Off-line real-time fault-tolerant scheduling

Cătălin Dima, Alain Girault,
INRIA Rhône-Alpes, ZIRST - 655 Av. de l'Europe,
38330 Montbonnot St. Martin, France

Christophe Lavarenne, Yves Sorel,
INRIA Rocquencourt, Domaine de Voluceau,
B.P.105 - 78153, Le Chesnay Cedex, France

Abstract

We address the problem of off-line fault tolerant scheduling of an algorithm onto a multiprocessor architecture with distributed memory and provide a generic algorithm which solves this problem. We take into account two kinds of failures: fail-silent and omission. The basic technique we use is the replication of operations and data communications. We then discuss the principles which govern the execution of schedulings with replication under the state-machine and the primary/backup arbitrations between replicas. We also show how to compute the execution date for each operation and the timeouts which are used for detecting failures. We end with a heuristic which, using this calculus, computes a possibly non optimal scheduling by finding plain schedulings for each failure pattern and then combining them into a scheduling with replication.

Keywords: *Fault-tolerance, Embedded distributed systems, Scheduling, Dependable systems.*

1 Introduction

Embedded systems are almost always associated to hard real-time constraints, i.e., deadlines whose missing may produce irrecoverable damage to the system. Moreover, such systems are often implemented on distributed architectures for reasons of performance increase, fault-tolerance or topological distribution. One of the main problems when programming such systems is the scheduling of the tasks onto the distributed target architecture such that the deadlines are always met. The two classical options are off-line and on-line scheduling. The off-line technique assures better real-time properties than the on-line technique, i.e., the possibility to meet tighter real-time constraints. In contrast, the on-line technique is more resilient and does not presuppose complete determinacy of the behavior of the system. For embedded systems complete determinacy of the behavior is desirable so off-line scheduling is often preferred to on-line scheduling.

The problem changes dramatically when failures have to be taken into consideration. Since failures cannot, by their nature, be predicted, the very basic assumption for off-line scheduling, the determinism, is demolished and it seems that this technique should leave space to the other [Gh96]. But, as some other studies have shown this is not the case [CP99, GLSS00, BMR99]: a certain degree of nondeterminism can be permitted at the scheduling time within the system. However these studies have focused on processor failures, assuming restricted architecture graphs with reliable channels [CP99, GLSS00] or independent tasks [BMR99]. On the other hand, the studies which focus on channel failures tend to consider only on-line scheduling since they are naturally connected to communication protocols [Kop94, RS91, VC00].

We investigate here the problem of *off-line fault-tolerant scheduling*, where both processors and channels may get faulty and no assumption is made on the topology of the architectures. We provide a heuristic based algorithm which solves this problem. Concretely, our algorithm takes as input a specification of the algorithm to be distributed ($\mathcal{A}$), a specification of the target architecture ($\mathcal{B}$), a list of pattern failures ($\mathcal{C}$), some placement constraints ($\mathcal{D}$), some real-time constraints ($\mathcal{E}$), and information about the execution duration of the algorithm onto the architecture ($\mathcal{F}$). It gives as output a schedule of $\mathcal{A}$ onto $\mathcal{B}$, satisfying $\mathcal{D}$, and tolerant to the failures of $\mathcal{C}$. It also indicates thanks to $\mathcal{F}$ whether or not this schedule satisfies $\mathcal{E}$.

We are not interested here into an algorithm that gives the *best* fault tolerant scheduling w.r.t. the execution durations. This problem embodies the problem of real-time scheduling, which is a well-known NP-complete problem [GJ79], and therefore our problem is NP-complete too. Rather we provide a heuristic that gives *one* scheduling, possibly not the best. This scheduling is then checked for meeting the given real-time constraints. In the eventuality of a negative answer, the user can modify the placement constraints or even add more hardware and start the heuristic again with the modified problem instance. The whole process ends when the given real-time constraints are met.

The contributions of this paper are:

- The statement of the off-line fault tolerant scheduling problem for a generic class of distributed systems lacking centralized control and for two types of failures: fail-silent and omission.

- The statement of the principles that govern the execution of fault tolerant schedulings on distributed architectures.

- An algorithm which solves the stated off-line fault-tolerant scheduling problem.

The paper runs as follows: we introduce, in Section 2, the notion of scheduling with replication. We then discuss the principles that govern the execution of these schedulings in Section 3. The algorithm is presented in Section 4, illustrated with an example. We end with a short section containing conclusions and directions of further study.

2 Schedulings

We work with distributed systems composed of processors and channels. We model this by an *undirected bipartite graph* $G_p = (P \cup C, E)$ where P is the set of nodes that represent the processors, C is the set of nodes that represent the channels and $E \subseteq P \times C \cup C \times P$ represents the processor-channel connections. We call two channels c, c' *adjacent* iff there exists some processor p connected to both, i.e., $(c, p) \in E$ and $(p, c') \in E$.

Processors perform different operations and deliver their results to other processors by means of channels. Each processor-channel connection is governed by a communication coprocessor. The connection between the processor and each of its coprocessors is by means of several buffers of length 1, with blocking read/write facility, meaning that a processor cannot put more than one data into the same buffer[1]. We also assume that distinct communications use distinct buffers. Each coprocessor runs some communication protocol which assures real-time message delivery with a known upper bound on the duration of the communication in the absence of competition for getting control of the channel. Communications on multipoint channels are assumed to be broadcast-like. This feature, combined with a strict order on the `send` and `receive` actions on each coprocessor, is necessary in order to achieve a deterministic behavior as well as the predictability of the maximal duration of execution.

The algorithms to be scheduled are represented by directed acyclic graphs, or *task dags* for short, $G_a = (R, D)$. The nodes $r \in R$ represent operations and the edges $(r_1, r_2) \in D$ represent data dependencies between operations, i.e., the fact that a specific output of the source operation r_1 is needed by the target operation r_2 for its computation. The task dags are to be executed repeatedly on the architecture graph, that is, their execution must be *cyclic*, and the duration of this cycle depends upon the placement of the operations onto the architecture.

An example of a task dag and an algorithm graph is presented in Figure 1 (a and b resp.). In the task dag data are supposed to flow from left to right. In the architecture graph, the nodes P_1, P_2, P_3 and P_4 are the processors and the nodes C_1 and C_2 are the channels.

(a)

(b)

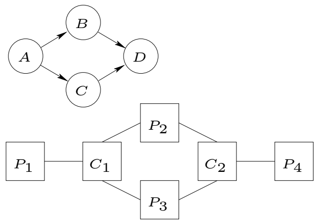

Figure 1. (a) **A task dag;** (b) **An architecture graph.**

The failures we want to tolerate are of two types:

1. **fail-silent**: once a component is faulty it will never recover and it will not provide any output for all inputs it receives henceforth.

2. **omission**: a component may fail to produce the output it should have produced when receiving a certain input, but a new input may produce the correct output *for this new input*, as if the previous input was discarded.

A **failure pattern** is a pair (P', C') with $P' \subseteq P$ and $C' \subseteq C$. The intuition is that the processors in P' and the channels in C' are faulty and therefore it is *the rest of the architecture graph* that has to assure the execution of the task dag. As fail-silent behaviors are particular cases of omission behaviors, when a failure pattern is a mix of both, we consider, for the ease of reasoning, that all the failures are omission failures.

When studying the fault-tolerance problem one usually is concerned with tolerating a number n_f of faults within each cycle of the execution. This can be generalized by considering also that only some designated failure patterns may occur. For example we may consider that for a certain type of processors a certain ratio of failures is tolerable, different types having different ratios. It could also happen that some of our processor nodes are *actuators* whose action cannot be replicated, therefore we cannot tolerate any failure of these actuators. Hence we will take into account *families of failure patterns*, i.e., families $\big((P_i, C_i)\big)_{1 \le i \le t}$ where $P_i \subseteq P$ and $C_i \subseteq C$.

The *placement constraints* and *execution durations* are encoded in the following two functions:

[1] Hence our model of the architecture is a *bounded asynchronous* one.

- $\rho : P \times R \longrightarrow \mathbb{N} \cup \{\infty\}$ defines, for each operation r and processor p, the *maximal* duration of executing r on p. The value ∞ means that r cannot be scheduled on the processor p.

- $\sigma : C \times D \longrightarrow \mathbb{N}$ defines, for each data dependency $d = (r_1, r_2)$ and channel $c \in C$ the *maximal* duration of transmitting the data produced by r_1 and needed by r_2 along c.

We assume that any type of data can be sent onto any channel. In contrast, the restriction that some operation cannot be executed onto some processor stands for situations like the lack of sufficient local resources or for input/output operations.

A scheduling is a mapping associating to each processor and channel in the architecture some sequence of operations, resp. data dependencies in the task dag such that the architecture "behaves like" the task dag. As we want the scheduling to be tolerant to failures we need to have replicas of the same operation on several distinct processors, and similarly for data dependencies. Hence what we call scheduling is slightly more general than the usual notion of scheduling of an algorithm onto an architecture [GLS99]. Formally, a **scheduling** is a pair of functions $S = (f, h)$ with $f : P \longrightarrow R^*$ and $h : C \longrightarrow D^*$ where R^* and D^* are the sets of sequences over R, resp. D. We denote $|f(p)|$ the length of the sequence $f(p)$, $f(p)[i]$ the i-th element in this sequence, and ε the empty sequence. A scheduling must satisfy the following requirements:

1. If $(r', r) \in D$ and $f(p)[i] = r$ then
   - there exists $i' < i$ such that $f(p)[i'] = r'$

   or
   - there exists $c \in C$ with $(p, c) \in E$ and $j \leq |h(c)|$ such that $h(c)[j] = (r', r)$.

2. If $h(c)[i] = (r_1, r_2)$ then there exists some $p \in P$ with $(p, c) \in E$ such that
   - there exists $i \leq |f(p)|$ such that $f(p)[i] = r_1$

   or
   - there exists another $c' \in C$ with $(c', p) \in E$ and $i \leq |h(c)|$ such that $h(c')[i] = (r_1, r_2)$.

3. If $f(p)[i] = r$ and $f(p)[i'] = r$ then $i = i'$; similarly, if $h(c)[i] = (r_1, r_2)$ and $h(c)[i'] = (r_1, r_2)$ then $i = i'$.

4. For each $p \in P$ and $1 \leq i \leq i' \leq |f(p)|$, $\big(f(p)[i'], f(p)[i]\big) \notin D$.

5. For each $r \in R$, there exists $p \in P$ and $i \leq |f(p)|$ such that $f(p)[i] = r$.

The **or**s above are inclusive, i.e., both conditions may occur. Requirement 2 allows *routing* of data dependencies and assume that this takes no time from the "router" processor.

A **plain scheduling** is a scheduling in which each operation is scheduled only once. Hence plain schedulings are the schedulings without failure tolerance.

Finally we are given a real-time constraint Δ as an upper bound on the duration of the scheduling within each cycle.

3 How to "execute" a fault tolerant scheduling

We discuss here the principles which govern the executions of schedulings. This section might also be seen as a discussion of the principles of *code generation*. This discussion is necessary since our model abstracts from the existence of coprocessors. This implies that, after a fault-tolerant scheduling is obtained, the sequence of `send` or `receive` operations *on each coprocessor* has to be deduced from the sequence of communications on each channel.

The first principle is related to *normal* executions of a scheduling, i.e., executions within each cycle in the absence of faults, while the second principle is related to *transitory* executions, i.e., to cycles in which some failure pattern occurs.

The first principle governs "normal" executions of schedulings (i.e., in the absence of faults) and concerns the *arbitration between multiple replicas of an operation*:

Assume operation r is scheduled onto processor p and needs some data provided by operation r'. As fault tolerance is achieved by replication, it will often be the case that multiple copies of r' are scheduled on different processors and send their data to r on different channels. Two problems occur:

1. Which copy of (r', r) is to be used by r when these copies arrive at r on different channels.

2. Which copy of r' will send the data dependency (r', r) on the same channel if two or more copies compete for the channel.

In the first case it is natural to assume that the first communication arrived is the one actually used by r, the others being simply discarded upon their arrival. This is a *state machine* principle [Sch86] of arbitration between copies.

In the second case we require that on each channel c there exists at most one copy of the data dependency (r', r). That is, the different copies of r' *compete* for sending (r', r) and only the *winning copy* will proceed. Classical choices for the arbitration mechanism are:

- The *state-machine* arbitration [Sch86]: the first operation completing its execution is the one that wins the arbitration, the others simply discarding their communication operations.

- The *primary/backup* arbitration [Ja94]: the copies of the same operation are divided into *primary* and *backup*. In a normal execution, it is the primary copy which always delivers the data. When its failure is detected by the backup copies, a coherent choice of a new primary copy is performed using a table of choices computed beforehand.

In the state-machine case there is nothing to choose at runtime: the concurrency between the copies assures that the first that completes its execution is the one that sends the data. In the primary/backup case it is still natural to design the choice of the primary copy as the one that assures the minimal latest time of delivery.

Hence the first principle implies that, for each processor p_i, the generated code for the coprocessor of the connection (p_i, c) contains a `send`, while the coprocessor of the connection (p, c) contains a `receive`. For the state-machine approach, the `send`s must be "conditioned by success" while for the primary-backup approach the `send` of the primary copy is "unconditioned" and the others are "triggered" by some watchdog timeout.

This principle allows us to compute the starting and ending execution time (in the absence of faults) for each scheduling. These are in fact two partial functions α (the *starting time*) and β (the *ending time*) whose definition is presented in the sequel, together with the intuitive explanation:

1. The domains of both α and β consists of

   - Pairs (p, r) where $p \in P$ and $r \in R$ is scheduled on p, i.e. $f(p)[i] = r$ for some $i \leq |f(p)|$.
   - Triples (c, r, r') where $c \in C$ and $(r, r') \in D$ is scheduled on c, i.e. $h(c)[i] = (r, r')$ for some $i \leq |h(c)|$.

   $\alpha(p, r)$ and $\beta(p, r)$ represent the starting and ending execution time for the replica of r which is scheduled on p, while $\alpha(c, r, r')$ and $\beta(c, r, r')$ represent the starting and ending execution time for the replica of (r, r') which is scheduled on c.

2. The replica of r executed on p is executed only after the operation which precedes r on p was executed and only after the reception of at least one copy of each data dependency (r', r) needed by r:

$$\alpha(p,r) = \max\Big(\big\{\min\{\beta(c,r',r) \mid c \in C, \exists j \leq |h(c)| \text{ s.t. } h(c)[j] = (r',r)\big\} \mid (r',r) \in D\big\} \cup \big\{\beta(p, f(p)[i-1]) \mid i \geq 2\big\}\Big)$$

   Here, in the innermost set $\big\{\beta(c, r', r) \mid \exists c \in C, \exists j \leq |h(c)| \text{ s.t. } h(c)[j] = (r', r)\big\}$ the data dependency (r', r) is fixed, only c may vary.

3. The replica of (r, r') transmitted on c is executed only after the communication which precedes (r, r') on c was executed and only after at least one processor connected to p has ended r or after this data dependency has been transmitted on a channel adjacent to c:

$$\begin{aligned}\alpha\big(c,(r,r')\big) &= \\ &= \max\Big[\big\{\beta(c,r_1,r_2) \mid h(c)[j-1] = (r_1,r_2), j \geq 2\big\} \\ &\quad \cup \min\Big(\big\{\beta(p,r) \mid p \in P, \exists j \leq |f(p)| \text{ s.t. } f(p)[j] = r\big\} \\ &\qquad \cup \big\{\beta(c',r,r') \mid \exists c' \in C, c' \text{ adjacent to } c, \\ &\qquad\qquad \exists j \leq |h(c')| \text{ s.t. } h(c')[j] = (r,r')\big\}\Big)\Big]\end{aligned}$$

4. $\beta(p, r) = \alpha(p, r) + \rho(p, r)$ and $\beta(c, r_1, r_2) = \alpha(c, r_1, r_2) + \sigma(c, r_1, r_2)$.

Also, we denote $dur(S)$ the largest ending time of execution of all operations,

$$dur(S) = \max\Big(\big\{\beta(p,r) \mid \exists j \leq |f(p)| \text{ s.t. } f(p)[j] = r\big\} \cup \big\{\beta(c,r,r') \mid \exists j \leq |h(c)| \text{ s.t. } h(c)[j] = (r,r')\big\}\Big)$$

Concerning the computation of the starting and ending execution times in the primary/backup approach, we note that an execution without failures in this approach can be seen as an execution of a plain scheduling by the fact that each operation is entitled to receive all its data dependencies from a single source. Hence, instead of the mixed max-min calculus above, we would have a plain max-calculus, like in e.g. [GLS99].

The second principle is related to a *transitory* execution of a scheduling, i.e., the cycle in which a failure pattern occurs, and we call it the **principle of reconfiguration**. Since we want the system to perform real-time computation and not to stop upon the occurrence of a failure and run some reconfiguration protocol, we need to settle a consistent reconfiguration policy for each processor. The only information that is accessible to each processor at the time of a failure is the absence of a certain communication, which implies that either the source processor, or the communication channel, or both are faulty.

We detect failures using the *watchdog* mechanism: each communication operation on a coprocessor (including the `send` operations!) is guarded by a watchdog which is armed at the moment when the coprocessor has finished the previous communication. The timeout value each watchdog is loaded with represents the latest time at which the communication should take place. When the watchdog reaches its timeout, the coprocessor interrupts its processor and notifies it about the absence of communication.

The processor which is interrupted by a coprocessor due to a watchdog timeout must perform some reconfigurations on its scheduling. These reconfigurations are dependent upon the failure type, i.e., fail-stop or omission:

- In the fail-stop case, the processor drops the faulty communication operation from the sequence of operations to be executed on the coprocessor, since this communication will never take place any more.

- In the omission case the processor has nothing to reconfigure since the faulty communication might take place during a future cycle.

We will also need timeouts for each operation r scheduled on some processor p: each operation r must wait for the arrival of all its data dependencies. Then, this waiting is guarded by a watchdog which is loaded with a timeout representing the latest time r should receive its data dependencies. When the watchdog reaches its timeout, the processor is interrupted from its waiting and drops r is skipped and starts its waiting for the next operation. In the case of fail-silent assumption, r is removed definitively from the scheduling on p. However we require no reconfigurations on the *communications* which were triggered after the execution of r: the failure of just one replica of r does not imply that on the channels on which r was supposed to send its data, say (r, r') there will be no more replicas of (r, r') sent by other processors. It is only after the occurrence of a failure of *all* replicas of (r, r') that we need a reconfiguration on that channel, and this reconfiguration is assured by the use of watchdogs for the communication operations, including the `send`s.

We will present the timeout computation in the full version of this paper. We just mention it is based upon a calculus of maximal execution times, done at the scheduling time. Also we mention that the timeouts corresponding to the primary/backup approach can be much larger than the computed timeouts for the state machine approach, because the delay when the primary copy and several backup copies get faulty is the *sum* of the delays for each of the copy, while in the state machine approach it is simply the *max* of the delays.

The reconfiguration process can be formally described as follows:

1. Suppose $f(p)[j] = r$ for some $j \leq |f(p)|$ and $\exists (r', r) \in D$ such that there exist no $j' < j$ such that $f(p)[j'] = r'$ and there exists no $c \in C$ with $(p, c) \in E$ and with $j' \leq |h(c)|$ such that $h(c)[j'] = (r', r)$.

   Then drop the operation r from $f(p)$ and reduce the size of $f(p)$ by one by shifting the other operations.

2. Suppose $h(c)[j] = (r, r')$ for some $j \leq |h(c)|$. Suppose also that there exists no $p \in P$ with $(p, c) \in E$ such that for some $j' \leq |f(p)|$ we would have $f(p)[j'] = r$. Moreover, suppose that for no $c' \in C$ with c' adjacent to c and no $j'' \leq |h(c')|$ do we have $h(c)[j''] = (r, r')$.

   Then drop the tuple (r, r') from $h(c)$ and reduce the size of $h(c)$ by one by shifting the other operations.

If we cannot apply any of these steps and we did not get a void pair (f, h) (i.e. with $|f(p)| = 0$ for all $p \in P$) then this pair is a correct scheduling.

4 The heuristic

We start with the given dag G_a, the architecture graph G_p, the placement constraints ρ and σ and the family of failure patterns $\big((P_i, C_i)\big)_{1 \leq i \leq t}$. We denote G_{p_i} the reduced architecture graph that results due to the occurrence of the failure pattern (P_i, C_i), defined as: $G_{p_i} := \big(\overline{P}_i \cup \overline{C}_i, \overline{E}_i\big)$ where

$$\overline{P}_i = P \setminus P_i,\ \overline{C}_i = C \setminus C_i,\ \overline{E}_i = E \cap \big(\overline{P}_i \times \overline{C}_i \cup \overline{C}_i \times \overline{P}_i\big)$$

We consider that the failure patterns are *incomparable* one to another, i.e., that for each $1 \leq i < j \leq t$ we have $P_i \not\subseteq P_j$, $P_j \not\subseteq P_i$ or $C_j \not\subseteq C_i$, $C_i \not\subseteq C_j$. Smaller failure patterns, i.e., (P', C') with $P' \subseteq P_i$ and $C' \subseteq C_i$ for some $1 \leq i \leq t$ are tolerated by the fact that their occurrence reduces less the architecture graph.

We first try to schedule the task dag on *each* of the reduced architecture graphs that result due to some failure pattern. This phase is done with the aid of some scheduling algorithm, e.g., SynDEx's [GLS99]. At this time the schedulings are plain. Then, if at this phase, for some of the failure patterns there exists no plain scheduling, the algorithm stops with a negative answer because no scheduling with replication can be found to support this failure pattern. Note that some of the scheduling algorithms may work only if the reduced architecture graphs are *connected*, that is, if none of the failure patterns splits the architecture graph into two or more connected components. If we fall in this case, we might need to run the plain scheduling algorithm for each of the connected components resulting from one failure pattern.

The first phase provides a family of plain schedulings $\big((f_i, h_i)\big)_{1 \leq i \leq t}$. We say that the operation r is *assigned* to p iff r is scheduled onto p in one of the plain schedulings, and similarly for data dependencies. Consider then, for each plain scheduling (f_i, h_i), the "replica" of the task dag, where operations are assigned to processors and data dependencies are transformed into sequences of routings.

Formally, this graph is $G_0^i = (V_i^P \cup V_i^C, M_i)$ where:

$$V_i^P = \{(p,r) \mid \exists j \leq |f(p)| \text{ s.t. } f_i(p)[j] = r\}$$
$$V_i^C = \{(c,r,r') \mid \exists j \leq |h(c)| \text{ s.t. } h_i(c)[j] = (r,r')\}$$
$$M_i = \Big\{\big((p,r),(c,r,r')\big), \big((c,r,r'),(p',r')\big) \mid (p,r),(p',r') \in V_i^P, (c,r,r') \in V_i^C\Big\} \cup \Big\{\big((c,r,r'),(c',r,r')\big) \mid (c,r,r'),(c',r,r') \in V_i^C\Big\} \cup \Big\{\big((p,r),(p,r')\big) \mid (r,r') \in D, (p,r),(p,r') \in V_i^P\Big\}$$

This graph is nothing but a "replica" of the task dag, in which operations are assigned to processors and data dependencies are transformed into sequences of V_i^C nodes (sequences because of the possible reroutings!). We also denote $V^P = \bigcup_{i=1}^t V_i^P$ and $V^C = \bigcup_{i=1}^t V_i^C$.

The plain schedulings $(f_i, h_i)_{1 \leq i \leq t}$ will be transformed into a scheduling with replications as follows: the algorithm inductively builds, for each processor p, the order in which the operations assigned to p are to be executed and, similarly, for each channel c the order in which the data dependencies assigned to c are to be executed. The algorithm starts with the "void" order, in which no operation (or data dependency) is ordered. Each iteration of the algorithm works on the partial order constructed in the previous iteration. In each iteration, an operation or data dependency $x \in R \cup D$, which minimizes some cost function is chosen. If $x \in R$ then for all processors $p \in P$, if $(p,r) \in V_i^P$ for some $i \leq t$ then r is appended at the end of $f(p)$. If $x = (r,r') \in D$ we append (r,r') at the end of $h(c)$ for each $c \in C$ for which $(c,r,r') \in V_i^C$ for some $i \leq t$. The algorithm ends when all the orders are total and this is the scheduling. The combination procedure is presented in figure 2.

The optimality criteria (on which the cost function is based) are the following:

1. Choose for each $i \in I$ the processor with the *minimal timeout*. Hence, whenever we try to place an operation r on a processor p in the failure pattern i, we compute the timeout of this placement and choose the placement with the minimal timeout.

2. Choose for each $i \in I$ the processor with the *earliest execution time*. Hence, whenever we try to place an operation r on a processor p in the failure pattern i, we compute the execution time of this placement and choose the placement with the minimal starting time.

Once this procedure gives the scheduling S, we compute the maximal of the duration of executions in the presence of *any* of the failure patterns:

$$\mu = \max \big\{ dur(S_i') \mid S_i' \text{ is the scheduling which occurs due to the reconfiguration after the occurrence of the } i\text{-th failure pattern} \big\}$$

```
foreach p ∈ P do f(p) := ε; endforeach
foreach c ∈ C do h(c) := ε; endforeach
foreach r ∈ R do unmark r; endforeach
foreach (r, r') ∈ R do unmark (r, r'); endforeach
while ∃ some unmarked item in R ∪ D do
  foreach r ∈ R, r unmarked, do
    if ∀(r', r) ∈ D, (r', r) is marked then
      compute cost(r)
  endforeach
  foreach (r, r') ∈ D such that (r, r') is unmarked
      and is r marked do compute cost(r, r')
    endforeach
  endforeach
  choose some x ∈ R ∪ D with the minimal cost;
  if x = r then
    foreach p do
      f(p) := { f(p)·r   iff (p, r) ∈ V_i^P for some i ≤ t;
              { f(p)     otherwise ;
    endforeach
  endif
  if x = (r, r') then
    foreach c ∈ C do
      h(c) := { h(c)·(r, r')   iff (c, r, r') ∈ V_i^C for some i ≤ t;
              { h(c)           otherwise ;
    endforeach
  endif
  mark x;
endwhile
```

Figure 2. The combination algorithm.

and compare the result with Δ which is the real-time constraint. If $\mu > \Delta$, i.e., the scheduling does not meet the real-time constraint imposed by the user, it is the user's task to modify the placement constraints, the architecture, or even the real-time constraint, and to rerun the algorithm on the modified problem instance.

5 An example.

We start with the task dag and the architecture given in Figure 1, a and b. We use the following constraints:

- $\rho(P_3, r) = \infty$ for all $r \in \{A, B, C\}$, and $\rho(P_i, D) = \infty$ for all $i \in \{1, 2, 4\}$.
- $\rho(P_1, A) = \rho(P_2, A) = 3$, $\rho(P_4, A) = 2$, $\rho(P_i, C) = 3$ for all $i \in \{1, 2, 4\}$; $\rho(P_1, B) = \rho(P_4, B) = 3$, $\rho(P_2, B) = 2$, $\rho(P_3, D) = 3$,.
- $\sigma(C_i, (r, r')) = 1$ for all $i \in \{1, 2\}$ and $(r, r') \in \{(A, B), (A, C), (B, D), (C, D)\}$.

Observe that D is an operation which can be executed only on P_3 (say, P_3 is an actuator).

The failure patterns to be tolerated are

$$\Big(\big(\{P_1\},\emptyset\big),\big(\{P_2\},\emptyset\big),\big(\{P_4\},\emptyset\big),\big(\emptyset,\{C_1\}\big),\big(\emptyset,\{C_2\}\big)\Big)$$

The plain schedulings corresponding to each failure pattern are represented in Figures 3, 4 and 5. Note that in figure 4 the data dependency (A, B) is routed from C_2 to C_1 through P_3. The scheduling with replications which combines these schedulings is represented in figure 6.

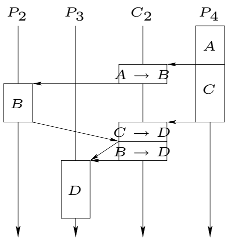

Figure 3. A plain scheduling for the failure patterns $(P_1, \emptyset)$ and $(\emptyset, C_1)$.

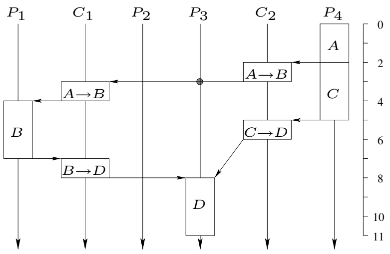

Figure 4. A plain scheduling for the failure pattern $(P_2, \emptyset)$.

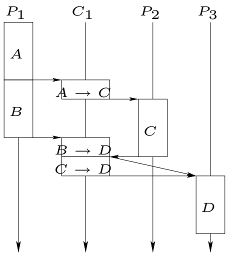

Figure 5. A plain scheduling for the failure patterns $(P_4, \emptyset)$ and $(\emptyset, C_2)$.

The conventions on the scheduling graphs are the following:

- A one-letter labeled box represents an operation; a two-letter labeled box represents a data dependency. The vertical height of a box is proportional to its duration of execution.
- In 6, an arrow which does not touch a box represents a communication that is ignored by its target operation which "already received" the respective data dependency from another source.

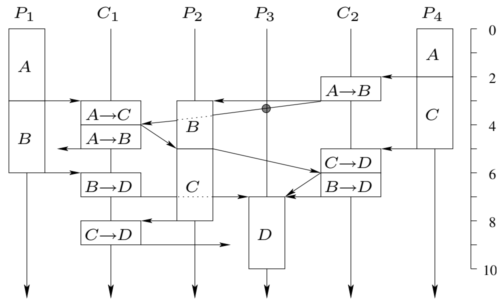

Figure 6. The combination of the plain schedulings in figures 3, 4 and 5.

An example of reconfiguration (in the fail-silent case) after the occurrence of a failure pattern is presented in Figure 7. The initial scheduling is the one in Figure 6, and our example presents the reconfiguration after the occurrence of the failure pattern $(P_1, \emptyset)$.

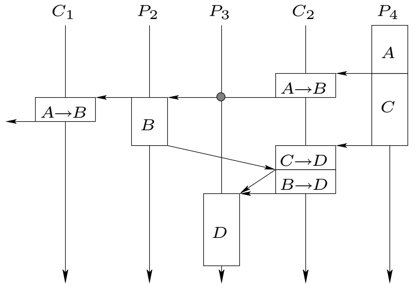

Figure 7. Reconfiguration after the occurrence of failure pattern $(P_1, \emptyset)$.

6 Conclusions

The two main contributions of this article are the formalization of the problem of scheduling an algorithm onto a distributed architecture, with the possibility to tolerate some pre-established family of failure patterns and the algorithm which solves this problem using a greedy-like heuristic. The algorithm gives a possibly suboptimal solution which can be checked for satisfying some real-time constraints. We plan to implement this heuristic and test it on several architectures using different types of channels, i.e., different communication protocols.

One direction of further study is the investigation of some criteria which may assure that some schedulings are better than the others. Another direction is to find protocols which meet our arbitration policy on channels and to formally verify that these protocols correctly combine with the principles of execution.

References

[BMR99] A. Bertosi, L.V. Mancini and F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard real-time systems, *IEEE Trans on Parallel and Distributed Systems,* vol. 10:934-945, 1999.

[CP99] P. Chevochot and I. Puaut. *An approach for fault-tolerance in hard real-time distributed systems*, IRISA Research Report No. 1257, July 1999,

[GJ79] M.R. Garey and D.S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman Company, 1979.

[Gh96] S. Ghosh. *Guaranteeing Fault-tolerance Through Scheduling in Real-time Systems*, PhD Thesis, Univ. of Pittsburgh, 1996.

[GLSS00] A. Girault, Ch. Lavarenne, M. Sighireanu and Y. Sorel. *Automatic Generation of Fault Tolerant Code for Distributed Real-Time Embedded Systems*, INRIA Research Report, 2000.

[GLS99] T. Grandpierre, C. Lavarenne and Y. Sorel. Optimized Rapid Prototyping For Real Time Embedded Heterogeneous Multiprocessors, in *CODES'99, 7th International Workshop on Hardware/Software Co-Design*, Rome, 1999.

[Ja94] P. Jalote. *Fault Tolerance in Distributed Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[KP97] B. Kalyanasundaram and K. Pruhs. Fault-tolerant Real-time Scheduling, in *Proceedings of 5'th ESA'97*:296-307, LNCS 1284, 1997.

[Kop94] H. Kopetz. *TTP/A - The Fireworks Protocol,* research report 23/1994, Institüt für technische Informatik, Technische Universität Wien, 1994.

[RS91] Th. Radehoffer and M. Schroeder. *Automatic Reconfiguration in Autonet*, SRC Research Report 77, 1991.

[Sch86] F.B. Schneider. *Implementing Fault-tolerant Services Using the State-machine Approach: A Tutorial*, TR 86-800, Cornell University, Dept. of Computer Science, 1986.

[VC00] S. Varadarajan and T.-c. Chiueh. *Fault Recovery in a Real-Time Switched Ethernet Architecture*, available at `http://sequoia.ecsl.cs.sunysb.edu/~srinidhi/TPDS/Paper.html`.