Dynamic Reconfiguration in Distributed Hard Real-Time Systems

by

Dick Alstein

91/01

April, 1991

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author or the editor.

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Dynamic Reconfiguration in Distributed Hard Real-Time Systems

Dick Alstein

January 23, 1991

Abstract

The paper focuses on the problem of dynamic workload reconfiguration in distributed systems. After the failure of a node, the processes that were allocated to that node must be redistributed among the remaining nodes. The main technique considered is process redundancy.

After an overview of design issues and solving techniques, the relevant literature is discussed. Proposals are given for the methods to be used in the DEDOS project, with directions for further research.

1 Introduction

This paper is intended to offer starting points for the development of fault-tolerance concepts and dynamic reconfiguration algorithms for the DEpendable Distributed Operating System DEDOS. Dynamic reconfiguration allows a distributed system to continue service after the occurrence of a failure, i.e. to achieve fault-tolerance.

Distributed computer systems are necessary in order to control a physically dispersed environment. In addition, they offer the possibility to achieve real-time response by exploiting true parallellism, transparency with respect to service usage and resource allocation, a high level of fault-tolerance by exploiting redundancy, flexibility and extendability. A distributed system consists of a number of processing units (*nodes*), connected to each other by a network of communication channels (*links*). Information is exchanged between the nodes by passing messages. As opposed to parallel machines, the nodes in a distributed system work asynchronously, and not necessarily on the same task. The workload is distributed among the nodes by a *global scheduling algorithm.* A *local scheduling algorithm* is responsible for allocating the resources of one node to several, possibly concurrent, processes.

During operation, it may happen that one or more nodes fail. In that case, we don't want a total breakdown, but a continuation of service or a graceful degradation of performance. That is, we want the system to continue working with the remaining nodes as well

as possible and complete the assigned work. The reorganization that is needed to achieve this is called *dynamic reconfiguration.*

An event related to node failure is that of a node coming up, either as an addition to the system or after it has been repaired. Though this is not a failure, and does not endanger the proper functioning of the system, we would like to use the new node, for reasons of efficiency and increased reliability. This requires a similar reorganization.

We can distinguish two kinds of reconfiguration:

Network reconfiguration is needed when a link or a node fails. Failure of a node implies that the connected communication links are out of order. The network must be checked for partitionings, its new topology determined, and new routing schemes constructed.

Workload reconfiguration. Each failed node was executing a number of processes, which must now be allocated to other nodes. Workload reconfiguration is also necesarry when a single process fails, e.g. because of a fault in the execution environment.

This paper will concentrate on the problem of workload reconfiguration. Special attention will be paid to the specific issues related to reconfiguration in hard real-time systems.

The rest of the paper is laid out as follows. The next section contains a general discussion of some of the issues related to reconfiguration, and techniques for solving them. The third section gives an overview of a number of relevant publications. After that, suggestions are made which methods to use in the DEDOS project. The final section contains concluding remarks and directions for further research.

2 Design issues and techniques

Phases. In order to achieve fault-tolerance by recovery the following phases are necessary:

- Error detection: Recognizing the activation of a fault. The reconfiguration procedure is initiated by the detection of an error. (e.g. after a timeout in communication, a 'disagreement' in voting, or by special checking hardware.)

- Fault passivation and diagnosis: Isolation of the part of the system that is in error, to prevent further propagation of the error throughout the system. This is followed by a diagnosis to determine the location and nature of the fault.

- Fault treatment: System repair, followed by the restoration of a consistent system state to make continued operation possible. The repair may be done either manually (e.g. replacement or off-line reconfiguration) or automatically (e.g. on-line reconfiguration). In this paper we concentrate on the latter.

Alternatively, diagnosis may be executed as a periodic check, instead of event-driven diagnosis.

The lowest level component of the system that the diagnosis can recognize as faulty is called the *unit of failure*; the whole component is diagnosed as having failed, even if part of it is still functioning correctly. The reconfiguration also has a certain 'granularity', the *unit of reconfiguration.* This will have to be at least as big as the unit of failure, since diagnosis information is used by the reconfiguration algorithm.

Failure types. Failures[1] can be classified by two characteristics: single/multiple and independent/related. The first division is on the number of failures in a given time interval. Multiple failures can have separate causes, in which case they are called independent, or they can be related. In the latter case the failures result from the propagation of errors through the system.

Failure mode. The designer has to make assumptions about the behaviour of a failed node. He has a choice of *fail-silent* (no output after failing), *exceptional* (recognizably indicating a failure), and *byzantine* (any output is possible) behaviour. This choice will deeply affect the complexity of the reconfiguration procedure, as it must take care to use only information from correctly working nodes.

The types and modes of failure and the combinations of failures that the reconfiguration algorithm can handle are stated in the *fault hypothesis.* Failures outside the fault hypothesis are called *catastrophic*, as they result in unpredictable system behaviour.

Fault-tolerance taxonomy. Basically, tolerance to failures is achieved by redundancy, either in hardware or software. When using hardware redundancy, the network contains a number of extra nodes. During normal operation, these nodes will remain idle. When the failure of a node is detected, its workload is transferred to one of the extra nodes. This technique, though simple, has important disadvantages:

- Low efficiency. The extra nodes are not to be used until needed as a replacement.
- Low speed. Time is needed to retrieve the data for the lost processes[2] from stable storage. If speed is crucial, this delay can be avoided by giving each node its private 'shadow node', running the same processes. However, this will greatly increase hardware costs.
- Low flexibility. If all nodes have the same hardware there is no problem, but if a node is equipped with special hardware (e.g. sensors, actuators), then the replacement node must have the same equipment.

[1] In this paper we will use the fault/error/failure model from [Lapr85]. The division of failure types given in that publication, however, is not used here.

[2] Confusion may arise on the meaning of the word *process*, as various definitions are used in the publications mentioned in section 3. For the general discussion in this section, we will define a process as the unit of global scheduling, i.e. the process as a whole will be allocated to a node.

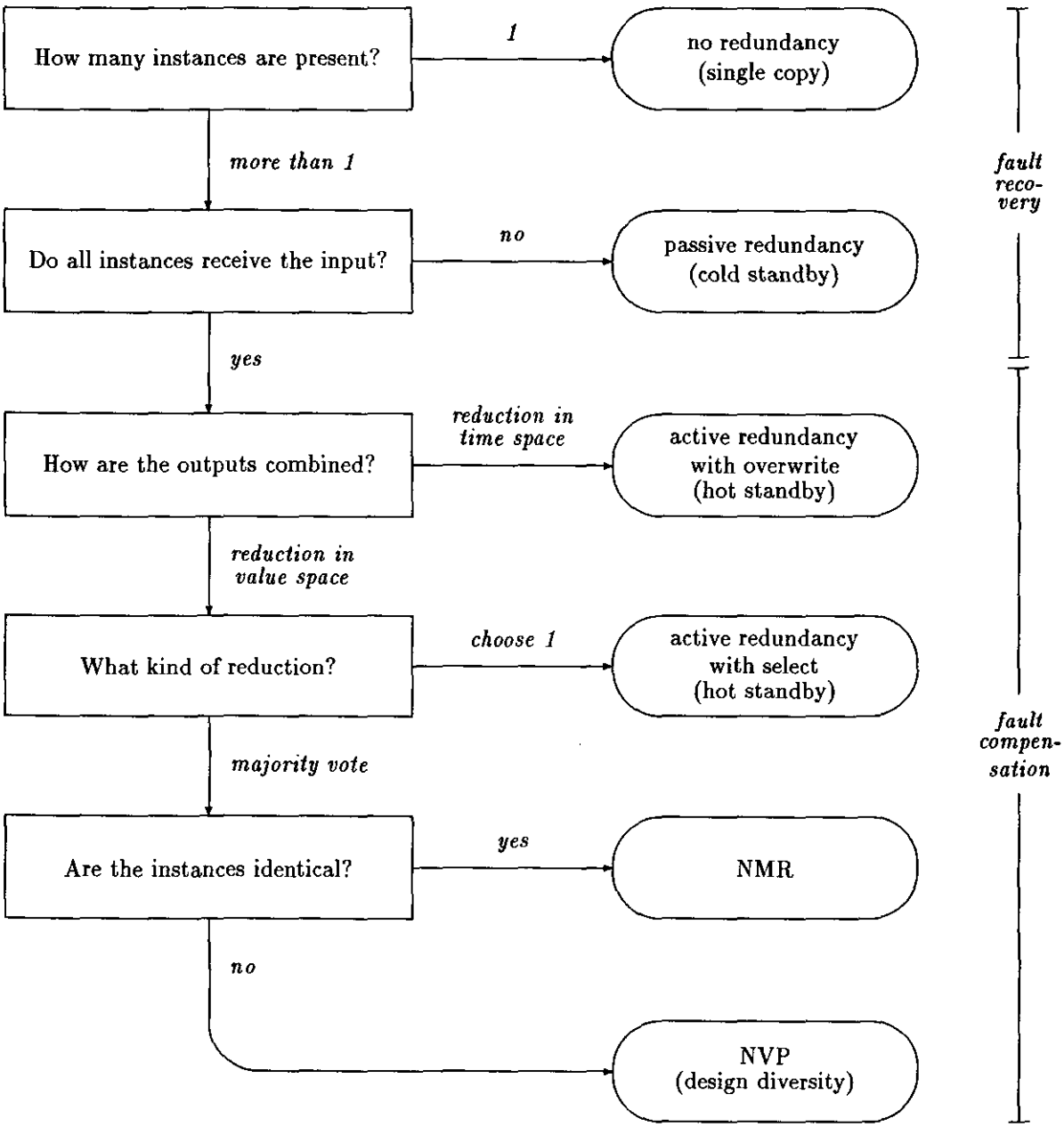
Therefore, the main technique we will consider for workload reconfiguration is software redundancy in the form of *process redundancy*: creating more than one instance of a process. A number of posibilities emerge:

- The new instances may be created at the same time as the original (*standby*), or only after detection of a failure (*single copy*).
- In case of standby, the redundant processes may be idle (*passive redundancy* or *'cold' standby*), or run in parallel, receiving the same input (*active redundancy* or *'hot' standby*).

  There is an intermediate method between passive and active redundancy (which might be called 'tepid standby'). We have one active instance and a number of passive ones. Instead of writing checkpoints of the active instance to stable storage (which is relatively slow), they are sent to the passive instances, to update their state. If the active instance is lost, we only need to replay the message history in order to restore a correct state.
- In case of active redundancy, the outputs of the instances must be combined to one. This can be done by a reduction in either 'time space' or 'value space'. In the first case, the outputs overwrite each other; a result is stored until it is replaced by one from an other instance. Reduction in value space can be done by selecting the output of a single instance while discarding the rest, or by majority voting (*n-modular redundancy, NMR*).
- When using majority voting, there is the option to use instances that are not completely identical, but have been developed separately. They are based on the same specification, but ideally use different algorithms, in order to prevent failures caused by software faults (*n-version programming, NVP*).

There is a tradeoff between the amount of resources taken by redundancy and the number of measures necessary to restore a proper functioning in case of a failure. Passive redundancy has an advantage over single copy in that the new instances are already allocated and loaded, but at the cost of higher memory usage. With active redundancy, we don't need to get the new instances into a consistent state, as they run in parallel. However, this does increase the workload under normal (i.e. failure-free) working conditions. Voting increases tolerance to failures, but it takes time to collect the votes and calculate the result.

All these possibilities are summarized in the figure below.



In the first two cases we talk about *fault recovery*: the occurrence of a failure leads to an erroneous system state, and specific measures must be taken in order to transform it to a correct one. System performance is degraded during these recovery actions. The other cases are examples of *fault compensation* (or *fault masking*): failures have no influence on

system service, as long as they remain within the fault hypothesis. Fault compensation decreases the effective performance under normal operating conditions.

State consistency. When fault recovery is used, the reconfiguration algorithm must not only provide new instances to replace the processes on the failed node, it must also ensure that they have a consistent initial state. Feasible methods for achieving this goal are:

- Forward recovery: Simply restart the processes affected by the failure. This method is often convenient in simple process control systems where the correct system state can be deduced from the environment, e.g. for control loops.
- Backward recovery: Read the latest checkpoint from stable storage, and replay the message history since that moment. If no checkpointing is done, then the initial state is used. Backward recovery is usually found in transaction processing systems, e.g. database systems.

Redundancy level. Normally, when a process is invoked, it will specify the desired *redundancy level* (the number of instances). When one or more instances are lost because of node failure, the actual redundancy level decreases. The reconfiguration algorithm then has a choice of:

- maintaining the redundancy level (by generating new instances).
- maintaining only a minimum level (coming into action when the number of instances drops below a given minimum).
- taking no action at all.

This design decision depends upon the probability of a node failure, and on the required level of fault-tolerance.

A more elaborate method of 'redundancy level control' is to let the operating system decide on the generation of instances, based on process fault-tolerance level and current workload.

Reallocation. In the case of no redundancy, or when the redundancy level is maintained, new instances must be generated to replace lost ones. These replacement processes must of course be allocated to correct (i.e. non-failed) nodes. This job is closely related to global scheduling. The scheduler must ensure that duplicate instances are assigned to different nodes.

In hard real-time systems, the scheduler must guarantee that deadlines are met. Node failures will invalidate this guarantee, unless the time consumed by the reconfiguration algorithm is taken into account. This, however, is often very difficult because of the complexity of the scheduling problem.

Network partitioning. If a partitioning of the network is possible (i.e. included in the fault hypothesis), special care should be taken when designing the reconfiguration algorithm. When a failure has partitioned the network, each part may view the rest as being out of order. Consequently, it will try to take over every task that was running. For non-idempotent operations this leads to catastrophic situations in the environment.

A possible solution would be to continue operation only with the part that contains more than half the original number of nodes, and execute a shutdown procedure on the other parts. However, there may be no part of the required size.

Communication. The techniques discussed above pose some demands on the communication services of the system:

- When active redundancy is used, all instances must receive input messages. For reasons of transparency, it is desirable that the sender does not need to know the number of instances and their location. The communication layer should take care of this.

  In addition, the reduction of multiple outputs to a single one may be offered as a system service.

- In order to be able to replay the message history, it must be logged on stable storage. When replaying it, the processes that are being restored to their state before the failure will also be generating output. The messages that had already been sent before must be intercepted. This can be realized e.g. by message numbering.

Output consistency. When active redundancy is used, the multiple outputs must be combined. To facilitate this, we require *output consistency*: all instances on correct nodes must produce the same output messages, and in the same order. If the computation is deterministic, then a sufficient condition for output consistency is *input consistency*: all instances process the input messages in the same order. In practice, input consistency is often provided by the communication system.

However, nondeterminsm may be introduced into the computation, either by the scheduler (e.g. receive-statements) or by the program itself (e.g. case-statements, random number generation). If nondeterministic constructs are allowed, then special synchronization measures must be taken to ensure that all instances follow the same execution path.

3 Relevant literature

The relevant literature was scanned. Only four publications relating to workload reconfiguration were found. They are summarized below.

Distributed fault-tolerant real-time systems: The MARS approach [Kope89]. This article gives an overview of the architecture of the MARS system. Features to be noted when looking at reconfiguration are:

- The use of fail-silent components. Fail-silent behaviour is guaranteed through the addition of self-checking software and specially designed checking hardware to each node (as long as the failure stays within the fault hypothesis). Upon detection of a failure, all output is cut off.

- Process redundancy by using hot standby. MARS knows two kinds of output messages, and treats them differently. *State messages* (e.g. about the temperature or pressure) simply overwrite each other (reduction in time space). With *event messages* (like the pressing of a button), the first one that arrives is accepted; subsequent ones are discarded (reduction in value space with select). Because of the fail-silent behaviour, any output that is received is correct. Consequently, voting does not increase fault-tolerance.

Decentralized decision making for task reallocation in a hard real-time system [Stan89]. As expressed in the title, the article presents a decentralized algorithm for the reallocation of tasks[3]. It was specifically designed for hard real-time systems, implying that it considers deadlines and is aimed at making decisions fast. The algorithm is meant to work in cooperation with a local scheduler, a global scheduler and a task dispatcher that were presented in previous publications [Zhao87].

Tasks can run with both active and passive redundancy. When a task is invoked, it specifies two *replication factors*, one for the number of active copies and one for passive copies. For active copies, this redundancy level is kept above a minimum value (typically the number of instances necessary for a valid vote on the outputs). The redundancy level for passive copies is maintained.

The basic concept for reallocation is the *buddy site.* When a task is started, a list of buddy sites is chosen. The list consists of nodes that can meet the tasks' resource requirements. The length of the list (the *reallocation factor*) and the resource requirements are specified by the task. If the original node fails, the first node in the list becomes responsible for the execution of the task. If it cannot accommodate the task by itself, a number of methods can be used to find another node:

- Bidding. A broadcast is sent, asking which node can take the task.

- Focused addressing. The task is sent directly to a node that is viewed as having a low workload.

- Cancelling lower priority tasks.

[3]In this article, a task is defined as the non-preemptable unit of computation. Tasks are assumed to be independent.

The order in which these methods are employed depends on policy and current system state.

The paper concludes with a performance analysis based on a simulation program.

Reconfiguration procedure in a distributed multiprocessor system [Bari82]. This paper describes the reconfiguration algorithm of the MuTEAM prototype. After detection of a failure and its diagnosis, reallocation of nonredundant processes is achieved by means of *choice messages.* Each node decides which processes it can accommodate, and makes this choice known to the other nodes by broadcasting a choice message. A conflict arises if a process is claimed by two nodes at the same time. This is resolved by assigning arbitrarily chosen priorities to the nodes. The node with the highest priority gets the process.

Each node keeps an allocation table, containing an entry for each process that must be reallocated. The choice messages fill this table. Filled entries can only be overwritten by choice messages from a node with higher priority. When the table is full, the node waits for a period equal to twice the maximum communication delay, to allow for the processing of choice messages under way. Then it exits the algorithm. In this way, it is ensured that, upon exit, the same reallocation information is present at each node.

A decentralized fault diagnosis algorithm was decribed in a previous article [Ciom81].

The Delta-4 Extra Performance Architecture [Barr90]. In the Delta-4 system, several process redundancy techniques are available to the applications programmer: active redundancy, passive redundancy, and a new technique called the leader/follower model. This is a form of hot standby that tries to combine the advantages of active and passive redundancy. All instances are active, but only one, the *leader*, delivers its output. The output messages from the other instances (the *followers*) are discarded by the communication system. Program code is not required to be deterministic; if the leader reaches a point in the computation where a decision is taken which affects determinism, it sends its decision to the followers. To retain state consistency in case of preemption, the leader/follower model also incorporates a preemption synchronization mechanism.

Output validation is not necessary, since the nodes in a Delta-4 system are fail-silent. Each node consists of a host computer and a specialized communication processor, the Network Attachment Controller (NAC). The NAC ensures the fail-silent behaviour.

4 Fault-tolerance models for DEDOS

The DEDOS project is currently in progress at the Eindhoven University of Technology. It is aimed at the development of techniques and methods for the construction of a DEpendable Distributed Operating System, for use in hard real-time environments. For an overview of the project, see [Stok91].

In this section, suggestions are made for the reconfiguration methods to be used in the DEDOS project. Two versions are presented, differring in the complexity of measures

taken. Before looking at the differences and the characteristics that they have in common, we list the assumptions underlying both versions.

- Processes are dependent.
- The nodes are connected by a broadcast network; there is a central communication channel ('backbone') to which each node is connected by a single link. This implies that when a link fails, the connected node is isolated from the rest of the network, leading to a node failure.
- Reliable communication services are present.
- The fault hypothesis includes node and process failures, single and multiple failures, and independent and related failures. Failures caused by software faults, other than process or node crashes, are excluded.
- Nodes are assumed to have a fail-silent behaviour.
- All hard real-time (*HRT*) processes are periodic and known to the scheduler before system startup. For simplicity, we assume that the nature of HRT processes is such that forward recovery is possible.

Below, the characteristics shared by both versions are discussed.

- Process redundancy is a technique which takes a lot of processing resources, and should be used only for critical processes. Therefore, only HRT processes run with hot standby. The required redundancy level is specified by the process.
- Soft real-time (*SRT*) processes run as a single copy. After a failure, they are reallocated and restarted.
- Loss of all remaining instances of an HRT process will lead to violation of the deadline, and initiates an exception schedule (using forward recovery). SRT processes that were running on a failed node will be aborted.
- In DEDOS, backward recovery will be used for objects of SRT processes if they are declared recoverable by the programmer. To make an SRT process recoverable after reallocation, all the objects it uses must be recoverable as well.

  Because hot standby is used, HRT processes do not need backward recovery. If some instances are lost, the remaining ones ensure a proper functioning. If all instances are lost, forward recovery is preferred to reading checkpoints.
- For HRT, the unit of failure is the process.

In the DEDOS project, a choice has been made to perform the scheduling of HRT threads statically (off-line). SRT threads will be scheduled on-line. Static scheduling allows one to use scheduling methods that would be too time-consuming when used dynamically. Timely execution can be guaranteed before system startup, provided that no failures outside the fault hypothesis occur. However, the situation becomes complicated when we consider the possibility of node failure. Unless the fault hypothesis is extremely simple, generating a (partial) schedule to adjust to every type and moment of failure will be too expensive in terms of calculation time and storage space. If we want to avoid this, there are basically two possibilities:

1. Restrict the reconfiguration, so that no new HRT processes are generated. Thus, no recalculation of the schedule is needed.

2. Use a second, faster method to generate the new schedule on-line.

Along these lines, two versions can be constructed:

Simple version. Characteristics:

- If an instance of an HRT process is lost, no new instance is generated (i.e. the redundancy level is not maintained). The remaining instances ensure that the work is completed.
- A node that is added to the system is used to take over the workload of a node that has failed. However, at the moment a node comes up, there may be no failed node (i.e. the system has not yet experienced a node failure). In that case, the new node can only be used for SRT processes. In order to get a balanced workload, SRT processes already allocated to other nodes may be migrated to the new node.
- Diagnosis is performed periodically, in order to make it better schedulable. It consists of simply updating state information, i.e. the involved process instances are marked as lost, and no further fault diagnosis is done. The worst-case execution time needed for diagnosis and subsequent reconfiguration is low, so that it can be incorporated in the HRT schedule.

Because the redundancy level decreases in time, this version can only be of use if the failure rate is low in comparison with the time that the system must be operational.

Elaborate version. Characteristics:

- The redundancy level is maintained, ensuring a constant level of tolerance to node and process failures.
- A fast method for modifying the existing HRT schedule (or generating new ones) must be devised. Because of this speed requirement, it will be heuristic and sub-optimal.

- Diagnosis is event-driven, in order to adapt to the new situation fast. In addition, the diagnosis is more detailed than in the simple version and tries to locate failures down to the lowest possible unit of failure.

A drawback of this version is the extra design effort, as it requires both an on-line and an off-line scheduler. Furthermore, optimality of the schedule is lost at the first failure. It remains to be seen whether an on-line scheduler can be designed that is both fast enough and able to generate a feasible schedule.

As usual, we come up with an intermediate solution.

Mixed version. In this version, reconfiguration is done in two steps:

- For a quick response after failure detection, we will only update state information and continue with the remaining instances of the HRT processes involved, like in the simple version.
- At the same time, an SRT task is started, to calculate a new schedule, optimized for use with the new configuration. When the new schedule is ready (and provided no additional failures are detected in the mean time), the workload is allocated anew. Thus, an upper bound for the initial response time can be guaranteed, while eventual optimality of the schedule is retained if the failure rate is low enough. The redundancy level is maintained in the new schedule.
- Diagnosis is event-driven, as in the elaborate version.

5 Conclusion

Since not much literature on fault-tolerance and dynamic reconfiguration is available, and this area is becoming increasingly important, further research is justified. Especially the following aspects require further investigation.

- The assumption that failed nodes have fail-silent behaviour is not quite realistic, but it avoids design complications. In a later stadium, this assumption may be relaxed towards exceptional behaviour. Byzantine failures remain excluded.
- Care must be taken to avoid unwanted effects of duplicate outputs from instances of HRT processes. This can be done either by making sure that overwriting is harmless, or by intercepting the messages in the communication layer. In any case, these measures must be such that an instance need not be 'aware' of the existence of other instances; it is allowed to generate output, and it is up to the operating system to handle it correctly.
- When a new schedule has been calculated after a failure, complications may be encountered. The switchover to the new schedule must be synchronized, e.g. to the begin of a new period.

- As yet, no diagnosis algorithm has been developed for DEDOS. The diagnosis should be decentralized, so that no single node can decide on the shutting off of other nodes or processes.
- An on-line scheduler, both for SRT and HRT tasks in case of a node or process failure is needed.
- The problems encountered when the network is partitioned, as described in section 2, have not yet been dealt with.
- It would be interesting to study the effects of nondeterminism on output consistency (as mentioned in section 2), and to determine what synchronization measures would be sufficient.

After elaboration of these issues the fault-tolerance models for DEDOS must be formalized. How this can be done is presently investigated by a related project[4]. Only then the various algorithms can be designed and verified.

References

[Bari82] G. Barigazzi, A. Ciuffoletti, L. Strigini, "Reconfiguration procedure in a distributed multiprocessor system", Proceedings of the 12^{th} Fault-tolerant computing symposium (FTCS-12), June 1982, pp. 73–80.

[Barr90] P.A. Barrett e.a.,"The Delta-4 Extra Performance Architecture", Proceedings of the 20^{th} Fault-tolerant computing symposium (FTCS-20), June 1990, pp. 481–488.

[Ciom81] P. Ciompi, F. Grandoni, L. Simoncini, "Distributed diagnosis in distributed multiprocessor systems: the MuTEAM approach", Proceedings of the 11^{th} Fault-tolerant computing symposium (FTCS-11), June 1981, pp. 25–29.

[Kope89] H. Kopetz e.a., "Distributed fault-tolerant real-time systems: The MARS approach", IEEE Micro, February 1989, pp. 25–40.

[Lapr85] J.C. Laprie, "Dependable computing and fault tolerance: concepts and terminology", Proceedings of the 15^{th} Fault-tolerant computing symposium (FTCS-15), June 1985, pp. 2–11.

[Stan89] J.A. Stankovic, "Decentralized decision making for task reallocation in a hard real-time system", IEEE Transactions on Computers, Vol. 38, No. 3, March 1989, pp. 341–355.

[4]The Dutch STW project "Fault-Tolerance: Paradigms, Models, Logics, Construction" (grant number NWI88.1517).

[Stok91] P.D.V. v.d. Stok e.a., "The dependable distributed operating system DEDOS", to be published.

[Zhao87] W. Zhao, K. Ramamritham, J.A. Stankovic, "Scheduling tasks with resource requirements in hard real-time systems", IEEE Transactions on Software Engineering, Vol. 13, No. 5, May 1987, pp. 564–577.

86/14	R. Koymans	Specifying passing systems requires extending temporal logic.
87/01	R. Gerth	On the existence of sound and complete axiomati zations of the monitor concept.
87/02	Simon J. Klaver Chris F.M. Verberne	Federatieve Databases.
87/03	G.J. Houben J.Paredaens	A formal approach to distributed information systems.
87/04	T.Verhoeff	Delay-insensitive codes - An overview.
87/05	R.Kuiper	Enforcing non-determinism via linear time temporal logic specification.
87/06	R.Koymans	Temporele logica specificatie van message passing en real-time systemen (in Dutch).
87/07	R.Koymans	Specifying message passing and real-time systems with real-time temporal logic.
87/08	H.M.J.L. Schols	The maximum number of states after projection.
87/09	J. Kalisvaart L.R.A. Kessener W.J.M. Lemmens M.L.P. van Lierop F.J. Peters H.M.M. van de Wetering	Language extensions to study structures for raster graphics.
87/10	T.Verhoeff	Three families of maximally nondeterministic automata.
87/11	P.Lemmens	Eldorado ins and outs. Specifications of a data base management toolkit according to the functional model.
87/12	K.M. van Hee and A.Lapinski	OR and AI approaches to decision support systems.
87/13	J.C.S.P. van der Woude	Playing with patterns - searching for strings.
87/14	J. Hooman	A compositional proof system for an occam-like real-time language.
87/15	C. Huizing R. Gerth W.P. de Roever	A compositional semantics for statecharts.
87/16	H.M.M. ten Eikelder J.C.F. Wilmont	Normal forms for a class of formulas.
87/17	K.M. van Hee G.-J.Houben J.L.G. Dietz	Modelling of discrete dynamic systems framework and examples.

In this series appeared :

No.	Author(s)	Title
85/01	R.H. Mak	The formal specification and derivation of CMOS-circuits.
85/02	W.M.C.J. van Overveld	On arithmetic operations with M-out-of-N-codes.
85/03	W.J.M. Lemmens	Use of a computer for evaluation of flow films.
85/04	T. Verhoeff H.M.L.J.Schols	Delay insensitive directed trace structures satisfy the foam the foam rubber wrapper postulate.
86/01	R. Koymans	Specifying message passing and real-time systems.
86/02	G.A. Bussing K.M. van Hee M. Voorhoeve	ELISA, A language for formal specification of information systems.
86/03	Rob Hoogerwoord	Some reflections on the implementation of trace structures.
86/04	G.J. Houben J. Paredaens K.M. van Hee	The partition of an information system in several systems.
86/05	J.L.G. Dietz K.M. van Hee	A framework for the conceptual modeling of discrete dynamic systems.
86/06	Tom Verhoeff	Nondeterminism and divergence created by concealment in CSP.
86/07	R. Gerth L. Shira	On proving communication closedness of distributed layers.
86/08	R. Koymans R.K. Shyamasundar W.P. de Roever R. Gerth S. Arun Kumar	Compositional semantics for real-time distributed computing (Inf.&Control 1987).
86/09	C. Huizing R. Gerth W.P. de Roever	Full abstraction of a real-time denotational semantics for an OCCAM-like language.
86/10	J. Hooman	A compositional proof theory for real-time distributed message passing.
86/11	W.P. de Roever	Questions to Robin Milner - A responder's commentary (IFIP86).
86/12	A. Boucher R. Gerth	A timed failures model for extended communicating processes.
86/13	R. Gerth W.P. de Roever	Proving monitors revisited: a first step towards verifying object oriented systems (Fund. Informatica IX-4).

87/18	C.W.A.M. van Overveld	An integer algorithm for rendering curved surfaces.
87/19	A.J.Seebregts	Optimalisering van file allocatie in gedistribueerde database systemen.
87/20	G.J. Houben J. Paredaens	The R^2 -Algebra: An extension of an algebra for nested relations.
87/21	R. Gerth M. Codish Y. Lichtenstein E. Shapiro	Fully abstract denotational semantics for concurrent PROLOG.
88/01	T. Verhoeff	A Parallel Program That Generates the Möbius Sequence.
88/02	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specification for Information Systems.
88/03	T. Verhoeff	Settling a Question about Pythagorean Triples.
88/04	G.J. Houben J.Paredaens D.Tahon	The Nested Relational Algebra: A Tool to Handle Structured Information.
88/05	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable Specifications for Information Systems.
88/06	H.M.J.L. Schols	Notes on Delay-Insensitive Communication.
88/07	C. Huizing R. Gerth W.P. de Roever	Modelling Statecharts behaviour in a fully abstract way.
88/08	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	A Formal model for System Specification.
88/09	A.T.M. Aerts K.M. van Hee	A Tutorial for Data Modelling.
88/10	J.C. Ebergen	A Formal Approach to Designing Delay Insensitive Circuits.
88/11	G.J. Houben J.Paredaens	A graphical interface formalism: specifying nested relational databases.
88/12	A.E. Eiben	Abstract theory of planning.
88/13	A. Bijlsma	A unified approach to sequences, bags, and trees.
88/14	H.M.M. ten Eikelder R.H. Mak	Language theory of a lambda-calculus with recursive types.

88/15	R. Bos C. Hemerik	An introduction to the category theoretic solution of recursive domain equations.
88/16	C.Hemerik J.P.Katoen	Bottom-up tree acceptors.
88/17	K.M. van Hee G.J. Houben L.J. Somers M. Voorhoeve	Executable specifications for discrete event systems.
88/18	K.M. van Hee P.M.P. Rambags	Discrete event systems: concepts and basic results.
88/19	D.K. Hammer K.M. van Hee	Fasering en documentatie in software engineering.
88/20	K.M. van Hee L. Somers M.Voorhoeve	EXSPECT, the functional part.
89/1	E.Zs.Lepoeter-Molnar	Reconstruction of a 3-D surface from its normal vectors.
89/2	R.H. Mak P.Struik	A systolic design for dynamic programming.
89/3	H.M.M. Ten Eikelder C. Hemerik	Some category theoretical properties related to a model for a polymorphic lambda-calculus.
89/4	J.Zwiers W.P. de Roever	Compositionality and modularity in process specification and design: A trace-state based approach.
89/5	Wei Chen T.Verhoeff J.T.Udding	Networks of Communicating Processes and their (De-)Composition.
89/6	T.Verhoeff	Characterizations of Delay-Insensitive Communication Protocols.
89/7	P.Struik	A systematic design of a paralell program for Dirichlet convolution.
89/8	E.H.L.Aarts A.E.Eiben K.M. van Hee	A general theory of genetic algorithms.
89/9	K.M. van Hee P.M.P. Rambags	Discrete event systems: Dynamic versus static topology.
89/10	S.Ramesh	A new efficient implementation of CSP with output guards.
89/11	S.Ramesh	Algebraic specification and implementation of infinite processes.
89/12	A.T.M.Aerts K.M. van Hee	A concise formal framework for data modeling.

90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in concurrent logic languages: a fully abstract model based on sequences, p. 29.
90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses of "if..., then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.

89/13	A.T.M.Aerts K.M. van Hee M.W.H. Hesen	A program generator for simulated annealing problems.
89/14	H.C.Haesen	ELDA, data manipulatie taal.
89/15	J.S.C.P. van der Woude	Optimal segmentations.
89/16	A.T.M.Aerts K.M. van Hee	Towards a framework for comparing data models.
89/17	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra.
90/1	W.P.de Roever-H.Barringer C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, pp. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.