INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems

Alain Girault , Christophe Lavarenne , Mihaela Sighireanu , Yves Sorel

No 4006

Septembre 2000

THÈME 4

ISSN 0249-6399

Fault-Tolerant Static Scheduling for Real-Time Distributed Embedded Systems

Alain Girault[*] , Christophe Lavarenne[†] , Mihaela Sighireanu[‡] , Yves Sorel[§]

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet Bip

Rapport de recherche n ° 4006 — Septembre 2000 — 45 pages

Abstract: This paper investigates fault-tolerance issues in real-time distributed embedded systems. Our goal is to propose solutions to automatically produce distributed and fault-tolerant code. We first characterize the systems considered by giving the main assumptions about the physical and logical architecture of these systems. In particular, we consider only processor failures, with a fail-stop behavior. Then, we give a state of the art of the techniques used for fault-tolerance. We also briefly present the "Algorithm Architecture Adequation" method (AAA), used to obtain automatically distributed code. The heart of AAA is a scheduling heuristic that produces automatically a static distributed schedule of a given algorithm onto a given distributed architecture. Our idea is to adapt the AAA method so that it produces automatically a static distributed and fault-tolerant schedule. For this purpose, we discuss several tracks of software implemented fault-tolerance within the AAA method. We present in details two new scheduling heuristics that achieve this goal.

Key-words: Fault-tolerant systems, safety critical systems, real-time systems, software implemented fault-tolerance, Algorithm Architecture Adequation method, SynDEx, static scheduling, distribution heuristics.

(Résumé : tsvp)

This work was funded by Inria under the Tolère research action and has been done while Mihaela Sighireanu had a post-doctoral position at Inria.

[*] Inria-Bip, Alain.Girault@inrialpes.fr.
[†] Inria-Sosso, Christophe.Lavarenne@inria.fr
[‡] University of Paris 7, Liafa, sighirea@liafa.jussieu.fr
[§] Inria-Sosso, Yves.Sorel@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : 04 76 61 52 00 - International: +33 4 76 61 52 00
Télécopie : 04 76 61 52 52 - International: +33 4 76 61 52 52

Ordonnancement statique tolérant aux pannes pour systèmes répartis, temps-réel et embarqués

Résumé : Ce rapport expose les problèmes liés à la tolérance aux pannes dans les systèmes répartis, temps-réel et embarqués. Notre but est de proposer des solutions pour générer automatiquement du code réparti et tolérant aux pannes. Nous commençons par un rappel des caractéristiques des systèmes considérés. En particulier nous ne prenons en compte que les pannes de processeurs et supposons un comportement de type ≪ fail-stop ≫. Ensuite, nous présentons un état de l'art des techniques de tolérance aux pannes développées actuellement. Nous présentons également brièvement la méthode ≪ Adéquation Algorithme Architecture ≫ (AAA), utilisée pour répartir automatiquement du code. Le coeur de AAA est une heuristique d'ordonnancement qui produit automatiquement un ordonnancement réparti et statique d'un algorithme donné sur une architecture répartie donnée. Notre idée est d'adapter la méthode AAA afin qu'elle produise automatiquement du code réparti qui soit en plus tolérant aux pannes. Dans cet objectif, nous discutons différentes pistes et nous détaillons deux solutions pour générer automatiquement du code réparti et tolérant aux pannes, pour lesquelles nous proposons des nouvellles heuristiques.

Mots-clé : Systèmes tolérants aux pannes, systèmes critiques, systèmes temps-réel, méthode Adéquation Algorithme Architecture, SynDEx, ordonnancement statique, heuristiques de répartition.

1 Introduction

1.1 Embedded Systems

Embedded systems account for a major part of critical applications (space, aeronautics, nuclear ...) as well as public domain applications (automotive, consumer electronics ...). Their main features are:

- *duality automatic-control/discrete-event*: they include control laws modeled as differential equations in sampled time and discrete event systems to schedule the control laws;
- *critical real-time*: timing constraints which are not met may involve a system failure leading to a human, ecological, and/or financial disaster;
- *limited resources*: they rely on limited computing power and memory because of weight, encumbrance, energy consumption (e.g., autonomous vehicles), radiation resistance (e.g., nuclear or space), or price constraints (e.g., consumer electronics);
- *distributed and heterogeneous architecture*: they are often distributed to provide enough computing power and to keep sensors and actuators close to the computing sites.

1.2 Synchronous Programming

Synchronous programming [19] offers specification methods and formal verification tools that give satisfying answers to the above mentioned needs. These methods are now successfully applied in industry [46]. They are based upon the modeling of the system with finite state automata, the specification with formally defined high level languages, and the theoretical analysis of the models to obtain formal validation methods. However, the following aspects, extremely important w.r.t. the target fields, are not taken into account:

- *Distribution*: Synchronous languages are parallel, but the parallelism used in the language aims only at making the designer's task easier, and is not related to the system's parallelism. Synchronous languages compilers produce centralized sequential code, which allows the debugging, verifying, and optimizing of the program [37, 5].
- *Fault-tolerance*: An embedded system being intrinsically critical [28, 38], it is essential to insure that its software is fault-tolerant. This can even motivate its distribution itself. In such a case, at the very least, the loss of one computing site must not lead to the loss of the whole application.

1.3 The "Algorithm/Architecture Adequation" Method

The "Algorithm/Architecture Adequation" method [16, 43] (AAA for short) has been successfully used to obtain distributed code optimizing the global computing time on the given

hardware. The typical target architectures are *multicomponent* ones. Such architectures are built from different types of programmed components (RISC, CISC, DSP processors ...) and/or of non-programmed components (ASIC, FPGA, full-custom integrated circuits ...), all together connected through a network of different types of communication components (point-to-point serial or parallel links, multi-point shared serial or parallel buses, with or without memory capacity ...). They typically include less than 10 processors.

AAA is presented shortly in Section 4. Basically, it first produces a static distributed schedule of a given algorithm onto a given distributed architecture, and then it generates a real-time distributed executive implementing this schedule. The nice point is that AAA preserves the above mentioned properties of synchronous programs. It is implemented in the SynDEx tool[1], which uses heuristics to automatically produce and optimize the desired real-time static schedule.

1.4 Motivation of this Work

Our goal is to produce automatically *distributed fault-tolerant code*. Taking advantage of AAA, we propose new scheduling heuristics that will produce automatically a static distributed fault-tolerant schedule of the given algorithm onto the given distributed architecture.

Our solutions must adapt existing work in fault-tolerance for distributed and real-time systems to the specificities of embedded systems and of AAA. In particular, the fault-tolerance should be obtained without any help from the user (automatically distributed constraint) or any added hardware redundancy (embedded system constraint). It will therefore fall in the class of software implemented fault-tolerance. The second requirement is essential: it implies that we have to do with the existing parallelism of the given architecture, and that we won't add extra hardware. Moreover, in order to perform optimizations and to minimize the executive overhead, the scheduling used in AAA is completely static: all scheduling decisions are taken off-line at compile time [16] based on the characteristics of each algorithm's operation relatively to the hardware component on which it is executed. Finally, neither the algorithm to be executed nor the architecture of the system are fixed, but they are inputs of the method. For these reasons, we cannot apply the existing methods, proposed for example in [1, 7, 4, 15, 14], which use preemptive scheduling or approximation methods.

There exists a huge amount of work in the domain of fault-tolerance of distributed and/or real-time systems. The following books summarize a part of this work:

- [28] gives an exhaustive list of the basic concepts and terminology on fault-tolerance,
- [38] gives a short survey and taxonomy for fault-tolerance and real-time systems, and
- [8] and [21] treat in details the fault-tolerance in distributed systems.

[1] SynDEx (Synchronized Distributed Executive) is available at the url `http://www-rocq.inria.fr/syndex`

We have to study the existing solutions in order to direct our research. For this reason, we build up a state of the art of the existing work, classified according to the class of systems it considers. This study allows us to cover the existing solutions and to propose two new scheduling heuristics. We discuss and compare each solution, and show that each is well adapted to different kinds of hardware architecture.

1.5 Paper Outline

Section 2 defines the context of this paper, i.e., distributed systems, real-time systems, and dependable systems. Section 3 presents an overview of the fault-tolerance techniques with a strong emphasis on distributed and real-time systems (readers familiar with real-time systems and fault-tolerance can skip these first two sections). Section 4 presents the principles of the AAA method and shows the relation with the context presented previously. Based on this overview, Section 5 refines our initial problem by making assumptions and discussing design choices driven by the AAA principles. Sections 6 and 7 present the two proposed solutions for providing fault-tolerance within AAA. Finally, Section 8 summarizes the more important issues and gives some concluding remarks.

2 Context of the Work

2.1 Distributed Systems

Since our focus is on automatically distributed code for distributed, real-time systems, we define in this section what we mean by a distributed system.

Remark 1 *This presentation summarizes the one given in [21, Chapter 2]. A presentation of distributed systems from the exclusive point of view of fault-tolerance is given in [8].*

There are two ways of viewing a distributed system:

- The *physical model* defines a distributed system by its physical components. In this model, a distributed system consists of several computers, usually called *nodes*, that are geographically at different locations, but are connected by a communication network.

  At this level, the distributed system is characterized by its physical *clocks*, by its *coupling*, and by its *communication network*.

- The *logical model* defines a distributed system from the point of view of processing, or computation. In this model, a distributed system consists of a finite set of *processes* and *channels* between the processes. Channels represent the logical connections between the processes, i.e., their interaction through messages.

  At this level, the distributed system is characterized by its logical *clock* and by its *time bounds* on execution and communication.

Physical clocks. In a distributed system, each node has a clock of its own, which can be used to control the instructions executed at that node, but it cannot control the instructions of an other node. In contrast, in a parallel systems, it is likely to have a global clock, which is used for controlling instructions of the different processing elements in the system.

Coupling. In a distributed system, the nodes can communicate only by message passing, i.e., the nodes have their local (non-shared) memory. In this case, nodes are said to be *loosely coupled*, i.e., the runs of nodes do not have timing constraints and can tolerate things being done in different sequences than those expected. In contrast, in a parallel system, the nodes communicate by a shared memory, and are are said to be *closely coupled*.

Note that this criteria does not appear in the classical classification of Flynn [12], where only the parallelism of computations is considered. In the Flynn classification, the distributed systems belong to the MIMD class.

Interprocess communication. At the physical level, communications between nodes are supported by a communication network consisting of several links, each link being either:

- a *point-to-point* link connecting two nodes through their respective network interface
- or a *multi-point* link (a bus) connecting several nodes of the network.

In the general case, a multi-point link may be considered like a set of point-to-point links. However, this abstraction is not always correct in the context of fault-tolerant distributed systems, for two reasons:

- The concurrent communications are always serialized over a multi-point link, whereas they may be executed in parallel inside a system using a set of point-to-point links. This is important for fault-tolerance since a "parallel" replication of communications may be done for point-to-point links, but not for a single multi-point link.
- A multi-point link may provide facilities for multi-cast communication and so avoiding the multiple send of a message. This also is important for fault-tolerance since one node may inform atomically several nodes about the execution of an operation by sending a single message.

The manner in which the different links are connected to different nodes is called the network *topology*. Finally, sending messages on a network requires a *communication protocol*.

Remark 2 *Many communication protocols (TCP/IP, OSI, CAN) ensure reliable communication between nodes in the network. Typically, some part of the protocol is executed by the hardware in the network interface and the remainder is performed by system software.*

Logical clocks. In the logical model of a distributed system, many processes are executed concurrently. In many schemes and algorithms, it is important to be able to specify whether an event occurred before another event or not. The relation $\rightarrow$, defined in [25], captures the "happened before" relation between two events in a distributed system. This relation is defined w.r.t. the order imposed by the communication.

The relation $\rightarrow$ defines a partial order. The logical clocks attach times to events in the system, such that the time-stamps they assign to events are consistent with the partial order defined by $\rightarrow$. The time-stamps define a total order of events on a processor and on a distributed system.

Time bounds. From the point of view of the time bounds of the execution, systems may be classified as [33]:

- *synchronous*, if whenever the system is working correctly, it always performs its intended function within a finite and known time bound. So, a *synchronous communication channel* is one in which the maximum message delay is known and finite. A *synchronous process* is one in which the time required to execute a sequence of instructions has a finite and known bound.
- *asynchronous* otherwise.

Since processes are executed in a distributed system by autonomous nodes, the only manner to observe that a process failed is the lack of a response. The main advantage of a synchronous system is that the failure of a component can be inferred from the lack of response within some defined time bound. Hence, if the distributed system is synchronous, a timeout-based scheme for detecting node failures or message losses may be employed.

Relationship between the physical and the logical models. Concurrent processes may be executed on a single node (processor) or on several nodes.

At the logical level, the topology of the network is not taken into account. The nodes of the network are considered to be fully connected, i.e., any node can send a message to all the other nodes.

From the point of view of the size of communication channels, the communication by message passing may be:

- *synchronous* (or zero place FIFO, or rendezvous), when the sender and the receiver are synchronized on the communication;
- *asynchronous* (or finite FIFO), when the sender and the receiver are not synchronized on the communication but are synchronized on the reception of the message (i.e., the receiver blocks when the buffer is empty and the sender blocks when the buffer is full);
- *fully asynchronous* (or infinite FIFO), when the buffer storing the messages is considered infinite, so only the receiver blocks when the buffer is empty.

Asynchronous and synchronous message passing is different from asynchronous and synchronous distributed systems. A synchronous distributed system may support both asynchronous and synchronous message passing.

2.2 Real-Time Systems

In this section we introduce some classical concepts used in the design of the real-time systems.

A real-time system is one whose correctness depends not only of its outputs, but also on the times at which they are produced. A real-time system must provide predictable response times. Therefore, the interprocess communications and synchronizations in a real-time distributed system must be *bounded in time* and *predictable*.

Timing constraints. Real-time systems may be classified w.r.t. timing constraints into two classes:

- *Hard real-time* systems have stringent timing constraints and the consequences of missing deadlines are potentially catastrophic.
- *Soft real-time* systems have less stringent timing constraints and the value of a process that missed its deadline have still a (diminished) utility.

Tasks. The processes in a real-time system are usually called *tasks*. The classical [44] states of a task and the possible transitions between them are given on Figure 1.

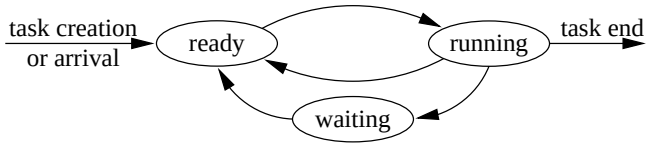

Figure 1: Task's states.

In real-time systems, tasks have time related attributes such as:

- The *deadline* is the late instant when the task must end.
- The *jitter* is the time between the input of an event and the response of the task to this event.
- The *computation time* is the time between the task creation and its end.
- The *period* is the time between two consecutive creations of a task.

Figure 2 illustrates some of these attributes:

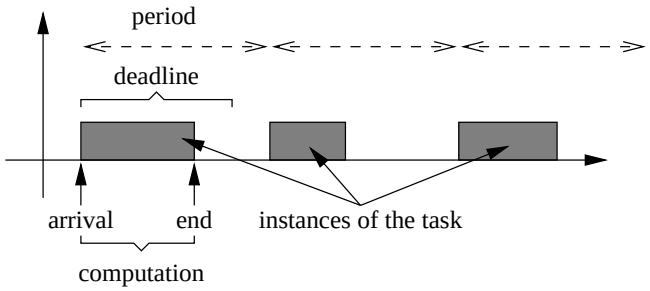

Figure 2: Time related attributes for a task.

With respect to the period, tasks may be classified as:

- *Periodic tasks* occur repeatedly after a fixed duration of time. For instance, process W

  - *Static priority* where the priorities of all tasks are assigned when the tasks are scheduled and remain unchanged for all executions of a tasks. The most known policy in this class is the Rate Monotonic Scheduling (RMS), where the priority of a task is inversely proportional to its period.
  - *Dynamic priority* where the priorities of a task may change during the execution of a particular instance of that task or for different instances. The most known policy in this class is the Earliest Deadline First (EDF), where at any time the task with the earliest deadline has the highest priority.

- *Non-preemptive* scheduling deals with tasks that cannot be interrupted during their execution; it is often used in systems where the overhead caused by preemption is too high, and the absence of interruption makes system validation easier [14].

  Almost all non-preemptive scheduling algorithms use *timeline-driven dispatching*. A time-line is used to schedule the tasks. Thus, the times at which a task will start and finish are known in advance. The general problem of optimal scheduling of non-preemptive tasks is NP-complete [13]. Therefore, different heuristics have been used to schedule non-preemptive real-time tasks with the aim of maximizing performance measures such as processor usage. For multiprocessor systems, various heuristics for non-preemptive scheduling have been developed in [36].

2.3 Dependable Systems

In this section, we introduce the classical concepts used in the domain of *dependable systems*, which includes the fault-tolerant systems.

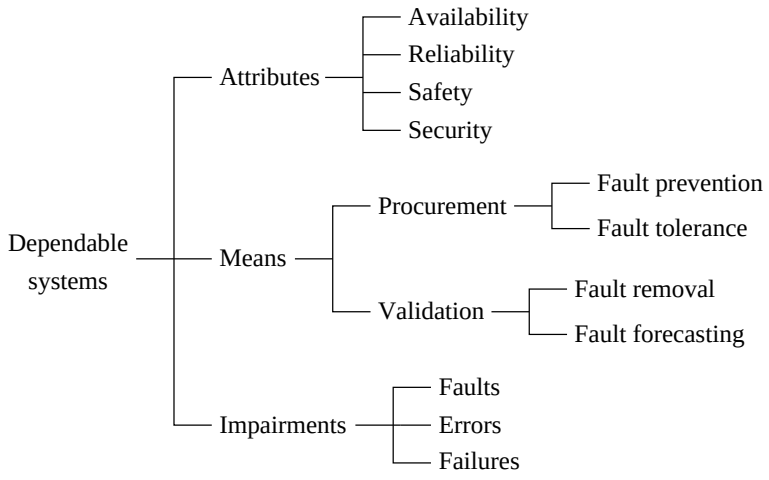

Figure 3: Tree of the dependable systems.

The *dependability* concept was introduced by Laprie [28] as a contribution to a standard framework and terminology for discussing reliable and fault-tolerant systems. A dependable system is one for which reliability may justifiably be placed on certain aspects of the quality of service that it delivers. The quality of the service includes both correctness and continuity of its delivery. Figure 3 shows the main notions which are related to dependable systems.

In this paper we focus on the fault-tolerance. In the following, we introduce some classical definitions used in fault-tolerance.

Faults. *Fault*, *error*, and *failure* are three important terms in the field of fault-tolerance. Figure 4 shows the causal relationship between these three terms.

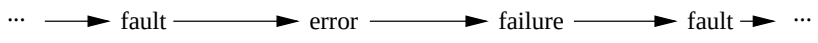

Figure 4: Relationship between impairments.

- A *fault* is a defect or flaw that occurs within some hardware or software component.

  Not all faults produce immediate failure: faults may be *latent* (activated but not apparent at the service level) and then become *effective* (affecting the services). Fault-tolerant systems attempt to detect and correct latent errors before they become effective.

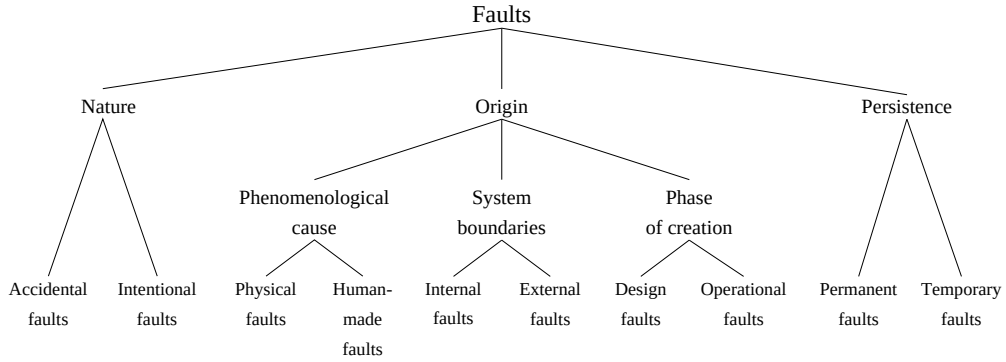

Figure 5: Classes of faults.

Faults may be classified using the following criteria, which are summarized on Figure 5 [28, page 15]:

- by their nature: accidental or intentional;
- by their origin: physical, human, internal, external, conception, operational;
- by their persistence: transient or permanent. A special kind of transient faults are the intermittent fault, i.e., repeated occurrences of transient faults.

- An *error* is a manifestation of a fault and is a deviation from accuracy.
- A *failure* ("*défaillance*" in French) is a departure of a system from the service required.

  Failures may be classified using different criteria [28, page 62]: the domain of the failure, the perception of the failure by the users, and the consequence of the failure on the environment. Using the domain of the failure, failures may be classified into:

  - *failures on values*, also called *response* failures [8]: the component delivers an incorrect service or value, and
  - *timing failures*: the timing conditions of the delivery are not satisfied; the timing failures may be *late* or *early*.

  The combination of these two classes gives several kinds of failures:

  - *omission failure*: the component omits to respond to an input;
  - *crash failure*: a permanent omission failure.
  - *stop failure*: the activity of the component is not observable and a constant, arbitrary value is provided for the output;
  - *arbitrary failure*: the timing conditions and the value of the service are not always correct.

  For distributed systems, a classical semantics of the failures of network nodes (called servers) is given in [8].

Means for fault-tolerance. The fault-tolerance is built on [28]:

- *error processing* which aims at removing errors from the computational state, if possible before failure occurrence;
- *fault treatment* which aims at preventing faults from being activated again.

Error processing may be carried out in two ways [28]:

- *error recovery*, where the erroneous state is substituted with a correct state by:
  - *backward recovery*, i.e., the system brings back a previous correct state or
  - *forward recovery*, i.e., the system rolls up to a next correct state.
- *error compensation*, where the error may be masked using the internal redundancy of the system.

Fault treatment consists of at least two steps [28]:

- *fault diagnostic*, which determines the cause(s) of error(s) in terms of location and nature, and

- *fault passivation*, which prevents the fault(s) from being activated again, thus making it (them) passive.

The last step is carried out by removing the component(s) identified as being faulty from further executions. If the system is no longer able to deliver the same service as before, then a *reconfiguration* may take place.

A more usual (but not standard) term used as a mean for fault-tolerance is the treatment of an error using a "damaged mode", which aims at providing the same service as the initial system in the presence of faults. This term is very ambiguous w.r.t. the mean employed for fault-tolerance, because it does not specify if the fault is eliminated or not. For example, a "damaged mode" may be either an error recovery followed by the continuation of the service in the presence of the fault, or a fault treatment followed by a reconfiguration. For this reason, we prefer to used the terms proposed by Laprie in [28].

Redundancy in fault-tolerance. All the means discussed above use the *redundancy* in order to treat errors. There are three forms of redundancy [21]:

- *Hardware redundancy* comprises the hardware components that are added to the system to support fault-tolerance. For instance, a spare processor can be used if one of the running processor fails; or three processors can be associated with a voting mechanism to deliver the correct result, even if one of the three processors fails.
- *Software redundancy* includes all programs and instructions that are employed for supporting fault-tolerance. For instance two implementations of the same algorithm can be computed and their results compared.
- *Time redundancy* consists in allowing extra time for performing tasks for fault-tolerance. For instance a faulty sequence of instructions can be re-executed.

Phases for fault-tolerance. In the solutions proposed for fault-tolerance, there are four phases:

1. *Error detection:* The presence of a (latent) fault is deduced by detecting an error in the state of the system.
2. *Damage confinement:* Any damage due to the failure is identified and delimited.
3. *Error recovery:* The erroneous state is replaced by an acceptable valid state.
4. *Fault treatment and continued system service:* The faulty component is identified and the system is executed such that the faulty components are not used or are used in a manner such that the fault does not cause any more failures.

3 Related Work on Fault-Tolerance in Distributed Systems

Since the systems we consider are distributed systems, we focus on the fault-tolerance in distributed systems. However, whenever appropriate (e.g. software design faults), the uniprocessor case is discussed as a special case of distributed systems.

Following the approach of Jalote [21], we present the fault-tolerance in distributed systems as a stack of different levels of abstraction, each level providing different fault-tolerant services. The various levels are shown on Figure 6.

At the lowest level are abstractions which are frequently needed by techniques for fault-tolerance at higher levels, also called "basic building blocks". The next five levels deal with fault-tolerant services to ensure either the consistency or the continuity of service under node failures or user design faults.

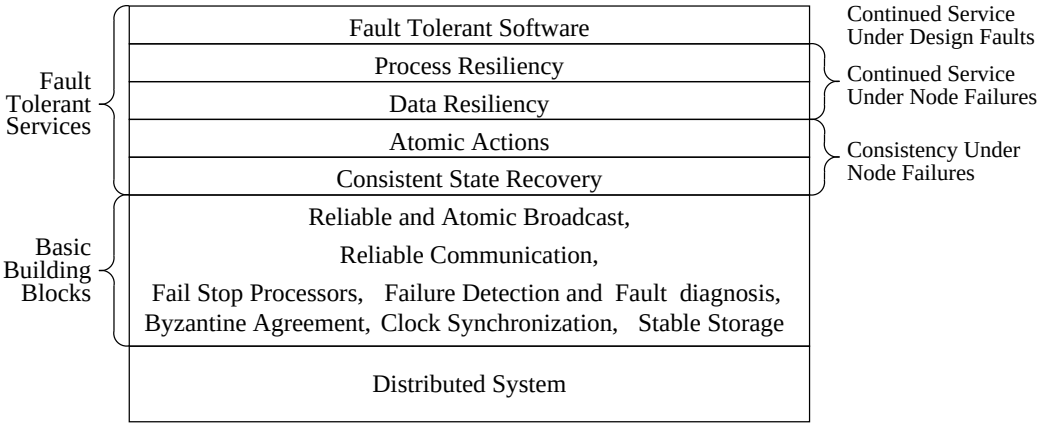

Figure 6: Levels in a fault-tolerant distributed system

It is worth noticing that, since our systems are distributed automatically, all the classical solutions proposed for the fault-tolerance for distributed systems *may not* be feasible. We discuss in Sections 6 and 7 the solutions which may be retained in our context.

3.1 Basic Building Blocks

Most schemes to provide fault-tolerant services in a distributed system make implicit and explicit assumptions about the behavior of the system. These assumptions are called *basic building blocks* [21]. A system using ordinary components does not always satisfy these properties, and specific software or hardware is needed to convert ordinary components into components that satisfy these assumptions. In the following paragraphs, the most common assumptions are discussed.

Byzantine agreement [21, Section 3.1] is the problem of reaching agreement between processors that can fail in an arbitrary manner. It has been shown in [26] that, with ordinary messages, if there are n nodes in the system, then reaching agreement is possible only if the number of failed nodes is strictly less than $n/3$. Several protocols have been proposed to solve the Byzantine agreement problem in a distributed system.

Clock synchronization [21, Section 3.2] is the problem of keeping the clocks of different nodes in a distributed system synchronized. That is, at any time, the clock values of two nodes differ by at most a constant. This property is needed in order to impose a total order between events. Clock synchronization approaches can be deterministic or probabilistic. In deterministic clock synchronization protocols, a required precision is guaranteed with the same condition as for Byzantine agreement. Most of the deterministic clock synchronization protocols assume some maximal bound on message delay in order to detect the absence of messages. Probabilistic clock synchronization can only ensure that the clocks will be synchronized with a very high probability, without any assumption about maximum message delay.

Stable storage [21, Section 3.3] ensures that read and write operations on the storage are always successful, even if the underlying storage hardware components fail. This is useful for fault-tolerance methods requiring that some state of the system be available after a failure occurs. [27] defines the problem of the stable storage and gives a protocol that works with one disk in the presence of arbitrary faults. Other classical protocols are disk shadowing and Redundant Arrays of Inexpensive Disks (RAID).

Fail stop processors [21, Section 3.4] is the assumption made on the behavior of a processor which stops (halts) upon failure. More precisely, the visible effects of a failure of a fail stop processor are [39, 40]:

- It stops executing.
- The internal state and the contents of the volatile storage connected to the processor are lost; the state of the stable storage is unaffected by it.
- Any processor can detect the failure of a fail stop processor.

The implementations proposed for this assumptions define a *k-fail-stop processor* as a computing system that behaves like a single fail stop processor unless $k+1$ or more components of the system fail [39]. In the model of computation used by these implementations, there are processors for running programs and processors that provide the stable storage. Clock synchronization is also supposed to be ensured.

In general, real processors are not fail stop and may leave their global memory in an inconsistent state when a failure occurs.

Failure detection and fault diagnosis [21, Section 3.5] is the assumption that the failure of a node is detected by other nodes in the system in a finite time. This assumption is essential for other nodes to perform any activities that may be needed for recovery. Failure detection can be viewed as a fault diagnosis problem, in which the goal is for the set of non faulty components to detect the set of faulty components, by using diagnostic tests and by disseminating information of their tests.

This assumption is easy to satisfy by a brute force method, in which each node uses timeouts to detect failures. However, in a large system, this approach may be costly (if each node has to test all other nodes) and unwieldy (there may be coherence problem between the test of each node). Considerable amount of research has been done in the fault diagnostic area. We only consider here the work on distributed systems having node failures.

For the purpose of determining how diagnosticable a system is and for performing diagnosis, a model is used. The PMC (Preparata, Metze, and Chien [34]) model was the first model proposed. It decomposes a system into a set of units $\{u_1, ..., u_n\}$, where some units can test the status of other units. The graph representing the testing relation (u_i tests u_j) is called the *connection assignment* graph. The edges of this graph are labeled with the result of the test as follows:

$$a_{ij} = u_i \text{ tests } u_j = \begin{cases} x & \text{if } u_i \text{ is faulty} \\ 1 & \text{if } u_i \text{ is non faulty and } u_j \text{ is faulty} \\ 0 & \text{if } u_i \text{ is non faulty and } u_j \text{ is non faulty} \end{cases}$$

The set of all the labels (a_{ij}) is called the *syndrome* of the system. In the PMC model, the syndrome is assumed to be analyzed by a centralized supervisor, which is an ultra-reliable processor. In this case, some conditions on the number of faulty nodes and the connection assignment graph are needed for a system to be diagnosable.

However, there exist approaches for distributed diagnosis, like *adaptive Distributed System-level Diagnosis* (adaptive DSD) [22, 23]. There is no bound on the number of faulty nodes in the system for diagnosis to succeed. The "test" used creates a subprocess on the node tested in order to verify several hardware or software facilities.

Reliable message delivery is the assumption that a message sent by a node to another node arrives uncorrupted at the receiver, and that the message order is preserved between two nodes. Real communication lines sometimes lose messages and introduce errors. Hence, some protocols are needed to ensure these assumptions. These protocols typically belong to the communication architecture requirement and are usually supported by the communication protocols. The key issue in handling failures is the *routing of messages*. For further details on reliable communication protocols see [45].

Reliable, atomic, and causal broadcast [21, Chapter 4] is the assumption that broadcast or multi-point communications mechanisms are available in the system and these mechanisms have three properties:

- *reliability*, which requires that a broadcast message be received by all the operational nodes;
- *consistent ordering*, which requires that different messages sent by different nodes be delivered to all the nodes in the same order;
- *causality preservation*, which requires that the order in which messages are delivered at the nodes be consistent with the causality between the send events of these messages.

These properties bring in three different types of broadcast primitives:

- *reliable broadcast* which supports only reliability property;
- *atomic broadcast* which supports reliability and ordering properties;
- *causal broadcast* which supports causality preservation property.

3.2 Fault-Tolerant Services

In the previous section we have discussed the basic building blocks and the communication architectures that are useful for developing fault-tolerant distributed applications. The services presented previously can be supported either by the hardware (e.g., the CAN bus for reliable communication) or by the operating system. In the following, we present services which are specific to some systems and need special mechanisms to be implemented.

Recovering a consistent state [21, Chapter 5] is a fault-tolerant service which allows the restoring of the system to a consistent state. This service completes the basic service of error recovering to an error-free state.

In uniprocessor systems, once an error is detected, error recovery is not difficult, mainly the backward recovery. *Checkpoints* (or recovery points) are established periodically during the normal execution of the process by saving all the information needed to restart a process on some stable storage device. When an error is detected, the process is rolled back to its previous saved state by restoring the last saved checkpoint of the process.

In a distributed system, the rollback is not straightforward. The problem is that though a consistent global state is needed, checkpointing and rollback can only be performed by individual processes. Hence, the problem is to ensure that the checkpointing and rollback done locally by each process is such that after the rollback, the global system state is consistent. A system state is consistent if it can be reached by a failure-free execution of the system.

There are two basic approaches to checkpointing and rollback:

- *Asynchronous checkpointing:* Processes establish checkpoints independently, but keep a record of their communications. If a rollback is performed, then the communication history is used to roll back the processes such that there are no missing or orphan

messages. The asynchronous checkpointing is simple because it does not need coordination between processes. However, it may have a domino effect (no consistent state is found) and it may require a lot of local checkpointing. As a result, it is feasible when failures are rare and communication between processes is limited.

- *Synchronous checkpointing:* (or distributed checkpointing) The processes cooperate in order to establish their local checkpoints, such that the set of checkpoints together is guaranteed to be consistent. Cooperation between processes requires interprocess communication for establishing local checkpoints. Hence, the checkpointing becomes more complex, but the recovery is simplified.

Atomic actions [21, Chapter 6] is a fault-tolerant service whose goal is to make some identified user-defined operations appear as atomic, even if failures occur in the system. An atomic action is an operation that appears as primitive and indivisible to all other operations in the system. Atomic operations have two fundamental properties:

- an atomic operation either completes fully, or it appears as if the operation had not performed any action at all, and
- no two atomic operations should appear to overlap or execute concurrently; therefore atomic operations are always executed sequentially in some order.

A lot of protocols have been designed to ensure atomicity of operations for both centralized and distributed data.

Data replication and resiliency [21, Chapter 7] is the fault-tolerant service which masks node and communication failures in the distributed system to users, in order to perform operations on data.

Resiliency is supported by replicating the data on multiple nodes. However, the replication should be transparent to the users of the data. That is, the execution of a sequence of actions should be equivalent to a sequential execution of these actions on non-replicated data. This correctness property is called the *one-copy serializability* criterion. The algorithms which support this criterion are also called replica control algorithms. One-copy serializability requires that the different copies of a data object be in a mutually consistent state, so the user actions get the same view of the data object. Due to this consistency criterion, the algorithms are also called consistency control algorithms.

There are two basic approaches for managing replication:

- *Optimistic* approach, where no restriction is placed on access to data; if a partition of the network occurs, for instance due to a node or link failure, the copies of the data in different groups may become divergent. The inconsistencies between copies are solved after the groups of the partition rejoin, and are not always successful. A method to solve inconsistencies is the *version vector*.

- *Pessimistic* approach, where the access to replicated data is controlled such that inconsistency never occurs. There are three pessimistic approaches for replica control:
  - *Primary site approach*, where each given data has a designed primary site, while the others are designated as backups. All the requests for operations on the data are sent to the primary site. The primary site periodically performs checkpoint of the state of the data on the backups. When the primary site fails, an election algorithm is run to designate a backup as the new primary site. When a communication failure causes the network to be partitioned, only the partition that includes the primary site is able to perform operations on the given data. This requires that the nodes which are alive must be able to distinguish between node failures and network partitions.
  - *Active replicas approach* is also known as the state machine approach. In this approach, all replicas of the data are kept simultaneously active. A request for an operation is sent to all the replicas, and any replica can service the request. To ensure that the replicas are mutually consistent, all the requests for operations should be sent to all replicas, and the different requests must be processed in the same order on each site. For these reason, the *atomic broadcast* of the requests is needed in the active replicas approach.
  - *Voting approach* where, for performing a read or an update operation on the data, the requesting node has first to get a "quorum" of votes from the other nodes. The group of nodes that give the quorum is such that a group performing a read operation always has a node with the latest update, and the mutual exclusion is supported. Voting algorithms can be static or dynamic. In static voting, all the parameters for voting are fixed, while in dynamic voting, some parameters may be changed as failures and recoveries take place in the system.

The main issue of replication is to find the optimum degree of replication, because the cost of managing the replicas may be important.

Process resiliency [21, Chapter 8] is the fault-tolerant service which ensures that a distributed computation proceeds despite the failure of some of its constituent processes. If the system consists of processes that do not communicate with each other, each process performs a checkpoint of its state periodically; when a process fails, it is restarted on one of its backup. If the processes communicate, the resiliency must be provided for the communication mechanisms used: remote procedure call, asynchronous message passing, or synchronous communication.

The main approach to support process resiliency is to replicate the process on two nodes: a primary and a backup. Hence, there are process pairs with a *primary process* and a *backup process*.

4 AAA Method and SynDEx

4.1 Principles of AAA

AAA [43] takes as input some real-time constraints, some distribution constraints, and two specifications, one describing the application algorithm to be distributed, and one describing the topology of the target architecture. The algorithm specification consists of operations and data-dependencies. The architecture specification consists of processors and communication links. The distribution constraints should assign a set of processors to each operation of the algorithm graph. As we have said in Section 1.4, we also need the characteristics of each operation relatively to the hardware component on which it is executed. Hence, the distribution constraints consist in assigning to each pair ⟨operation, processor⟩ the value of the execution duration of this operation onto this processor. Each value is expressed in time units, and the value "∞" means that this operation cannot be executed on this processor. Since we also want to take into account inter-processor communications, we assign a communication duration to each pair ⟨data dependency, communication link⟩, also in time units.

Both the algorithm and the architecture are specified as graphs. Then, the implementation of an algorithm on a distributed architecture is formalized in terms of graphs transformations. More precisely, AAA proceeds in two steps:

1. First it produces a *static distributed schedule* of the algorithm graph operations onto the processors, and of the data-dependencies onto the communication links. The real-time performances of the implementation are optimized by taking into account inter-processor communications which are critical.

2. Then, from this static schedule, it produces automatically a *real-time distributed executive*, and ensures the synchronization between the processors, as they are required by the algorithm specification. The obtained distributed executive is guaranteed to satisfy the real-time constraints, without deadlock and with minimum overhead.

The SynDEx [29] tool implements AAA. The architecture and the algorithm graphs can be both drawn with SynDEx's graphical user interface. The latter graph can also be imported from a file which is the result of the compilation of a source program written in synchronous languages like Esterel [3], Lustre [20], or Signal [31], through the common format DC [47].

In the following, we resume the main characteristics of the three models used in AAA: the algorithm model, the architecture model, and the implementation model. For a more detailed description, the reader should refer to [43, 48].

4.2 Algorithm Model

The algorithm is modeled by a *data-flow graph*. Each vertex is an *operation* and each edge is a *data-flow* channel. Since we deal with reactive systems which are in constant interaction

with the environment that they control, the algorithm is executed repeatedly for each input event from the sensors in order to compute the output events for actuators. The algorithm is therefore an infinitely wide, periodic, and directed acyclic dependence graph, reduced by factorization to its repetition pattern [30]. We call each execution of the data-flow graph an *iteration*. This model exhibits the potential parallelism of the algorithm through the partial order associated to the graph. Graph operations are of three kinds:

1. A computation operation (`comp`). The inputs must precede the outputs, whose values depend only on input values. There is no internal state variable and no other side effect: we say that `comps` are *safe*.

2. A memory operation (`mem`). The data is held by a `mem` in sequential order between iterations. The output precedes the input, like a register in Boolean circuits: we say that `mems` are *memory-safe*.

3. An external input/output operation (`extio`). Operations with no predecessor in the data flow graph stand for the external input interface handling the events produced by the environment (sensors). In this sense an `extio` is an operation of a higher level than an input/output driver. For instance, a given input `extio` might be in charge of collecting data produced by several sensors, and implementing some sensor fusion mechanism before outputting its value. Symmetrically, operations with no successor stand for the external output interface generating the reactions to the events (actuators). For instance, a given output `extio` might be in charge of sending to an actuator a value computed from its input.

   The `extios` are the only operations with side effects: we say that they are *unsafe*. However, we assume that two executions of a given input `extio` in the same iteration always produce the same output value.

An example of algorithm graph is given on Figure 7. It is composed of six operations: I and O are `extios` (respectively input and output), while A–E are `comps`.

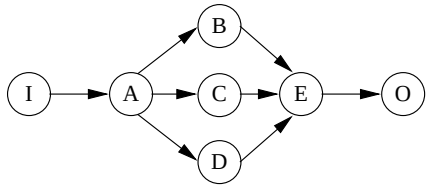

Figure 7: An example of algorithm graph: I and O are `extios`, A–E are `comps`.

4.3 Architecture Model

The parallel computing architecture is a network of processors connected by bidirectional communication links which may be point-to-point links or multi-point links (buses). Each

processor is made of a RAM memory shared by one computation unit, and one or several communications units which are connected to the communications links. Each computation unit is able to sequentially execute algorithm's operations. Communication units connected through a link are able to sequentially execute data transfers, called `comms`, between the memories of the processors. Also we can compute, for each data transfer, a worst case upper-bound for its transmission between any two processors.

The architecture is modeled by a non oriented hyper-graph, where each vertex is a computation unit or a communication unit, and each hyper-edge is a communication link (when it connects communication units) or a memory (when it connects communication units to computation unit).

Figure 8 represents an architecture graph with three processors and two point-to-point links.

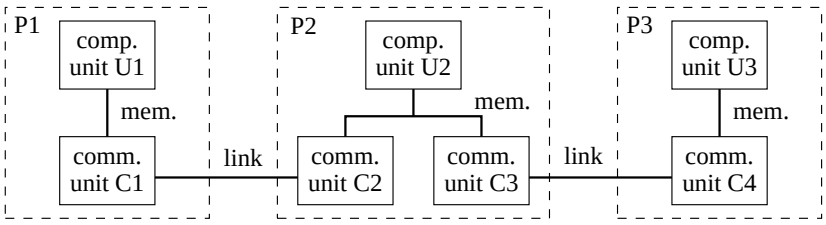

Figure 8: An example of architecture graph: three processors and two point-to-point links.

The `comms` are scheduled onto the communication units. All the communication units connected to the same communication link share this link, just like a shared memory. It ensures a hardware synchronization between the send operation and the receive operation. The link also has an arbiter which makes sure that the `comms` are executed sequentially over the medium.

4.4 Implementation Model

The implementation consists in reducing the potential parallelism of the algorithm graph into the available parallelism of the architecture graph. The graph models used for the algorithm and the architecture specifications lead to a formalization of the implementation in terms of three graphs transformations:

1. Distribution of the computation operations onto the computation units: each `comp`, `mem`, and `extio` is assigned to the computation unit of one processor, according to the distribution constraints. This distribution also transforms each inter-processor data-dependency (due to the distribution of `comps`/`mems`/`extios`) into a vertex, called `comm`, linked to the source operation with an input edge and to the destination operation with an output edge.

2. Distribution of the inter-processor data-dependencies generated by the first transformation: each `comm` is assigned to the set of communication units which are bound to the link connecting the processors executing the source and destination operations. They cooperate to transfer data between the local memories of their respective processors.

3. Scheduling of the `comps/mems/extios` (resp. `comms`) which have been assigned to a computation unit (resp. communication unit) during the first (resp. second) transformation. Each schedule is completely static.

The `comms` are thus totally ordered over each communication link. Provided that the network preserves the integrity and the ordering of messages, this total order of the `comms` guarantees that data will be transmitted correctly between processors. The obtained schedule also guarantees a deadlock free execution.

According to the architecture model, the communication scheme is the one presented in Figure 9(a), but for the sake of simplicity we shorten it to Figure 9(b). In this figure, the communication link is the one between processors P1 and P2 in Figure 8. A and B are two `comps` respectively scheduled on the computation unit of P1 and P2; they are represented by white boxes whose height is proportional to their respective execution time on their assigned processor. A and B are tied by an inter-processor dependency that is scheduled to the communication link between P1 and P2; it is represented by a gray box whose height is proportional to its communication time.

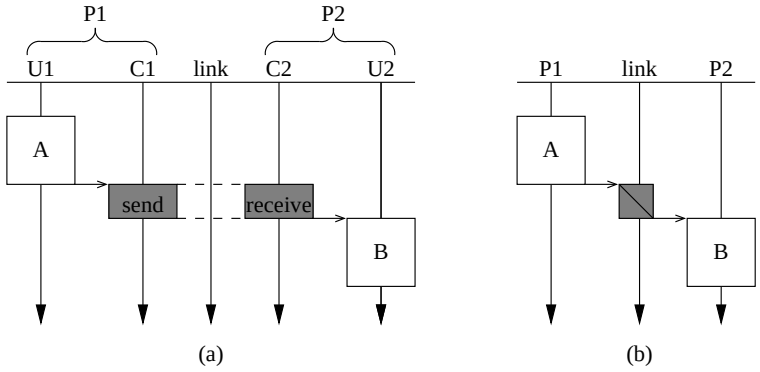

Figure 9: Send-receive over a communication link.

With these graphs transformations, a real-time distributed executive able to execute the algorithm on the multiprocessor [43] is automatically generated. Several graph transformations may be obtained for a given algorithm/architecture graphs pair. The real-time constraints require to select the transformation which minimizes the response time of the transformed graph. The response time is the critical path of the graph, computed from the execution durations of the operations, including `comms`. This is an optimization problem and, as other resource allocation optimization problems, it is known to be NP-complete.

Several heuristics have been proposed in [16, 48]. They use a cost function called *schedule pressure*. They build step by step the graph transformation by composing elementary graphs transformations. Each step transformation consists in assigning and scheduling a selected candidate computation operation (resp. `comm`) on a selected computation unit (resp. communication unit).

In the sequel, since there is only one computation unit in a processor, we will use "processor" instead of "computation unit" to lighten explanations.

5 Assumptions and Design Choices

Problem: *Given an algorithm, a distributed architecture, and some real-time constraints, produce automatically fault-tolerant distributed code for the given algorithm onto the given architecture, satisfying the given real-time constraints.*

We take advantage of AAA to reduce this problem to a scheduling problem; the code generation will then be achieved exactly like in the non fault-tolerant case (see Section 4.1).

The originality of our approach is that we do not add extra hardware to our distributed architecture in order to support failures. Rather, we use the existing parallelism of the given architecture. This approach falls in the class of *software implemented fault-tolerance*. The drawback is that only certain types of failures and means can be considered, and that a solution is not always possible, for instance if the available computing power does not allow to meet the real-time constraints, or if some operation can be executed by only one processor that can fail.

In this section, we first define precisely the kind of failures and means considered. Then we present the consequences of the AAA method on our fault-tolerance choices.

5.1 Classes of Failures

Choice: *The classes of failures considered are: accidental, physical, internal, operational, and permanent*

1. We assume that our systems are not submitted to exterior, malevolently attacks, so we do not treat intentional failures.
2. We restrict ourselves to failures caused by physical phenomena (e.g., a power glitch), and not human-made failures (e.g., an incorrect user command).
3. Although the external failures (resulting from interference or interaction with the physical environment) can appear in our systems, we abstract this kind of failures to their effects on the internal parts of the systems.
4. Since design faults may be eliminated through the formal validation of the algorithm graph (see Section 1.2), we consider only operational failures.

5. Finally, we consider only architectures where processors have a fail-stop behavior [39].

5.2 Means for Fault-Tolerance

Choice: *Fault-tolerance is achieved by using error compensation.*

Error compensation seems to be the best solution for systems having strong timing constraints [14]. Instead, error recovery usually implies a time consuming procedure in order to recover a correct state, be it forward or backward. Also, in the presence of permanent failures, the treatment should use reconfiguration to provide the same service, which is usually time consuming. Moreover, unlike error recovery which needs additional specifications describing the algorithms to be executed after an error occurs, error compensation allows the code generation for fault-tolerant implementations of a given algorithm in a fully automatic way.

5.3 Kind of Redundancy

Choice: *In the solutions proposed here, only software and time redundancy are used. We choose not to consider hardware redundancy (i.e., adding extra hardware).*

1. Unlike software and time redundancy, hardware redundancy can not be implemented completely automatically since the user must specify explicitly, e.g., the replication degree of each hardware resource, the voting mechanism, and so on.

2. One of the main goals in the design of embedded systems is the minimization of hardware resources used by the system, and hardware redundancy goes against this goal. That is, we do not want to add extra hardware to the given architecture.

3. Our target architectures *already* exhibit some redundancy since they are distributed ones and the computing resources are not fully exploited by the application. So we choose to exploit this *existing* parallelism by replicating the software modules and executing the replicas on distinct processors, in order to cope with eventual processor failures.

Compared to time redundancy, software redundancy minimizes the response time of the fault-tolerant system. For this reason, fault-tolerance in critical real-time systems is usually achieved with some sort of software redundancy, e.g., *active replicas* [17] or *state machine approach* [41]. The drawback is that software redundancy introduces computation and communication overhead in absence of failures.

At the opposite, time redundancy reduces the overhead of the system in the absence of failures, but it does not ensure a minimal response time in the presence of failures. A tradeoff between these two kinds of redundancy should be found in order to obtain good performances for the obtained fault-tolerant system in both cases.

5.4 AAA Algorithm Model

Assumption: *For each operation of the algorithm graph, the user assigns a set of processors of the architecture graph that can execute the operation.*

In doing this, the user actually specifies the distribution constraints for his algorithm onto his architecture. The assigned sets must be such that:

1. Since a `comp` is safe (see Section 4.2 Item 1), it can be replicated (software and/or time) at will: hence the assigned set can include all the processors.
2. Since a `mem` is memory-safe (see Section 4.2 Item 2), each replica must be initialized with the same initial value, since the output is computed deterministically from the initial value and the input values: the assigned set can include all the processors with a RAM.
3. Since `extios` are unsafe (see Section 4.2 Item 3) and associated to the sensors/actuators they control, their replication is related to the replication of the hardware controlled: the assigned set can only include the processors which control the corresponding sensor/actuator.

For instance, the distribution constraints for the algorithmic graph of Figure 7 and the architecture graph of Figure 8 can be given by the two following tables of time units:

		operation						
	time units	I	A	B	C	D	E	O
proc.	P1	1	2	3	2	3	1	1.5
	P2	1	2	1.5	3	1	1	1.5
	P3	∞	2	1.5	1	1	1	∞

		data-dependency							
	time units	I $\triangleright$ A	A $\triangleright$ B	A $\triangleright$ C	A $\triangleright$ D	B $\triangleright$ E	C $\triangleright$ E	D $\triangleright$ E	E $\triangleright$ O
link	C1 – C2	1.25	0.5	0.5	1	0.5	0.6	0.8	1
	C3 – C4	1.25	0.5	0.5	1	0.5	0.6	0.8	1

In this particular case, the time needed for communicating a given data-dependency is the same on both communication links, but in the general case it can be distinct. Also it takes more time to communicate the data-dependency I $\triangleright$ A than A $\triangleright$ B simply because there are more data to transmit.

Besides, the data-flow dependencies indicate when a `comp`/`mem`/`extio` may be executed. If any processor p is affected by a permanent failure, the operations whose inputs are produced by p, whatever the processor they are assigned to, cannot be executed. Therefore we use the data-flow dependencies to ensure that all the inputs are provided to each operation executed on a non faulty processor, possibly with replication or delay.

5.5 AAA Architecture Model

Choice: *Only the failures of processors are managed. Status flags are associated to communication units and propagated to maintain a knowledge of the failures in the architecture.*

1. Fault-tolerance can be achieved only if the target architecture specified by the user has some existing redundancy, i.e., at least two processors. Moreover, the degree of parallelism of the given architecture should be greater than the number of failures that the user wants to be supported. Otherwise, the user is informed about the impossibility to provide fault-tolerance.

2. We choose not to consider communication link failures, for several reasons. First, this problem can be handled at the network level. Indeed, industrial buses used in automotive and avionic (CAN, VAN ...) provide fault-tolerance capabilities, e.g., one the the three wires of a CAN bus may be cut without loosing messages. Then, there are two approaches for achieving fault-tolerant routing in a network: static routing (see for instance [9]) and adaptative routing (see for instance [10]). Several studies are identified cases when adaptative routing outperforms static routing [6, 24, 35], but in the general case, we think that static routing should be preferred because it allows to compute a worst case upper-bound on each communication.

   Second, if we want to take both link and processor failures into account in our approach, then we will face the problem that sometimes it is difficult to distinguish them: we would then have to run special detection procedures, which would lower the overall performance of the system. Finally, according to our architecture model (see Section 4.3), a failure of the processor means a failure of both the computation unit and all the communications units of that processor; this is for instance the case of the MOTOROLA `MCP 555` processor which includes two CAN bus interfaces.

A failure is detected after the expiration of the timeout associated to a `comm`. In Figure 8, suppose that P3 crashes. This can be detected only by P2 during a `comm`, after the expiration of the associated timeout. To avoid further `comms` with faulty units, each processor maintains an array of "fail" flags giving the state of each communication unit in the system. In this example, the flag of C4 on P2 is set. The problem of detection becomes more complex when a `comm` is routed over several communication units. Still in Figure 8, suppose that P1 awaits an input from P3. This `comm` is routed over P2. If P3 is faulty, P2 detects this failure, sets its C4 fail flag, and signals to P1 the end of the communication. Then, to propagate this information to P1, we propose the send/receive procedures of Figure 10.

```
procedure receive (from unit j, to unit i)      // unit = communication unit
begin
  if not fail [j] then
    wait t while message not arrived            // waiting a message from the unit j
    if message not arrived then                 // timeout expires
      fail [j] := true
      message := "unit j is faulty"
    end if
  else
    message := "unit j is faulty"
  end if
  signal message available                      // the message can be used by another unit
end receive

procedure send (from unit i, to unit k)
begin
  wait message available
  send message to unit k
end send
```

Figure 10: Failure propagation in the routed communication (j, i, k).

5.6 Specific Problem

Specific problem: *Given an algorithm specified as a data-flow graph, a distributed architecture specified as a graph, some distribution constraints, some real-time constraints, and a number K, produce automatically a distributed schedule for the algorithm onto the architecture w.r.t. the distribution constraints, satisfying the real-time constraints, and tolerant to K permanent fail-stop processor failures, by means of error compensation, using software and/or time redundancy.*

Remember that a solution is achievable only if the number of processors given in the architecture specification is sufficient: it must be greater than K and it must allow the obtained schedule to meet the desired real-time constraints.

In the following sections we propose two solutions, which both consist of a new scheduling heuristic to be used in the SynDEx tool; the remaining of the distribution method presented in Section 4.4 is unmodified. The first solution uses the software redundancy of `comps` and the time redundancy of `comms`. The second solution uses the software redundancy of `comps` and `comms`.

The performances of the solutions are evaluated according to the following criteria:

1. The computation and communication overhead introduced by fault-tolerance: this requires to compare the obtained fault-tolerant schedule with the non fault-tolerant one given by SynDEx.

2. The capability to support several failures within the same iteration.

3. The timing performances of the faulty system, i.e., a system presenting at least one failure. Here we distinguish the iteration in which the failure(s) actually occurs and the subsequent iterations where one or more processors are faulty but no new failure occurs. We call an iteration in which at least one failure occurs a *transient iteration.*

4. And finally the appropriateness to different kinds of architecture. As we will see, the first solution is suited to multi-point links architectures, while the second solution is suited to point-to-point links architectures.

6 First Solution

6.1 Principle

The first solution uses the active redundancy of `comps/mems/extios` and the time redundancy of `comms`. Each operation o of the algorithm graph is replicated on $K+1$ different processors of the architecture graph, where K is the number of permanent failures to be supported. Among these $K+1$ replicas, the one whose completion date is the earliest is designated to be the *main replica.* Without entering into details, completion dates are computed according to the execution duration of each operation and each data-dependency given by the user in the distribution constraints. The main replica sends its results to each processor executing one replica of each successor operation of o, except the processors already executing another replica of o (in which case it is an intra-procesor communication). The processor executing the main replica is called the *main processor* of o. The remaining K processors executing o, called *backup processors*, execute o and watch on the response of the main processor. If the main processor does not respond on time, it is considered as faulty, and another processor executing a replica of o sends o's results to the successor operations.

This solution raises the following problem: *What kind of communication mechanism should*

2. *How are computed the timeouts associated to the communications?* On the one hand, the timeouts should allow an accurate detection of failures in order to avoid unnecessary elections of the main processor and multiple sendings of messages. On the other hand, if the timeouts are too large, the accumulation of these timeouts in the case of multiple failures may lead to the missing of the real-time constraints. We choose to compute a given timeout as the worst case upper-bound of the message transmission delay. This upper-bound is computed from the characteristics of the communication network (see Section 4.3). Since we have discarded timing failures (see Section 5.1), this is the least possible value avoiding multiple sendings of messages.

3. *What are the consequences of making failure-detection mistakes?* As we have said in Item 2 above, this problem can be avoided by computing a timeout as the worst case upper-bound of the message transmission delay. Now if the resulting schedule does not meet the desired real-time constraints, one may wish to relax this computing rule for the timeouts. This problem is related to the handling of intermittent failures, i.e., fail-silent behaviors [11]. If the given target architecture uses a single multi-point link, then the outputs of all operations will be broadcasted over this multi-point link to all the processors, including the faulty ones. Then a processor that was previously marked as being faulty and that is now running (either because of an intermittent failure, or because it was a detection mistake) can resume its computations and output its results on the bus. Therefore, if after a failure detection the healthy processors continue to scan the bus, they will detect that the faulty processor is now running and they will update their array of faulty processors accordingly. Consequently, this scheme, which requires to modify the *send* and *receive* procedures of Figure 10, could allow us to treat intermittent fail-silent behaviors as well and permanent fail-stop.

4. *According to which criterion is the main processor selected?* This criterion must be applied initially and each time the backup processors elect a new main processor following a failure. We choose the processor which finishes first the execution of the replica operation. For each operation, we thus compute from the static schedule a total order of all the backup processors. This total order is known by each processor, so the result of the election is the same for everybody.

6.2 Scheduling Heuristic

The heuristic implementing this solution is a greedy list scheduling, whose algorithm is given on Figure 11. It is adapted from [16, 48].

S0. Initialize the lists of candidate and scheduled operations:
$O_{sched}(0) = \emptyset, \quad O_{cand}(0) = \{o \in O \mid pred(o) \subseteq O_{sched}(0)\}$
Sn. **while** not empty $O_{cand}(n)$ **do**
mSn.1 Compute the scheduling pressure for each operation of $O_{cand}(n)$ and keep the first $K+1$ results for each operation:
$\forall o_i \in O_{cand}(n), \quad \cup_{l=1}^{l=K+1} \{\sigma^{opt}(n)(o_i, p_{i_l})\} = min_{p_j \in P}^{K+1} \{\sigma(n)(o_i, p_j)\}$
$P^{(K+1)}(o_i) = \cup_{l=1}^{l=K+1} \{p_{i_l}\}$
mSn.2 Select the best candidate operation:
$\sigma^{best}(n)(o) = \max_{o_i \in O_{impl}(n)} \cup_{l=1}^{l=K+1} \{\sigma^{opt}(n)(o_i, p_{i_l})\}$
mSn.3 The operation o is implemented on the first $K+1$ processors computed at mSn.1 as well as the communications implied by these assignments. The main processor is $p_m \in P^{(K+1)}(o)$ such that
$S(n)(o, p_m) + \Delta(o, p_m) = min_{p_l \in P^{(K+1)}(o)} \{(S(n)(o, p_l) + \Delta(o, p_l))\}$
mSn.4 Update the lists of candidate and scheduled operations:
$O_{sched}(n) = O_{sched}(n-1) \cup \{o\}$
$O_{cand}(n+1) = O_{cand}(n) - \{o\} \cup \{o' \in succ(o) \mid pred(o') \subseteq O_{sched}(n)\}$
end while

Figure 11: Scheduling heuristic for the first solution.

At the n-th step ($n \geq 1$), a list of candidate operations $O_{cand}(n)$ is built from the algorithm graph vertices. An operation is said to be candidate if all its predecessors are already scheduled, i.e., assigned to a processor. The list of scheduled operations $O_{sched}()$ keeps at each step the operations that have already been scheduled. Initially, $O_{sched}(0)$ is empty. By using a cost function called *schedule pressure*, one operation of the list $O_{cand}(n)$ is selected to be scheduled at the step n.

The schedule pressure is computed in two phases. The first one is done before the scheduling heuristics. It computes, using the algorithm graph and the characteristics lookup table, the critical (maximal) path of the algorithm (noted R) and, for each operation o_i the maximal date at which o_i may end (noted $E(o_i)$) computed from the end of the critical path. The second phase takes place at each step of the scheduling algorithm. It computes for an operation $o_i \in O_{cand}(n)$ and a processor $p_j \in P$ (P is the processor's set) the "earliest start date from start" (noted $S(n)(o_i, p_j)$), i.e., the execution time of the part of the distributed algorithm scheduled at the step $n-1$. $S(n)(o_i, p_j))$ takes into account the communication times between o_i and the main processor of its predecessors and successors, when they differ from p_j. This choice improves the execution time for the system without failures, but may give longer execution times in the faulty cases.

$$\sigma(n)(o_i, p_j) = S(n)(o_i, p_j) + \Delta(o_i, p_j) + E(o_i) - R$$

where $\Delta(o_i, p_j)$ is the execution duration of o_i on processor p_j; this value is given in p_j's characteristics lookup table. The schedule presume measures how much the scheduling of the operation lengthens the critical path of the algorithm. Therefore it introduces a priority between the operations to be scheduled.

The selected operation is obtained as follows. First, in the micro-step mSn.1, we compute for each candidate operation o_i the set $P^{(K+1)}(o_i)$ of the first $K+1$ execution units

minimizing the schedule pressure. The first $K+1$ minimal schedule presumes for o_i, called $\sigma^{opt}(n)(o_i, p_{i_l})$, give the processors p_{i_l} from which the set $P^{(K+1)}(o_i)$ is computed (the superscript $(K+1)$ for P indicates its cardinality). We thus obtain for each operation $K+1$ pairs $\langle$operation, processor$\rangle$. Then, in the micro-step mSn.2, the operation belonging to the couple having the greatest schedule pressure is selected. If there exists more than one couple having the greatest schedule pressure, one is randomly chosen among them.

The implementation of the selected operation at the micro-step mSn.3 implies the choice of a main processor for the operation and the computation of timeouts for the communication operations implemented on the backup processors. We select as main processor the processor of the set $P^{(K+1)}(o)$ (the first $K+1$ processors computed for o at the micro-step mSn.1) which finishes first the execution of the operation, i.e., the one which minimizes the sum $S(n)(o, p_l) + \Delta(o, p_l)$. The K backup processors are ordered according to the increasing order of the sum $S(n)(o, p_l) + \Delta(o, p_l)$, i.e., to the increasing order of the completion date of the operation o.

6.3 Communication Procedure

For an architecture model using one multi-point link, we present in Figure 12 a solution for the implementation of the communication operation executed by the communication unit of the i-th backup processor of the operation o. We suppose that the operation o is replicated on processors $p_0, \ldots, p_K$, where p_0 is the main processor and $p_1, \ldots, p_K$ is the ordered list of backup processors. The durations of timeouts $t_k^{(i)}$, where $k \in \{0, \ldots, i-1\}$, are computed statically from the durations of inter-processor communications $\Delta(p_i, p_k)$ and the durations of execution of o on each processor. The general formulas are:

$$\begin{aligned} t_i^{(i)} &= S(o, p_i) + \Delta(o, p_i) \\ t_k^{(i)} &= \max(t_0^{(k)}, \ldots, t_k^{(k)}) + \Delta(p_k, p_i) \end{aligned}$$

```
procedure OpComm (from p_i)
begin
  m := 0    // the main processor is initially p_0
  while m < i do
    if not fail [unit of p_m] then
      wait t_m^(i) while message not arrived
      if message arrived then break
      else fail [unit of p_m] := true; m := m + 1
      end if
    end if
  end while
  if m = i then
    wait while o is not finished
    send the result of o to the units of successors and remainder backup processors
  end if
end OpComm
```

Figure 12: Communication operation for the ith backup processor.

6.4 Fault-Tolerance Overhead

Now let us study the optimality of the number of inter-processor communications generated in the fault-tolerant schedule by our heuristic. Firstly, each operation of the algorithm graph is replicated $K+1$ times, but each replica only receives its inputs only once, namely from the main replica of the predecessor operation. Therefore each data-dependency of the algorithm graph leads to at most $K+1$ inter-processor communications. Indeed, when the two operations linked by the data-dependency are scheduled on the same processor, we have an intra-processor communication. In this sense we say that the number of messages in the fault-tolerant schedule is minimal.

Secondly, when a failure occurs, we claim that the number of inter-processor communications in the resulting schedule is less than in the initial schedule. Remember that only the main replicas send their results through inter-processor communications. Let us call p the faulty processor. Consider an operation o with n successor operations in the algorithm graph, and whose main replica is assigned to p. In the initial schedule, this main replica of o sends its results to $(K+1) \times n$ replicas. Among these, a number k_{intra} are intra-processor communications because the corresponding replica is also assigned to p. The number k_{inter} of inter-processor communications actually sent by the main replica of o is such that:

$$k_{inter} + k_{intra} = (K+1) \times n$$

Now since o's main replica is assigned to p which fails, a new main processor will be chosen for o. The previous k_{intra} messages are no longer necessary since they concern operations assigned to p which is faulty. Among the remaining k_{inter} messages, some more are intra-processor because they concern operations assigned to the new main processor of o. As a result, the new number of inter-processor communications needed to send the results of o to all the replicas of all its successor operations is less than in the initial schedule.

6.5 An Example

We apply our first heuristic to the example of Figure 13.

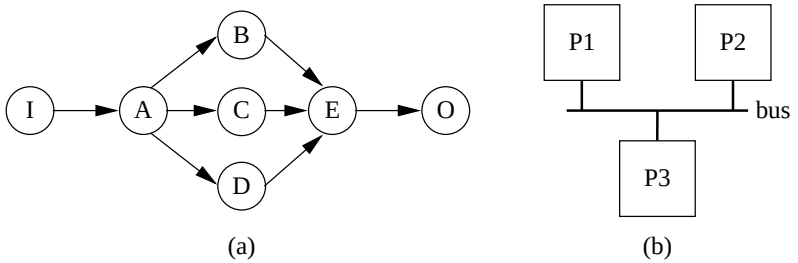

Figure 13: First example: (a) algorithm and (b) encapsulated architecture graph.

The user requires to tolerate one permanent processor failure. The execution characteristics of each `comp`/`mem`/`extio` and `comm` are specified by the two following tables of time units:

		operation						
	time units	I	A	B	C	D	E	O
proc.	P1	1	2	3	2	3	1	1.5
	P2	1	2	1.5	3	1	1	1.5
	P3	∞	2	1.5	1	1	1	∞

	data-dependency			
time units	I $\triangleright$ A	A $\triangleright$ B	A $\triangleright$ C	A $\triangleright$ D

At the next step, operation C is assigned to processors P1 and P3, P1 being the main processor for this operation. We obtain the temporary schedule of Figure 16.

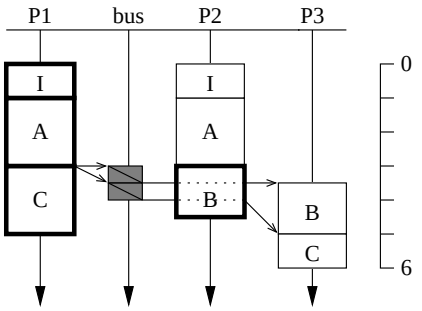

Figure 16: Temporary schedule: operations I, A, B, and C are scheduled.

At the end of our heuristic, we obtain the final schedule presented in Figure 17. Each operation of the algorithm graph is replicated twice and these replicas are assigned to different processors, the main replica being represented by a thicker white box.

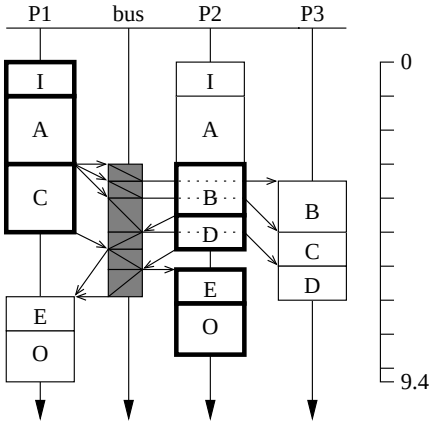

Figure 17: Final fault-tolerant schedule for the first example.

The timing diagrams of Figure 18 show the executions when P2 crashes: (a) is the transient schedule while (b) is the permanent subsequent schedule.

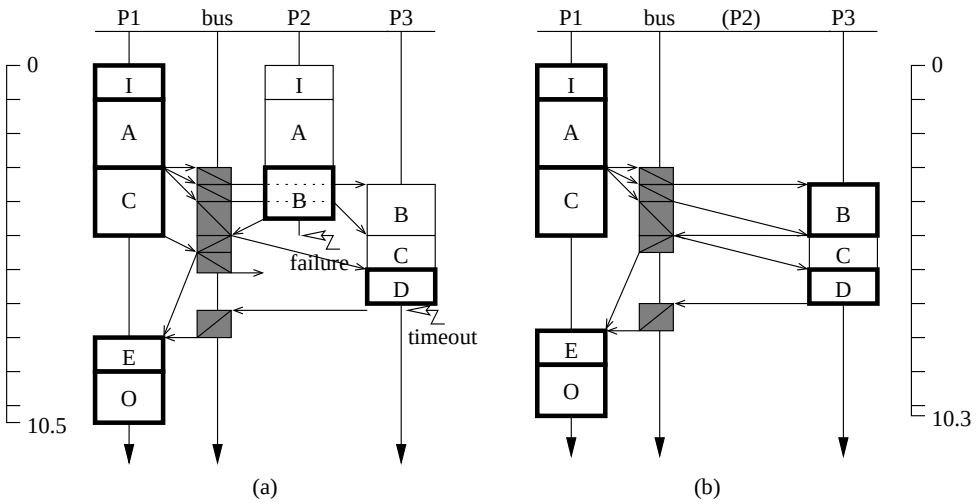

Figure 18: Timed execution of the first example when P2 crashes: (a) is the transient schedule; (b) is the permanent schedule.

- Like we have said, the number of communications does not increase when a failure occurs.
- The diagram (a) shows that, during the transient iteration following a failure, the response time is increased by the waiting delay of the response from the faulty processor.

6.6 Analysis of the First Example

To evaluate the overhead introduced by the fault-tolerant iteration for [illegible] a) shows that [illegible] shows that [illegible] shows tha

In this particular case, the overhead is therefore $9.4 - 8.6 = 0.8$. With a bigger example, the overhead would probably be larger. Part of this overhead is due to the extra computations (for the replica operations), and part is due to the extra communications (for sending their input data to all the replica operations instead of only one successor operation). However, the replica operations do not send their result until a failure occurs. Therefore the communication overhead is minimal.

When a failure occurs, extra communication can take place. This is the case of our example when P2 crashes (see Figure 18). The response time of the faulty system is greater than in absence of failures, since some overtime is necessary to detect the failure of the main processors. For the same reason, the arrival of several failures at the same time is not well supported since there is a risk that the sum of timeouts amassed overtakes other timeouts. As already said, the current solution is appropriate to an architecture where the communication units are bound by a unique multi-point link. With point-to-point links, the detection of the failure for the main processor requires an agreement protocol.

7 Second Solution

7.1 Principle

For architectures using point-to-point links, we propose a second solution, based on the active redundancy of both `comps` and `comms`. As in the first solution, each `comp` is replicated $K+1$ times, on $K+1$ different processors. The difference is that these $K+1$ replicas send their results in parallel to all the replicas of all the successor operations in the data-flow graph. Therefore, each operation will receive its set of inputs $K+1$ times, through intra and inter-processor communications; as soon as it receives the first set, the operation is executed and ignores the later inputs. However, in some case, the replica of an operation will only receive some of its inputs once, through an intra-processor communication. For the sake of simplicity, suppose we have an operation o with only one input produced by its predecessor o'. Consider the replica of o which is assigned to processor p. If one of the replicas of o' is also assigned to p, then the `comm` from o' to o will not be replicated and will therefore be implemented as a single intra-processor communication. Indeed, the replicas of this `comm` would only be used if p would fail, but in this case the replica of o assigned to p would not need this input. In the reverse case, i.e., if all the replicas of o' are assigned to processors distinct from p, then the `comm` from o' to o will be replicated $K+1$ times.

The advantage of the second solution is that no timeouts are computed, the propagation of the state of faulty communication units is not necessary, and the response time at the occurrence of a failure is minimal. On the other hand, the communication overhead is greater than with the first solution since there are more inter-processor communications. In particular, if the architecture uses a multi-point link, the communications sent over this medium are serialized, so the overhead introduced is greater than in the first solution.

7.2 Scheduling Heuristic

As in the first solution, the heuristic proposed for the second solution implements a greedy list scheduling. The algorithm is given in Figure 20, and it follows closely the heuristic presented for the first solution in Figure 11. The main differences concern the computation of the scheduling pressure at the micro-step mSn.1 and the implementation of the operation at the micro-step mSn.4.

S0. Initialize the lists of candidate and scheduled operations:
$O_{sched}(0) = \emptyset, \quad O_{cand}(0) = \{o \in O \mid pred(o) = O_{sched}(0)\}$
Sn. **while** not empty $O_{cand}(n)$ **do**
mSn.1 Compute the scheduling pressure for each operation of $O_{cand}(n)$ and keep the first $K+1$ results for each operation:
$\forall o_i \in O_{cand}(n), \quad \cup_{l=1}^{l=K+1} \{\sigma^{opt}(n)(o_i, p_{i_l})\} = min_{p_j \in P}^{K+1} \{\sigma(n)(o_i, p_j)\}$
$P^{(K+1)}(o_i) = \cup_{l=1}^{l=K+1} \{p_{i_l}\}$
mSn.2 Select the best candidate operation:
$\sigma^{best}(n)(o) = \max_{o_i \in O_{impl}(n)} \cup_{l=1}^{l=K+1} \{\sigma^{opt}(n)(o_i, p_{i_l})\}$
mSn.3 The operation o is implemented on the first $K+1$ processors computed at mSn.1 as well as the communications implied by these assignments.
mSn.4 Update the lists of candidate and scheduled operations:
$O_{sched}(n) = O_{sched}(n-1) \cup \{o\}$
$O_{cand}(n+1) = O_{cand}(n) - \{o\} \cup \{o' \in succ(o) \mid pred(o') \subseteq O_{sched}(n)\}$
end while

Figure 20: Scheduling heuristic for the second solution.

The schedule pressure σ is computed by:

$$\sigma(n)(o_i, p_j) = S(n)(o_i, p_j) + \Delta(o_i, p_j) + E(o_i) - R$$

where the communication time computed for a predecessor (in $S(n)(o_i, p_j)$) is the minimum of the communication times with each replica of the predecessor. The selected operation is implemented at micro-step mSn.4 on the $K+1$ processors computed at micro-step mSn.1, as well as the communication operations implied by these schedulings. As we have said, for each couple (predecessor, operation replica), `comms` are added if and only if all the replicas of the predecessor are on different processors. If it is not the case, i.e., if there exists a replica of the predecessor on the same processor, no `comm` is added.

7.3 An Example

We apply our second heuristic to the example of Figure 21.

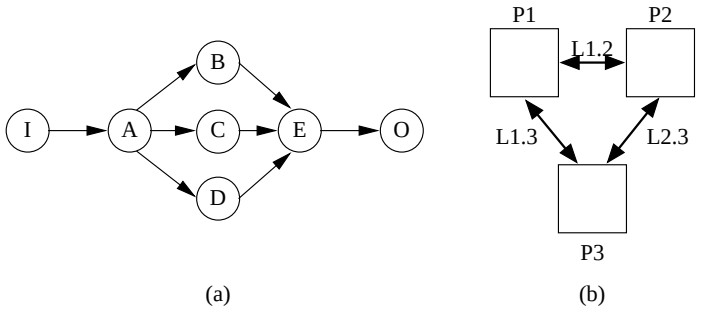

Figure 21: Second example: (a) algorithm and (b) encapsulated architecture graph.

This is the same algorithm graph as the example of Section 6.5 and the architecture also has three processors; but it uses three point-to-point links instead of a bus. Again, the user requires to tolerate one permanent failure of a processor. The execution characteristics of each `comp`/`mem`/`extio` and `comm` are specified by the following tables. The execution times are the same as in Section 6.5:

		operation						
	time units	I	A	B	C	D	E	O
proc.	P1	1	2	3	2	3	1	1.5
	P2	1	2	1.5	3	1	1	1.5
	P3	∞	2	1.5	1	1	1	∞

	data-dependency							
time units	I$\triangleright$A	A$\triangleright$B	A$\triangleright$C	A$\triangleright$D	B$\triangleright$E	C$\triangleright$E	D$\triangleright$E	E$\triangleright$O
L1.2	1.25	0.5	0.5	1	0.5	0.6		

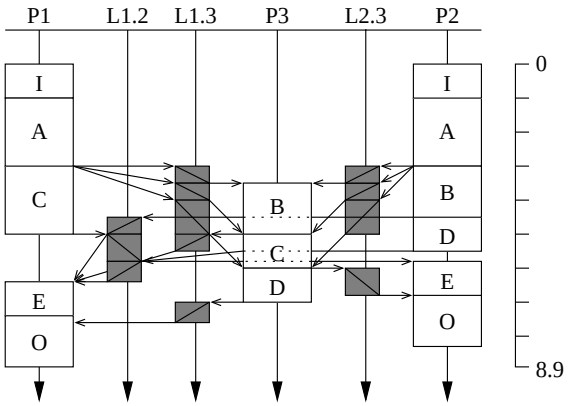

Figure 22: Final fault-tolerant schedule for the second example.

This timing diagram shows that some communications are not useful in the absence of failures. For example, the communication of the result of the operation (D, P3) to the operation (E, P1) is not used since the result sent by (D, P2) arrives first. However, these communications may become useful during a faulty execution.

The diagram of Figure 23 shows the transient schedule when P2 crashes after executing `comp` A.

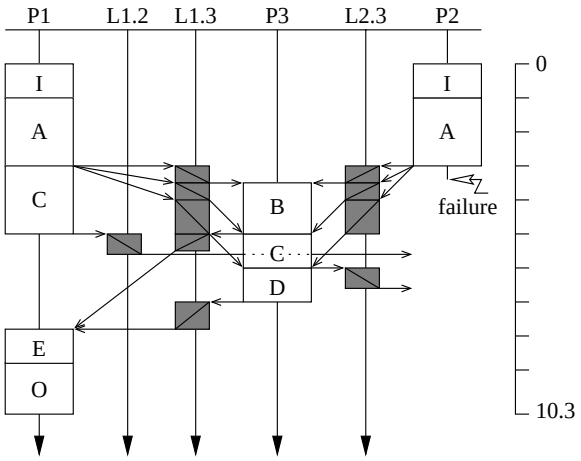

Figure 23: Timed execution of the second example when P2 crashes.

As expected, the data sent by all the `comms` toward the faulty processor P2 are discarded. The schedule corresponding to the subsequent iterations is the same except that the useless `comms` have disappeared.

7.4 Analysis of the Second Example

The overhead introduced by the fault-tolerance is maximal in the sense that both computations and communications are executed in parallel. Figure 24 shows the timed schedule obtained with SynDEx in the non fault-tolerant case.

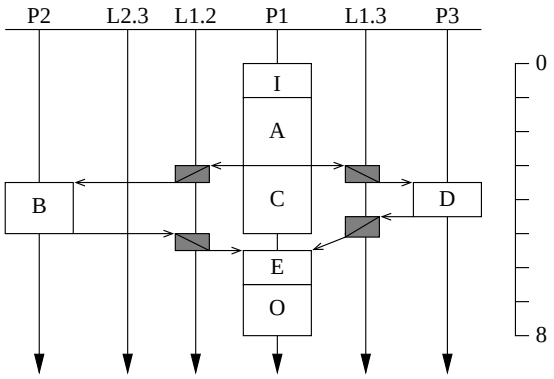

Figure 24: Non fault-tolerant schedule for the second example.

In this particular case, the overhead is therefore $8.9 - 8 = 0.9$. In the fault-tolerant schedule, some communications take place although they are not necessary. On the other hand, the response time of the faulty system is minimized, since results are sent without waiting for the timeouts (see Figure 23). For the same reason, the system supports the arrival of several failures at the same time since there is no risks that the sum of pending timeouts overtakes the desired real-time constraints.

This solution is appropriate to an architecture where the communication units are bound by several point-to-point links which allow parallel communications. For multi-point links, the overhead introduced by the replication of communication operations may be important because of the serialization of `comms` on a single link.

The drawback of using point-to-point links is that intermittent failures and detection mistakes cannot be recovered. Indeed, when a processor is detected to be faulty, the other healthy processors will update their array of faulty processors, and will not send data anymore during the subsequent iterations. So even if this faulty processors comes back to life, either after an intermittent failure or because it was a detection mistake, it will not receive any inputs and will not be able to perform any computation. Therefore in the subsequent iterations it will fail to send any data on its communication links, and the other healthy processors will never be able to detect that it came back to life.

8 Conclusion

The literature about fault-tolerance of distributed and/or embedded real-time systems is very rich. Yet, there are few attempts to combine fault-tolerance and automatic generation

of distributed code for embedded systems. We have presented two solutions of software implemented fault-tolerance, which adapt the automatic code distribution algorithm of the "Algorithm Architecture Adequation" method (AAA). Basically, AAA takes as input a description of the algorithm to be distributed and a description of the target architecture. AAA first produces a static distributed schedule of a given algorithm onto a given distributed architecture, and then it generates a real-time distributed executive implementing this schedule.

Since we are dealing with embedded systems, we do not want to add redundant hardware. Rather, we choose to take advantage of the existing parallelism offered by the target distributed architecture. Also we consider only processor failures and assume they have a fail-stop behavior.

We are therefore given an algorithm specification, an architecture specification, some real-time constraints, and a number K of processor failures to be tolerated. Taking advantage of AAA, we have proposed two new scheduling heuristics that produce automatically a static distributed fault-tolerant schedule of the given algorithm onto the given distributed architecture.

Our first solution is based on the active redundancy of the computation operations and on the time redundancy of the communications. A replicated operation only sends its result, after some timeout, when the main processor running the same operation fails. The implementation uses a scheduling heuristic for optimizing the critical path of the obtained distributed algorithm. It is best suited to architectures with multi-point communication links, and the communication overhead is minimal; the occurrence of several failures in a row is not well supported. We have finally shown that if the given target architecture uses a single multi-point link, then intermittent fail-silent processor failures can also be treated.

Our second solution replicates both the computation operations and the communications. Here, all replicated operations send their results but only the one which is received first by the destination processor is used; the other results are discarded. Again the implementation is based on a scheduling heuristic. It is best suited to architectures with point-to-point links, and the communication overhead is more important than with the first solution; however, several several failures can be supported in a row.

Both solutions can fail, either if the real-time constraints can't be satisfied by the obtained distributed fault-tolerant schedule, or if less than K processor failures can be tolerated. This can happen if the intrinsic parallelism offered by the target architecture is not sufficient.

Finally, both solutions can only tolerate processor failures. We are currently working on new solutions to tolerate also the communication link failures. Also, our method is being experimented on an electric autonomous vehicle, the Cycab [42, 2], which a 5 processors distributed architecture and a CAN bus.

Acknowledgments

The authors would like to thank Cătălin Dima, Thierry Grandpierre, Claudio Pinello, and David Powell for their helpful suggestions.

References

[1] T. Anderson and J.C. Knight. A framework for software fault-tolerance in real-time systems. *IEEE Transactions on Software Engineering*, 9(3):355–364, May 1983.

[2] G. Baille, P. Garnier, H. Mathieu, and R. Pissard-Gibollet. The INRIA Rh ne-Alpes CYCAB. Technical Report n^o 0229, INRIA, Rocquencourt, France, April 1999.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] A.A. Bertossi and L.V. Mancini. Scheduling algorithms for fault-tolerance in hard-real-time systems. *Real-Time Systems Journal*, 7(3):229–245, 1994.

[5] A. Bouali. XEVE: An ESTEREL verification environement. In *International Conference on Computer Aided Verification, CAV'98*, LNCS, Vancouver, Canada, June 1998. Springer-Verlag.

[6] W. Chou, A.W. Bragg, and A.A. Nilsson. The need for adaptative routing in the chaotic and unbalanced traffic environment. *IEEE Transactions on Communications*, COM-29:481–490, April 1991.

[7] J.-Y. Chung, J.W.S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, September 1990.

[8] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991. Available at `ftp://ftp.cs.ucsd.edu/pub/team/understandingftsystems.ps.Z`.

[9] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36:547–553, May 1987.

[10] J. Duato. A new theory of deadlock-free adaptive routing in wormhole networks. *IEEE Transactions on Parallel and Distributed Systems*, 4(12):1320–1331, December 1993.

[11] D. Powell et al. The Delta-4 approach to dependability in open distributed systems. In *Proceeding of the 18th IEEE International Symposium on Fault-Tolerant Computing, FTCS'88*, pages 246–251, Tokyo, Japan, June 1988. IEEE Computer Society Press.

[12] M.J. Flynn. Some computer organization and their effectiveness. *IEEE Transations on Computer*, pages 948–960, September 1972.

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

[14] S. Ghosh. *Guaranteeing Fault-Tolerance through Scheduling in Real-Time Systems.* PhD Thesis, University of Pittsburgh, 1996.

[15] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerant scheduling on a hard real-time multiprocessor system. In H.J. Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 775–783, Los Alamitos, CA, April 1994. IEEE Computer Society Press.

[16] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of 7th International Workshop on Hardware/Software Co-Design, CODES'99*, Rome, Italy, May 1999.

[17] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing, FTCS'96*. IEEE Computer Society Press, June 1996.

[18] M. Gupta and E. Schonberg. Static analysis to reduce synchronization cost in data-parallel programs. In *23rd ACM Symposium on Principles of Programming Languages*, pages 322–332, January 1996.

[19] N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Pub., 1993.

[20] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

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

[22] R. Bianchini Jr. and R.W. Buskens. An adaptive distributed system-level diagnosis algorithm and its implementation. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing, FTCS'91*, pages 222–229, June 1991.

[23] R. Bianchini Jr. and R.W. Buskens. Implementation of on-line distributed system-level diagnosis theory. *IEEE Transactions on Computers*, 41(5):616–626, May 1992.

[24] C.K. Kim and D.A. Reed. Adaptative packet routing in a hypercube. In *Conference on Hypercube Concurrent Computer Applications*, January 1988.

[25] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communication of the ACM*, 21(7):558–565, July 1978.

[26] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[27] B.W. Lampson. Atomic transactions. In B.W. Lampson, M. Paul, and H.J. Siegert, editors, *Distributed Systems-Architecture and Implementation*, volume 105 of *LNCS*, pages 246–265. Springer Verlag, 1981.

[28] J.-C. Laprie et al. *Dependability: Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer-Verlag, 1992.

[29] C. Lavarenne, O. Seghrouchni, Y. Sorel, and M. Sorine. The SYNDEX software environment for real-time distributed systems design and implementation. In *Proceedings of the European Control Conference*, volume 2, pages 1684–1689. Hermès, July 1991.

[30] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 14(6):569–578, 1997.

[31] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[32] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in hard real-time environnement. *Journal of the ACM*, 20(1):46–61, January 1973.

[33] S. Mishra and R.D. Schlicthing. Abstractions for constructing dependable distributed systems. Technical Report TR 92-19, University of Arizona, Department of Computer Science, Tucson, Arizona, 1992.

[34] F.P. Preparata, G. Metze, and R.T. Chien. On the connection assignment problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC16:848–854, 1967.

[35] S. Ragupathy, M.R. Leutze, and S.R. Schach. Message routing schemes in a hypercube machine. In *Conference on Hypercube Concurrent Computer Applications*, January 1988.

[36] K. Ramamritham, J.A. Stankovic, and P.F. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, April 1990.

[37] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Transactions on Software Engineering*, 18(9):785–793, September 1992.

[38] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and Systems Safety*, 43(2):189–219, 1994. Research Report n^o CSL-93-01.

[39] R.D. Schlichting and F.B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[40] F.B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[41] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[42] S. Sekhavat and J. Hermosillo. The CYCAB robot: A differentially flat system. In *IEEE Intelligent Robots and Systems, IROS'00*, Takamatsu, Japan, November 2000. IEEE Society Press.

[43] Y. Sorel. Massively parallel computing systems with real time constraints, the "algorithm architecture adequation" methodology. In *Massively Parallel Computing Systems Conference*, Ischia, Italy, May 1994.

[44] A.S. Tanenbaum. *Distributed operating systems*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[45] A.S. Tanenbaum. *Computer Networks, 3rd edition*. Prentice Hall, Englewood Cliffs, NJ, June 1996.

[46] AEROSPATIALE, CEA, ELF, SCHNEIDER ELECTRIC, and SIEMENS ELECTROCOM. Survey of industrial practices. Technical Report, CRISYS ESPRIT Project 25.514, February 1998.

[47] VÉRIMAG. *The Declarative Code DC*, 1.2b edition, June 1999. Available at `http://www-verimag.imag.fr/~raoul/DC-WWW`.

[48] A. Vicard. *Formalisation et Optimisation des Systèmes Informatiques Distribués Temps-Réel Embarqués*. PhD Thesis, University of Paris XIII, July 1999.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399