

Design and Implementation of a Data Stabilizing Software Tool

Vincenzo De Florio, Geert Deconinck, Rudy Lauwereins
Katholieke Universiteit Leuven, Electrical Engineering Dept., ACCA Division,
Kard. Mercierlaan 94, B-3001 Leuven, Belgium

Stephan Graeber
Deutsches Zentrum für Luft- und Raumfahrt, Institut für Flugmechanik,
Lilienthalplatz 7, D-38108 Braunschweig, Germany

Abstract

We describe a software tool which implements a software system for stabilizing data values, capable of tolerating both permanent faults in memory and transient faults affecting computation, input and memory devices by means of a strategy coupling temporal and spatial redundancy. The tool maximizes data integrity allowing a new value to enter the system only after a user-parameterizable stabilization procedure has been successfully passed. Designed and developed in the framework of the ESPRIT project TIRAN, the tool can be used stand-alone but can also be coupled with other dependable mechanisms developed within that project. Its use is being currently investigated within ENEL, the main Italian electricity supplier, in order to replace a hardware stable storage device adopted in their high-voltage sub-stations.

1. Introduction

This paper describes the design and the implementation of a data stabilizing software system (DSS), viz., a fault-tolerant software component that allows to validate input and output data by means of a stabilization procedure. The DSS has been developed with the explicit goal of taking over a pre-existing stable storage hardware device used at ENEL S.p.A.—the main Italian electricity supplier, the third largest world-wide—within a programme for sub-station automation of their high voltage sub-stations. This hardware device is able to tolerate the typical faults of a highly disturbed environment subject to electro-magnetic interference: transient faults affecting memory modules and the processing devices, often resulting in bit flips or even in system crashes. This hardware component was mainly used

within control applications with a cyclic behavior only dependent on their state (Moore automata). Typically these applications:

- read their current state from stable storage,
- produce with that an output that, once validated, is propagated to the field,
- then they read their input from the field and compute a tentative future state and future output.

The whole cycle is repeated a number of times in order to validate the future state. When this temporal redundancy scheme succeeds, the tentative state is declared as a valid next state and the stable storage is updated accordingly. The cyclic execution is paced by an external periodic signal which resets the CPU and re-fetches the application code from an EPROM. External memory is not affected by this step. This policy and the nature of faults (frequency, duration etc.) allow to confine possible impairment affecting the internal state of the application within one cycle.

Developed in the framework of the ESPRIT project TIRAN¹, a prototypic version of this data stabilizing tool has been successfully integrated in a test-bed control application at ENEL, whose cyclic behaviour is regulated by a periodic restart device—the only custom, dedicated component of the architecture. Initially developed on a Parsytec CC system equipped with 4 processing nodes, the tool has been then ported to a number of runtime systems; at ENEL, the tool is currently running under the TEX nanokernel [7] and VxWorks on several hardware boards, each based on the DEC Alpha processor [3]. Preliminary results on these systems show that the tool is capable to fulfil its dependability and data integrity requirements, adapting itself to a number of different simulated disturbed environments thanks to

¹TIRAN, or Tailorable Fault Tolerance Framework for Embedded Application, is the name of ESPRIT project 28620.

its flexibility. Ongoing experience and validation activities, planned in the framework of the recently started IST Project DepAuDE (“*Dependability for embedded Automation systems in Dynamic Environments with intra-site and inter-site distribution aspects*”), will allow to gain additional confidence with the novel solution at ENEL and will culminate in its adoption in place of the old hardware equipment and in the analysis of its re-use in a wide class of mission-critical automation applications.

The structure of this paper is as follows: Section 2 draws an analysis of the requirements to the DSS tool. Basic functionalities of the tool are summarized in Sect. 3. The two “basic blocks” of our tool, viz. the manager of redundant memories and the data stabilizer, are described respectively in Sect. 4 and Sect. 5. Our conclusions are finally drawn in Sect. 6.

2. Requirements for the data stabilizer

In an electrical power network, automation is a fundamental requirement for the subsystems concerning production, transport and distribution of energy. In many sites of such a network, remotely controlled automation systems play an important role towards reaching a more efficient and cost-effective management of the networks. While considering the option to install high performance computing nodes as controllers into such environments, the question of a software solution for a data stabilizer arose [4]. The goal of a data stabilizer is to protect data in memory from permanent faults affecting memory devices and from temporary faults affecting data of systems running in disturbed environments, as they typically arise by electro-magnetic interference (EMI), and to validate these data by means of a stabilization procedure based on the conjoint exploitation of temporal and spatial redundancy. When controlling high voltage, an important source of faults is electricity itself—because all switching actions in the field cause electrical disturbances, which enter the control computers via their I/O devices, often overcoming the filtering barriers. Furthermore, EMI causes disturbs in the controllers. Clearly, due to the very nature of this class of environments, such faults cannot be avoided; on the other hand, they should not impair the expected behaviour of the computing systems that control the automation system.

In order to overcome the effects of transient faults, temporal redundancy is employed. This means that all computation is repeated several times, assuming that due to the nature (frequency, amplitude, and duration) of the disturbances, not all of the cyclic replications are affected. As in other redundancy schemes, a final decision is taken via a voting between the different redundant results. Clearly this calls for a memory component that be more resilient to transient faults with respect to conventional memory devices.

In traditional applications often a special hardware device, called *stable storage device*, is used for this. The idea was to replace this special hardware with a software solution, which offers on the one hand more flexibility, and provides on the other hand the same fault tolerance functionality.

The following requirements were deduced from this:

1. The data stabilizer has to be implemented in conventional memory available on the hardware platform running the control application.
2. Typical control applications show a cyclic behaviour, which is represented in the following steps:
 - read sensor data,
 - calculate control laws based on new sensor data and status variables,
 - update status variables,
 - output actuator data.

The data stabilizer has to interface with such kind of applications.

3. The DSS has to tolerate any transient faults affecting the elaboration or the inputs. Furthermore, it has to tolerate permanent and transient faults affecting the memory itself.
4. The DSS has to store and to stabilize status data, i.e. if input data to the DSS have been confirmed a number of times, they should be considered as *stable*.
5. Because of this stabilization the DSS has to provide a high integrity, i.e. only correct output should be released.

A few further requirements were added, namely:

1. minimising the number of custom, dedicated, hardware components in the system: in particular, the system has to work with conventional memory chips;
2. making use of the inherently available redundancy of a parallel or distributed system;
3. maximising flexibility and re-usability, so to favour the adoption of the system in a wide application field, which, in the case of ENEL, ranges from energy production and transport to energy distribution;
4. eliminating the use of mechanisms possibly affecting the deterministic behaviour, for instance by not using dynamic memory allocation during the critical phases of real-time applications;
5. focusing on portability, so to have minimal dependencies with specific hardware platforms or specific operating systems;
6. allowing scalability, at least from 1 to 8 processors.

3. Ideas and basic principles

In the following we deduce the functionality of the DSS from the requirements stated in the last paragraph.

First we need to clarify the concept of *data stabilization*. Let us assume we want to develop a controller with a short cycle time with respect to the dynamics of the input and the output data. Disturbances from temporal faults can influence either the input or the output data. We want to eliminate these temporal effects in particular on the output data. For this reason, we let the controller run several times with the same input data. Let us furthermore assume that, in the absence of faults, the same output data are produced. This allows us to compare the output of the controller from several runs. If the output doesn't change in a number of consecutive cycles, we call the output *stable*. The described process of repeated runs of the controller and comparison of the results is called *stabilization*.

The described procedure of cyclic repetition of a process is the basis of *temporal redundancy* [5]. In order to detect and overcome transient faults, even if their characteristics as distribution of frequency and duration is not known, temporal redundancy can be applied. If the computation time for a process is outspoken longer than the expected duration of a transient fault, and the frequency of the disturbances is low enough, we will assume that in several repetitive computations of the same data only one fault may show up. So if the algorithm performs the same computation several times, there will be a period of some consecutive, not impaired results.

The number N_{time} of consecutive equal data inputs to the DSS is the level of *temporal redundancy*. It is the minimum number of cycles the DSS has to execute until a new input can be assumed to be stable.

Another strong requirement of our design is that of maximizing data integrity. To reach this goal, the DSS tool adopts a strategy, to be described later on, aiming at ensuring that data are only allowed to be stored in the DSS the moment they have been certified as being "stable". On a read request from a given memory location, DSS will then return the last stabilized data, while a write into DSS will actually take place only when the strategy guarantees that data that are going to be written are stable. Another important requirement is that permanent faults affecting the system should not destroy the data. A standard method for increasing the reliability of memory is replication of data in redundant memories: a "write" is then translated into writing into each of a set of redundant memories, with voting of the data when reading out. No additional hardware is required for this, as the writings are done in the memories

of the processing nodes of the target, distributed memory platform.

Using the principles of *spatial redundancy*, the same data are replicated in different memory areas—let us call them *banks*. The *spatial redundancy factor* N_{spat} is the number of replicas stored in the DSS. Changing this parameter the user is allowed to trade off dependability with performance and resource consumption.

In order to fulfil the above mentioned requirements the DSS implements a strategy based on two buffers, one for reading the last stabilized data, and the other for receiving the new data. We call these two buffers the *memory banks*.

- The bank used for the output of the stabilized data is called the *current bank*.
- The other bank, called *future bank*, receives the N_{time} input data for the DSS one after the other and checks whether the results are stable.

If the results are stable the role of the banks is switched, so that the future bank becomes the new current bank and the output data are fetched from there.

During design and implementation of the DSS tool, the idea arose to isolate the spatial redundancy from the tool to build an own tool especially devoted to the distributed memory approach—we call this the distributed memory tool (DMT). It showed that this approach simplifies the design and the implementation of the DSS tool.

4. The Distributed Memory Tool

The Distributed Memory Tool (DMT) implements the spatial redundancy scheme for the DSS.

Let a *local user context* be either a thread or a process, which the user application sees as one task with its own local environment.

Assume that the user application consists of several such local user contexts, which are distributed among several nodes of a multiprocessor system. The basic component of the DMT is the *local handler*, which is defined as follows:

A *local memory handler* is a local software module connected to one local user context and to a set of fully interconnected fellows. The attribute "local" means that both user context and memory handler run on the same processing node and they represent the whole tool from the viewpoint of the processing node.

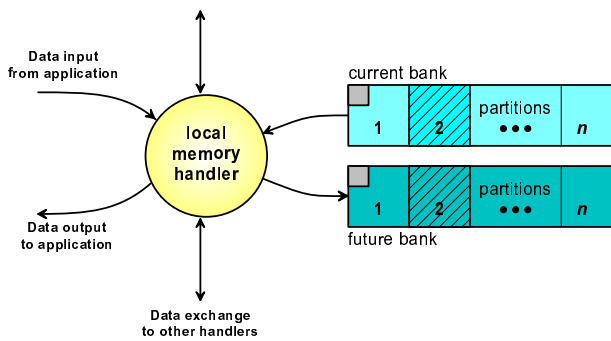


Figure 1. The local memory handler with its memory banks. The associated partitions are represented in a hatched way.

As a consequence of this definition, the local user context regards the local memory handler as the only interface to the distributed memory tool. The local memory handler and the attached user module are connected via an IPC mechanism based on shared memory, referred to in the following as *local link*. Commands to the distributed memory or messages from the memory will only flow between local user context and local memory handler. The tasks of the local memory handler are completely transparent to the user module.

4.1. The local memory handler and its tasks

As mentioned in the previous section, the local memory handler (see Fig. 1) is responsible for two banks of memory, i.e., the current bank, where it reads from, and the future bank, for all writing actions. Each bank is cut into N_{spat} partitions, where each handler is responsible for exactly one partition. This partition can be seen as the part of the redundant memory attached to the local user context, which is assigned to the local handler. We call this partition the one *associated* to the user module.

If a user module (i.e. a local user context) initiates to write data into its partition, this is done via a command to the local memory handler, which is sent via the connecting local link. The local handler then stores the data into the associated partition, and distributes them to the other local memory handlers residing on the other nodes. With this method the data are distributed as soon as they are received from the application.

For reading from the local memory handler there are several concurrent commands available. The common way is to request voted data from the local memory handler. In this case, one local handler receives a request for voted data from the attached user module. It then informs all other handlers and requests a voting service to vote among the replicas of the partition associated to the requesting user

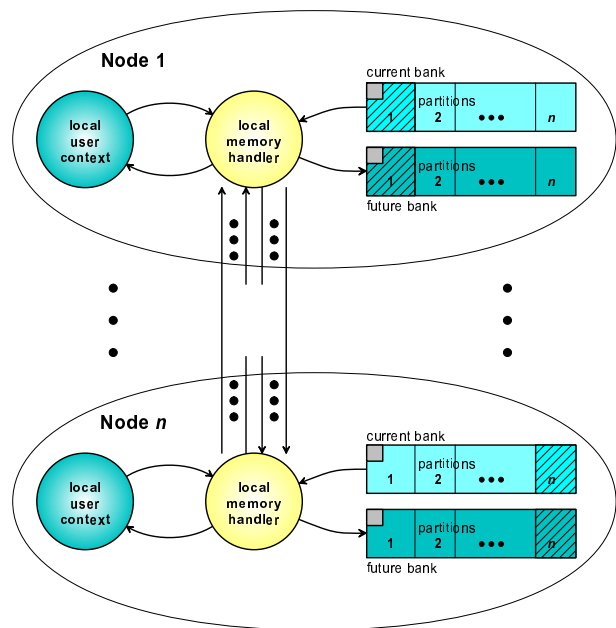


Figure 2. Structure of the Distributed Memory Tool. The associated partitions are represented hatched.

module. The result of the voting is provided to the calling user module as result of the read action. The kind of voting is user definable among those treated in [6]. If a voting task fails, a time-out system allows to regard such an event as the delivery of a dummy vote.

If the user application has a cyclic behaviour, such that the user modules on the different nodes all execute the same cycle, and under the hypothesis of each node serving exclusively the same set of tasks, then it is likely that the requests for reading may be processed more or less simultaneously on all nodes. In this case, this information can be used to synchronize the nodes via the set of local memory handlers and some data transfer between the nodes can be run in an optimized way.

Clearly the DMT is not aware of the logics pertaining to the stabilization mechanism, hence it is a task of the user application to inform the local memory handler about when to switch the banks (in the next section we will see how the temporal redundancy tasks take care of this). Normally this can be done once per cycle. The switching can also be connected with a checking phase, testing whether the current banks are equal on all nodes by means of an equality voting.

Clearly, both determining the role of each bank and switching these roles are crucial points for the whole strategy. In particular, these actions need to be atomic. A fast and effective way to reach this property is the use of two

| State | Flag A | Flag B | Current | Future |
|-------|--------|--------|---------|--------|
| 1 | 0 | 0 | A | B |
| 2 | 0 | 1 | B | A |
| 3 | 1 | 0 | B | A |
| 4 | 1 | 1 | A | B |

Table 1. Coding of the current/future bank flags

binary flags—one per bank—whose contents concurrently determine the roles of the banks. These flags have been protected by storing them in the bank themselves.

Table 1 shows how the coding of the flags in both banks is done. The idea is that, for changing from one state to another, *just one write action is needed in the future bank*. The current bank can therefore be regarded as being a read-only memory bank. If, e.g., the actual state is state 2, and we want to switch the banks, then it suffices to change the flag in the future bank, which is bank *A*. Changing Flag *A* from 0 to 1 brings the system to state 4.

4.2. API and Client side

Using function calls the user module is able to initialize the tool, to setup the net of local handlers, and to activate the tool. This process is done in several steps:

1. Each instance of a DMT is built up by declaring a pointer to a corresponding data structure:

```
dmt_t *dmt;
```

This data structure is the place to hold all information for one instance of the DMT on each node. So each user module that wants to use the DMT needs to declare a variable of this type.

2. In the next step, the DMT is defined and described. This creates a static map which holds all necessary information to drive the tool on each node. The following code fragment

```
dmt = DMT_open (id, VotingAlgorithm);
for (i = 0; i < NumHandlers; i++)
    DMT_add (dmt, i, PartitionSize,
            (i == MyNodeId));
```

when executed on every node where the DMT is intended to run, sets up a local memory handler to be used by the DMT. The voting algorithm can be selected by the user via a kind of call-back function.

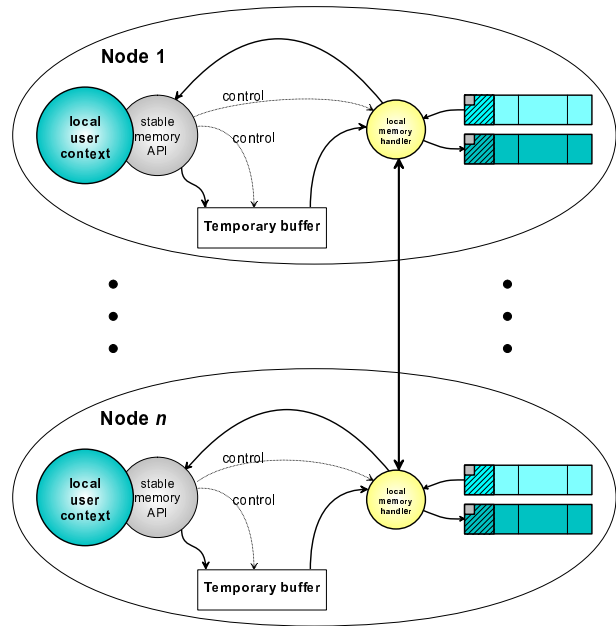


Figure 3. Structure of the DSS Tool.

3. After definition and description of the DMT, the latter has to be started to set up data structures and threads. This activation is done via function

```
int DMT_run (dmt_t *dmt);
```

This function simply spawns the local memory handler thread, after having checked the consistency of the structures defined in the description phase. All allocation of memory is done in the handler itself.

5. The DSS Tool

As already mentioned, the DSS tool builds on top of the DMT. While the latter is used for the management of the spatial redundancy (see Fig. 3), the DSS takes care of the management of the temporal redundancy strategy. On each node the writing requests to the DSS are done into a temporal redundancy buffer, which holds a user definable number N_{time} of copies of the last inputs. As the temporal buffers are only handled locally, this fits well to the concept of local memory handler. The DSS module performs a voting on the contents of the temporal buffers and, if this voting is successful, the result is fed into the DMT. The DSS handles all accesses as well as the temporal voting transparently of the local user context.

5.1. Algorithms

The DSS Module as a whole gives each local user context a combination of temporal and spatial redundant mem-

ory buffers, which are able to keep the local state variables. We call the *context family* associated to a DSS the set of all local user contexts that store data in an instance of that DSS.

The DSS completely hides the memory handling and the handling of the DMT, so that the user only needs to write to and read from the memory, all other handling is done transparently.

The following steps are executed automatically:

1. The new data are written into the temporal buffer.
2. Using internal flag values the current and the future banks are determined. This is done within the DMT, therefore only the flag values of the spatial redundancy buffers are used.
3. A voting on the temporal buffer takes place. If the temporal buffer is stabilized, i.e. the voting is positive, the content of the temporal buffer is stored into the spatial redundancy buffers, i.e. the respective calls to the DMT are done.
4. If the evaluation of the internal flags allows it, the DMT switches the memory banks.
5. A voting is applied to evaluate the output of the current bank in the distributed memory. Such output is returned to the calling local user context.

5.2. API

The DSS is identified via a control block structure. It must be seen in connection with the DMT—in fact it is a kind of front end to this tool, which provides additional functionality. The steps to be performed to set up the DSS are similar to those elaborated for the DMT:

1. Each instance of a DSS Tool is build up by declaring a pointer to a corresponding data structure:

```
smCB_t *sm;
```

Each local user context that is member of the associated context family needs to declare a pointer to a variable of type `smCB_t`.

2. Similarly to the DMT the DSS Tool has to be defined and described. With the `SM_open` statement a local instance of the is created. In addition to the parameters of the `DMT_open` statement some parameters regarding the temporal redundancy are passed to this statement.

```
sm = SM_open (id, Ntime,
              TemporalVotingAlgorithm,
              SpatialVotingAlgorithm);
```

```
for (i = 0; i < Nspat; i++)
    SM_add(sm, i, PartitionSize,
           (i == MyNodeId));
```

The above code sets up an instance of the DSS tool, assumed it is called in every local user context, which wants take part on the tool.

3. Up to now the DSS Tool is defined and described, i.e. the structure is set up, but no instance has been installed, no memory has been allocated and no handler for the spatial redundant memory is started. To do this the tool must be activated. This is done via the function `SM_run()`. This function allocates the temporal redundancy buffers, initializes the variables, and activates the attached Distributed Memory Tool using the function `DMT_run()`. The DSS can only be started up if all local user contexts belonging to its context family call `SM_run()` at the same point in their start-up phase.

At run-time, the DSS Tool is controlled via two functions, which read the data from the application or provide stabilized and voted data to the application:

```
int SM_write (smCB_t *MyCB, void *SM_in);
int SM_read (smCB_t *MyCB, void *SM_out);
```

The user provides data to the DSS via function `SM_write()`, which is then responsible for handling of these values. This function is used to input the local data of a local user context to the DSS. The parameters submitted to the function describe the DSS control block structure (`MyCB`) and the address of the data to be copied into the DSS (`SM_in`). When the function is returned, all data provided to the function using the `SM_in` pointer are copied out of this memory location into the temporary buffer of the DSS. The `SM_read()` function writes the local data of the calling local user context back to the address submitted through the pointer `SM_out`. The DSS control block structure `MyCB` is used to identify the DSS.

Switching from the current to the future bank is achieved by means of the following procedure:

- Determine the current meaning of the banks from their internal flags.
- Write data into the specified partition(s) of the future bank of the memory.
- Output data via a voting between the distributed copies of the current bank.
- If the contents of the future banks of all nodes are equal, switch the role of the memory banks.

Further information on this can be found in [4].

5.3. Fault Model

The overall strategy implemented in the DSS allows to mask a number of transient faults resulting in:

- an erroneous input value
- errors affecting the circular buffer,
- errors affecting the temporal redundancy modules,
- errors affecting the flag values,

occurring during the execution cycle, or caused by an external disturbance, or a wrong flag value, etc. These are tolerated either through the voting sessions (temporal redundancy) or are masked via the periodic restarts which invalidate the current cycle. In this latter case this is therefore perceived as a delay of the stabilized output. The same applies when a fault affects the phase of determining the current bank, or faults occurring during the voting among temporal redundancy modules, or faults affecting the spatial redundancy modules. More details on this can be found in [4].

Tolerance of permanent faults resulting in node crashes is achieved by using the DSS as a dependable mechanism compliant to the recovery language approach, a novel fault tolerance structuring technique introduced in [2] and discussed in [1]. As explained in the cited works, this approach exploits a high level distributed application (the TIRAN backbone) and a library of error detection tools in order to detect events such as node and task crashes. User defined error recovery actions can then be attached to the error detection events so to trigger corrective actions such as, e.g., reconfiguration of the DSS tasks.

6. Conclusion

In the above we have described a software system implementing a data stabilizing tool to be placed in highly disturbed environments like those typical of sub-station automation. Due to its design, based on a combination of spatial and temporal redundancy and on a cyclic restart policy, the tool proved to be capable of tolerating transient and permanent faults and to guarantee data stabilization. Initially developed on a Parsytec CC system, the tool has been then ported to several runtime systems. It is being successfully tested at ENEL where it is now compliant to their control applications and is being integrated within the cyclic-restart strategy adopted for these applications. Ongoing experience and validation activities will allow to gain additional confidence with the novel solution at ENEL and will culminate in its adoption as a replacement for the old hardware equipment and in the analysis of its re-use in a wide class of mission-critical automation applications.

Acknowledgements. This project is partly supported by the IST project “DepAuDE”. Geert Deconinck is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (FWO).

References

- [1] V. De Florio. *A Fault-Tolerance Linguistic Structure for Distributed Applications*. PhD thesis, Dept. of Electrical Engineering, Katholieke Universiteit Leuven, October 2000.
- [2] V. De Florio, G. Deconinck, and R. Lauwereins. The recovery language approach for software-implemented fault tolerance. In *Proc. of the 9th Euromicro Workshop on Parallel and Distributed Processing (Euro-PDP'01)*, Mantova, Italy, February 2001. IEEE Comp. Soc. Press.
- [3] DEC. *CS_Q66E Alpha Q-Bus CPU module: User's Manual*. Digital Equipment Corp., 1997.
- [4] G. Deconinck, O. Botti, F. Cassinari, V. De Florio, and R. Lauwereins. Stable memory in substation automation: a case study. In *Proc. of the 28th Int. Symposium on Fault-Tolerant Computing (FTCS-28)*, pages 452–457, Munich, Germany, June 1998. IEEE Comp. Soc. Press.
- [5] B. W. Johnson. *Design and Analysis of Fault-Tolerant Digital Systems*. Addison-Wesley, New York, 1989.
- [6] P. R. Lorczak, A. K. Caglayan, and D. E. Eckhardt. A theoretical investigation of generalized voters for redundant systems. In *Proc. of the 19th Int. Symposium on Fault-Tolerant Computing (FTCS-19)*, pages 444–451, Chicago, IL, June 1989.
- [7] TXT. *TEX User Manual*. TXT Ingegneria Informatica, Milano, Italy, 1997.