fault-tolerance building block *Rst*, which copies the states of resources of R_e^1 into the states of resources of R_e^2. If site 2 is primary (*i.e.* if site 1 has stopped) the execution of gr is recovered on this site, *i.e.* *Mon* selects execution of gr_2. The m fault-tolerance building blocks *Filter*$_i$ are in charge of collecting the output values of the primary site (site 1 or 2 depending on whether site 1 is stopped or not). These m *EUs* are located on the m sites where a successor sc of gr (or one successor of a copy of sc if sc is replicated) is placed.

Note that $T_{passive}$ transforms the *structure* of tasks and not their *attributes* (and in particular their arrival law). Thus, the checkpointing activity has the same arrival law as the task itself (in the case of a periodic task, the checkpointing period is equal to the task period). Also note that when using $T_{passive}$, the system is restarted from a consistent state [2] after a site stop, since a checkpoint is saved at each communication (see [3] for details).

3.3 Enforcing determinism of replicated computations

Let *Rep*(S) be the set of the transformed patterns of the whole system S by Hydra. System S is said to be *deterministic* [11] if, given the same initial state for all copies of replicated resources, after each correct execution of $g \in$ *Rep*(S), every copy of an output variable Out of g has the same value and the same state changes are made on every copy of a replicated resource.

Sources of non determinism. Two sources of non determinism are possible in our context. The first one, which appears only when using active replication, results from the presence of replicated sensors. This source of non determinism is solved by the provision of sensor-specific consensus functions (see § 3.1).

The second source of non-determinism is the execution in a different order on different sites of replicated computations which modify some common resources. Let us illustrate this source of non-determinism on passive replication. Let R be a resource and eu_a and eu_b be two (mono-*EU*) patterns that modify R. Let us further assume that a single instance of eu_a and eu_b exist at any time, and that eu_a and eu_b are both replicated using $T_{passive}$ on the same sites (1 and 2). Let Bck_a^1, Bck_b^1, Rst_a^2 and Rst_b^2 be the fault tolerance building blocks in charge of transferring the state of R from site 1 to site 2. Bck_a^1 and Bck_b^1 must be executed in the same order as Rst_a^2 and Rst_b^2 to observe the same state changes on the replicas of resource R. The ordering constraints required to enforce determinism must be compatible with the execution order computed by the scheduling algorithm. Thus, we propose to solve this second source of non-determinism by imposing a set of ordering constraints between copies of replicated *EUs* to the scheduling algorithm (see below).

Ordering constraints to enforce determinism. An ordering constraint is a relation *IO* (*Identical Order*) between two couples of *EUs*, each couple being composed of two instances of *EUs* that run on the same site. (eu_i, eu_j) *IO* (eu_k, eu_l) if:

$$(eu_i \mapsto eu_j \textit{ and } eu_k \mapsto eu_l) \textit{ or}$$
$$(eu_j \mapsto eu_i \textit{ and } eu_l \mapsto eu_k),$$

where $eu_i \mapsto eu_j$ means that eu_j can execute only after eu_i has finished. For every replication strategy Rep and every pattern gr, we define $E(Rep, gr)$, the set of groups of *EUs* for which scheduling decisions can lead to non deterministic executions. Let us give the contents of $E(T_{active}, gr)$ and $E(T_{passive}, gr)$, where *gr* is a subgraph composed of p *EUs* $eu_1, ..., eu_p$. Let $eu_i^1, ...eu_i^n$ be the replicas of *EU* eu_i when using active replication of level n, and let $Bck^1, ...Bck^{n-1}$ and $Rst^1, ...Rst^{n-1}$ be the fault tolerance building blocks generated by passive replication of level n.

$$E(\textit{Active}, gr) = ((eu_1^1, ..., eu_1^n), (eu_2^1, ..., eu_2^n), ..., (eu_p^1, ..., eu_p^n))$$

$$E(\textit{Passive}, gr) = ((Bck^1, Rst^1, ..., Rst^{n-1}), (Bck^2, Rst^2, ..., Rst^{n-1}), ..., (Bck^{n-1}, Rst^{n-1}))$$

The set of ordering constraints for the whole system is computed from the sets $E(Rep, gr)$ for all possible transformation schemes Rep. Let E be the union of all $E(Rep, gr)$ for all patterns *gr* in the system and all transformation schemes. Let $E/(s_i, s_j, R)$ be the set E restricted to the *EUs* that use a replica of resource R and execute on site s_i or on site s_j. For every possible $E/(s_i, s_j, R)$ and for every possible set $((eu_i, eu_j), (eu_k, eu_l)) \subset E/(s_i, s_j, R)$ such that a replica of R is requested in exclusive mode by one of these *EUs*, an ordering constraint (eu_i, eu_j) *IO* (eu_k, eu_l) is added.

In the above example, only two tasks, each composed of a single *EU*, execute. Both tasks use passive replication of level 2. In this example, we have $E = E(\textit{Passive}, eu_a) \cup E(\textit{Passive}, eu_b) = \{\{Bck_a^1, Rst_a^2\}, \{Bck_b^1, Rst_b^2\}\}$. Since there are only two sites (site 1 and site 2) and since a single resource R is modified, a single ordering constraint (Bck_a^1, Rst_a^2) *IO* (Bck_b^1, Rst_b^2) is created.

4 Scheduling Fault-Tolerant Tasks

To use Hydra, applications must conform to the task model described in paragraph 2.3. Hydra takes as parameters: the granularity of replication (whole task or task portions), the replication strategy and the replication parameters (the replication level and the sites where to replicate computations). Hydra then transforms the application graphs through the addition of: *(i)* replicas for the patterns that must be made fault-tolerant; *(ii)* fault-tolerance building blocks (*e.g.* state transfer, consensus, filtering); *(iii)* precedence constraints and resource inheritance between these components. Whatever replication strategy is used, fault-tolerant tasks are structured according to the same task model. As a consequence, to take fault-tolerant tasks into account, a scheduling algorithm must:

R1: support tasks that are *distributed* and synchronize through *shared resources* and *precedence constraints* (*i.e.* tasks that conform to the task model of § 2.3);

R2: respect the *ordering constraints* required to enforce determinism of replicated executions.

Once requirements $R1$ and $R2$ are met by a given scheduling algorithm, the way fault-tolerant tasks are integrated into scheduling algorithms is largely independent of the replication strategy; it depends neither on the type of replication strategy (active, passive of semi-active replication) nor on its parameters (replication level, replication granularity and location of replicas). The only information that must be provided by the designer of a replication strategy Rep are the sets $E(Rep, gr)$ for all patterns gr. These sets will be used to compute the ordering constraints required to enforce determinism of replicated computations.

Tasks replicated using any replication strategy that can be applied off-line can be scheduled in a unified manner. This also applies when several replication strategies are used within the same system (see Section 5 for an example). We have shown that active replication and passive replication can be applied off-line, and that the ordering constraints for these replication strategies can be computed.

Note that meeting requirements $R1$ and $R2$ does not impose any constraint on the type of scheduling algorithm to be used (static or dynamic, off-line or on-line). However, meeting requirement $R2$ in dynamic scheduling algorithms may be more expensive than in static scheduling algorithms because a consensus may be required to have identical execution orders of the replicated computations.

It now remains to demonstrate that it is possible to design scheduling algorithms that meet requirements $R1$ and $R2$ and still find feasible schedules in reasonable time. This is the purpose of the next section.

5 A Scheduling Algorithm for Fault-tolerant Tasks

An example of scheduling algorithm capable of scheduling the fault-tolerant tasks generated by Hydra, *i.e.* that meets requirements $R1$ and $R2$, is described (§ 5.2) and illustrated via a simple application (§ 5.3). It is an extension of the algorithm described by Xu and Parnas in [15] (§ 5.1).

5.1 Xu and Parnas algorithm

The Xu and Parnas static off-line scheduling algorithm [15] (hereafter *XuParnas*) operates on a set of non distributed tasks with release times, deadlines, precedence and exclusion relations. It generates an optimal preemptive schedule. A task is made of a set of *EUs* (*segments* in the referenced paper). Each *EU* eu_i has a release time, a deadline and a computation time. A set of exclude (*EX*) and precede (*PC*) relations are defined on ordered pairs of *EUs*: eu_i *PC* eu_j means that *EU* eu_j cannot start execution before *EU* eu_i has completed its computation; eu_i *EX* eu_j means that if *EU* eu_i started execution before *EU* eu_j, then *EU* eu_j can only start execution after *EU* eu_i has completed its computation. The structure of tasks in the scheduling algorithm *XuParnas* is identical to the structure of tasks in our task model. A slight difference exists concerning the definition of the exclusion relations between two *EUs* eu_1 and eu_2: the use of a resource R in exclusive mode by eu_1 and eu_2 in our task model is equivalent to eu_1 *EX* eu_2 *and* eu_2 *EX* eu_1 in the task model of *XuParnas*.

The problem of finding a schedule from a set of tasks with release times, deadlines, precedence and exclusion relations is NP-hard [15]. The scheduling algorithm *XuParnas* thus uses a branch-and-bound technique. It has a search tree where at its root node, an earliest-deadline-first (EDF) strategy is used to generate a valid initial solution (*i.e.* a schedule that respects release time, precedence and exclusion relations but in which some *EUs* can miss their deadlines). At each node of the search tree, the latest *EU* eu_l in the schedule S computed at that node is identified. Two sets of successor nodes are created: one containing the schedules in which eu_l is scheduled before an *EU* of S (addition of a *PC* relation), and another one containing the schedules in which eu_l *preempts* an *EU* of S (addition of a preempt relation *PM*, where eu_i *PM* eu_j means that *EU* eu_j cannot start its execution while *EU* eu_i is ready to execute). At each node of the search tree, schedules are obtained through the use of an EDF strategy. New nodes in the search tree are created until a *feasible* schedule (*i.e.* a schedule that respects both synchronization and temporal constraints) is found, or, until all feasible schedules are found.

5.2 Extended algorithm

Three extensions have been made to the scheduling algorithm *XuParnas* to meet requirements $R1$ and $R2$. Let us call *XuParnasExt* the extended scheduling algorithm.

Exclusions between groups of EUs. The graphs generated by Hydra contain resources that are requested by a *group* of *EUs* of the same task (*resource inheritance* in § 2.3). *XuParnas* cannot directly deal with such resource sharing constraints, because the *EX* relation is defined on an ordered pair of *EUs* and not on groups of *EUs*. Relation *EX* has thus been extended to deal with a group of *EUs* as first argument: I EX_e eu_j means that if an *EU* eu_i of set I started execution before *EU* eu_j, then *EU* eu_j can only start execution after all *EUs* of I have completed their computation.

The introduction of relation EX_e changes the computation of schedules that are created at every node of the search tree. More precisely, the function *Eligible(*eu_i*,t)* in charge of deciding whether an *EU* eu_i is eligible at time t is changed to cope with exclusions between groups of *EUs*.

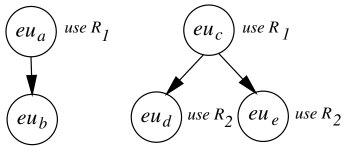
Distributed execution. The precede relation *PC* of *XuParnas* has thus been extended in *XuParnasExt* to deal with precedence relations between *EUs* running on different sites. The new precede relation PC_e is identical to *PC* when its arguments eu_i and eu_j are assigned to the same site; otherwise, PC_e specifies that the start time of *EU* eu_j must be at least $end(eu_i) + p$, where $end(eu_i)$ is the termination time of *EU* eu_i and p is an upper bound on the delay to validate a precedence constraint.

To take the relation PC_e into account, n schedules are created instead of a single schedule at every node of the search tree, where n is the number of sites (the latest *EU* eu_l of every schedule is identified at every node). In addition, function *Eligible(i,t)* is changed to deal with constant p.

Ordering constraints. To deal with the ordering constraints, a new relation *IO*, with the syntax and semantics given in § 3.3 has been added. It is taken into account during the computation of the schedule of every node in the search tree, by changing function *Eligible(eu_i,t)*. In addition, when a relation eu_i *PC* eu_j is added to create a new child node in the search tree, if a relation (eu_i, eu_j) *IO* (eu_k, eu_l) exists, then a new relation eu_k *PC* eu_l is added.

5.3 Example

Figure 3 describes a simple system made of two tasks, in which all resources are requested in exclusive mode. The figure gives the tasks structures, the temporal attributes of their *EUs* (release time r, worst-case computation time c and deadline d) and their scheduling relations according to the terminology of *XuParnas* (*EX, PC* and *PM*).



eu	r	c	d
eu_a	0	10	40
eu_b	38	9	66
eu_c	0	10	30
eu_d	45	9	69
eu_e	53	9	63

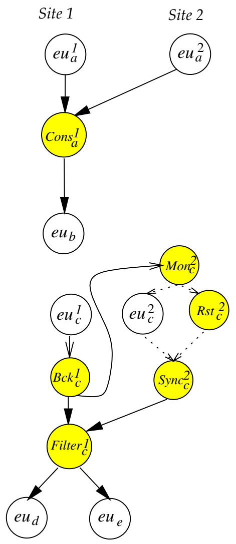
eu_a EX eu_c
eu_c EX eu_a
eu_d EX eu_e
eu_e EX eu_d
eu_a PC eu_b
eu_c PC eu_d
eu_c PC eu_e

Figure 3. Initial system

The system of Figure 3 is then made fault-tolerant through the use of Hydra: active replication is used to replicate *EU* eu_a on sites 1 and 2; passive replication is used to replicate *EU* eu_c on the same sites 1 and 2. Figure 4 describes the new fault-tolerant system (tasks structures, temporal attributes of their *EUs* and scheduling relations according to the terminology of *XuParnasExt*, *i.e.* *EX*$_e$, *PC*$_e$, *PM*$_e$ and *IO*). In figures 3 and 4, values for temporal attribute c are example values chosen for the purpose of illustration and not the actual values.

Figure 5 is devoted to the application of the scheduling algorithm *XuParnasExt* to the fault-tolerant system.

The top schedule is generated by *XuParnas* on the initial (non fault-tolerant) system. In this example, the schedule built at the root of the search tree is a feasible schedule. The next solution, labeled *Initial solution* in Figure 5, is the initial solution of *XuParnasExt*. It is made of two schedules, corresponding to sites 1 and 2. In this solution, the deadline of *EU* eu_e is not met, and the *EU* with the maximum lateness is *EU* eu_e. Two children of the root node in the search tree are then created, and the two corresponding sets of schedules are created: S_1 (obtained through the addition of a relation eu_e *PC*$_e$ eu_d) and S_2 (obtained through the addition of a relation eu_e *PR*$_e$ eu_b). In the first schedule of S_1, the deadline of *EU* eu_d is not met. In the two schedules



eu	r	c	d
eu_a^1	0	10	40
eu_a^2	0	10	40
$Cons_a^1$	0	1	66
eu_b	38	9	66
eu_c^1	0	10	30
Bck_c^1	0	2	69
Mon_c^2	0	1	69
Rst_c^2	0	2	69
eu_c^2	0	10	30
$Sync_c^2$	0	1	69
$Filter_c^1$	0	1	69
eu_d	45	9	69
eu_e	53	9	63

$\{eu_a^1\}$ EX$_e$ eu_c^1
$\{eu_a^1\}$ EX$_e$ Bck_c^1
$\{eu_c^1, Bck_c^1\}$ EX$_e$ eu_a^1
$\{eu_a^2\}$ EX$_e$ eu_c^2
$\{eu_c^2\}$ EX$_e$ eu_a^2
$\{eu_a^2\}$ EX$_e$ Rst_c^2
$\{Rst_c^2\}$ EX$_e$ eu_a^2
$\{eu_c^2\}$ EX$_e$ Rst_c^2
$\{Rst_c^2\}$ EX$_e$ eu_c^2
$\{eu_d\}$ EX$_e$ eu_e
$\{eu_e\}$ EX$_e$ eu_d
eu_a^1 PC$_e$ $Cons_a^1$
eu_a^2 PC$_e$ $Cons_a^1$
$Cons_a^1$ PC$_e$ eu_b
eu_c^1 PC$_e$ Bck_c^1
Bck_c^1 PC$_e$ Mon_c^2
Bck_c^1 PC$_e$ $Filter_c^1$
Mon_c^2 PC$_e$ eu_c^2
Mon_c^2 PC$_e$ Rst_c^2
eu_c^2 PC$_e$ $Sync_c^2$
Rst_c^2 PC$_e$ $Sync_c^2$
$Sync_c^2$ PC$_e$ $Filter_c^1$
$Filter_c^1$ PC$_e$ eu_d
$Filter_c^1$ PC$_e$ eu_e
(eu_a^1, eu_a^2) IO (Bck_c^1, Rst_c^2)

Figure 4. Fault-tolerant system

of S_2, all deadlines are met, and the search for a feasible solution is stopped.

This example illustrates the low dependence between task replication and task scheduling. The only information given by the replication tool to the scheduling algorithm is the set of ordering constraints (a single *IO* relation in the example).

6 Conclusion

We have described in this paper a framework for integrating fault-tolerant tasks into scheduling algorithms of hard real-time distributed systems. The contribution of our work is that the way fault-tolerant tasks are integrated into scheduling algorithms depends neither on the type of replication strategy (active, passive or semi-active replication) nor on its parameters (replication level, replication granularity and location of replicas). We have shown on a simple application that different replication strategies can coexist and can be integrated in a scheduling algorithm in a unified manner.

Work in progress consists in examining if most replica-

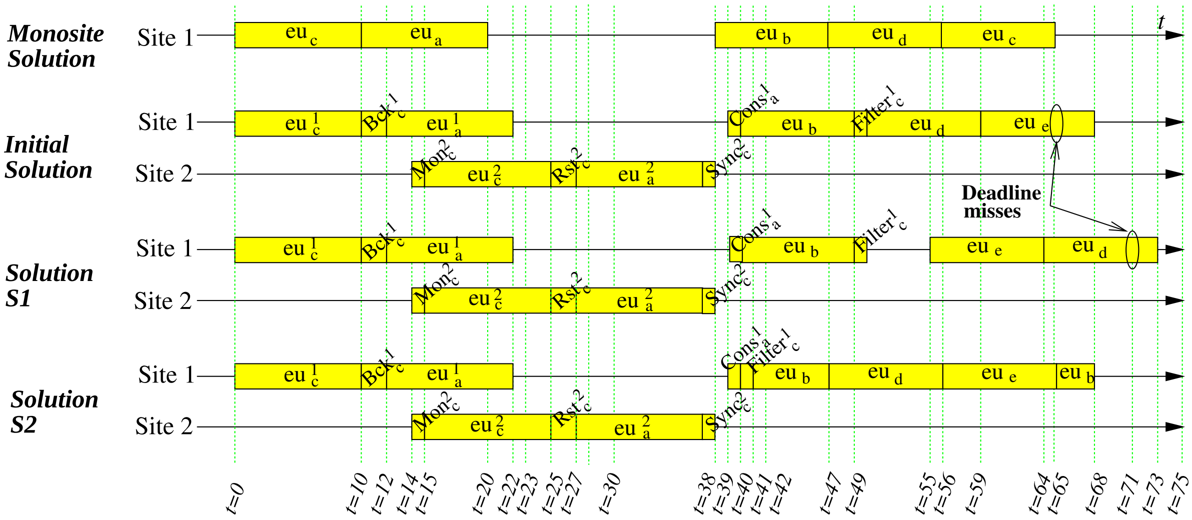


Figure 5. Scheduling of the fault-tolerant system using XuParnasExt

tion strategies can be applied off-line. We plan to enlarge the set of replication strategies (*e.g.* semi-passive replication) and to provide variants of the base replication strategies (*e.g.* to provide passive replication where the checkpointing period is independent of the task period). Future work also concerns scheduling issues, in particular to find an optimal algorithm capable of scheduling fault-tolerant tasks. Lastly, our approach is currently being validated through the development of an application from the field of aeronautics, provided by our industrial partner Dassault-Aviation.

Acknowledgments

This work has been carried out in the Solidor research group at IRISA, France. We would like to thank Michel Banâtre, the group head, for his support throughout this work. We are also grateful to D. Decotigny, F. Parain and the anonymous reviewers for useful comments on preliminary drafts of this paper. Further details on the context into which this work has been carried out can be found in the Web page *http://www.irisa.fr/solidor/work/hades.html*.

References

[1] A. Bertossi, A. Fusiello, and L. Mancini. Fault-tolerant deadline-monotonic algorithm for scheduling hard real-time tasks. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[3] P. Chevochot and I. Puaut. An approach for fault-tolerance in hard real-time distributed systems. Technical Report 1257, IRISA, July 1999.

[4] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl. Reaching approximate agreement in the presence of faults. In *Proc. of the Third Symp. on Reliability in Dist. Software and Database Systems*, pages 145–154, Clearwater Beach, USA, Oct. 1983.

[5] M. M. Elvany and E. Hugue. Guaranteed task deadlines for fault-tolerant workloads with conditional branches. *Real-Time Systems*, 3(3):275–305, 1991.

[6] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerant scheduling on a hard real-time multiprocessor system. *IEEE Trans. on Parallel and Dist. Systems*, 8(3):272–284, Mar. 1997.

[7] C.-J. Hou and K. Shin. Replication and allocation of task modules in distributed real-time systems. In *Proc. the 24th International Symposium on Fault-Tolerant Computing Systems*, pages 26–35, Austin, Texas, USA, June 1994.

[8] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Scwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, 9:25–40, Feb. 1989.

[9] D. Mossé. *Design, Development, and Deployment of Fault-Tolerant Applications for Distributed Real-Time Systems*. PhD thesis, University of Maryland, USA, 1993.

[10] Y. Oh and S. Son. Scheduling hard real-time tasks with tolerance of multiple processor failures. Technical Report CS-93-28, University of Virginia, May 1993.

[11] S. Poledna. *Fault-Tolerant Real-Time Systems – The Problem of Replica Determinism*. Kluwer Academic Publishers, 1996.

[12] D. Powell, editor. *Delta-4 : A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[13] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, and A. Wellings. GUARDS: A generic upgradable architecture for real-time

dependable systems. *IEEE Trans. on Parallel and Dist. Systems*, 10(6):204–226, June 1999.

[14] A. Wellings, L. Beus-Dukic, and D. Powell. Real-time scheduling in a generic fault-tolerant architecture. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 390–398, Madrid, Spain, Dec. 1998.

[15] J. Xu and D. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. on Software Engineering*, 16(3):360–369, Mar. 1990.