Effective Fault Treatment for Improving the Dependability of COTS- and Legacy-based Applications

A. Bondavalli*,
Dipartimento di Sistemi e Informatica,
Universitá di Firenze,
Via Lombroso 6/17, 50134 Firenze, Italy
a.bondavalli@dsi.unifi.it

S. Chiaradonna
ISTI-CNR
Via A. Moruzzi 1, Loc. S. Cataldo, 56124 Pisa, Italy
Silvano.Chiaradonna@cnuce.cnr.it

D. Cotroneo, L. Romano
Dipartimento di Informatica e Sistemistica
Universitá di Napoli Federico II
Via Claudio 21, 80125 Napoli, Italy
(cotroneo, lrom)@unina.it

Abstract

In this paper, we present a complete architecture suitable for improving the dependability of a wide class of distributed systems consisting of COTS components and Legacy systems. The paper advocates the need for careful diagnosis and damage assessment, and for precise and effective recovery actions, specifically tailored to the affecting fault and/or to the extent of the damage in the affected unit. In our proposal, threshold-based mechanisms are exploited to trigger alternative actions. The design and implementation of the resulting solution is illustrated with respect to a case study. This consists of a distributed architectural framework which replicates an application built from COTS components and Legacy systems. Replication and voting are used for error detection and masking. Dependability analysis has been conducted via combined use of direct measurements and analytical modeling.

Keywords: Legacy Systems & COTS Components, Fault Diagnosis & Treatment, Fault Injection, Modeling & Evaluation, Performability.

1 Introduction

We are witnessing the construction of complex distributed systems, which are the result of the integration of a large number of components, including COTS (Commercial Off-The-Shelf) components and Legacy systems. While there is agreement about the meaning

*Contact author

of the term COTS, it is worth defining what we mean by Legacy system. By Legacy system it is intended here a software program for which maintenance actions consisting in modifications to the source code are either impossible or prohibitively costly. This may be due to a variety of reasons, including:

- the component is written in a programming language which has become obsolete, as compared to the rest of the technologies used by the enterprise to develop its business applications. As an example, a COBOL program is a Legacy system in an enterprise environment where web oriented technologies are being massively used;

- the component is not well documented. As an example, it has been modified over the years by different people, who are not available anymore and/or who have not adequately reported in the software documentation the modifications they have performed.

COTS- and Legacy-based distributed applications are being used to provide services, which have become critical in our everyday life. It is thus paramount that such systems be able to survive failures of individual components, i.e. they must provide some level of (reduced) functionality also in the presence of faults. To this aim, effective fault tolerance strategies, specifically tailored to COTS- and Legacy-based applications, need to be revisited. More precisely, mechanisms and strategies to implement fault tolerance functions have to be tuned, to account for the differences between COTS and legacy-based applications and traditional safety-critical systems. Several proposals have been made, which allow to build dependable systems by integrating COTS and legacy components, as detailed in section 2. These proposals mainly concentrate on error processing, typically by replication (in a variety of flavors), possibly supported by facilities such as distributed membership of replicas, and/or reflective programming. However, little attention has been paid to the problem of maintaining system health, and of preserving tolerance capabilities.

Although diagnosis has been extensively studied, its application to COTS- and Legacy-based systems raises a variety of issues which have not been addressed before. Such issues stem from a number of factors, which are briefly described in the following:

- First, the designer (system integrator) has limited control over individual components and subsystems, since for most of them the internal design is not known.

- Second, COTS and legacy components are heterogeneous, whereas the targets of traditional diagnosis, typically the components of safety critical systems, are – to a large extent – homogeneous.
- Third, diagnostic activities must be conducted with respect to components which are large grained, whereas traditional applications (such as safety critical control systems) typically consist of relatively fine grained components.
- Fourth, repair or replacement of system units is costly and in some cases not possible at all.

In a COTS- and Legacy-based infrastructure, diagnosis must thus be able to assess the extent of the damage in individual components, so to carefully identify fault treatment and system reconfiguration actions, which are most appropriate, and when such actions are to be performed. To this aim, it is paramount that data about error symptoms and failure modes be carefully gathered and processed. One shot-diagnosis is inadequate: an approach is needed, which collects streams of data and filters them by observing component behavior over time. Several heuristics, based on the notion of threshold, have been proposed and have effectively been applied for diagnosis in many fields [20, 21, 22, 23], and in particular to highly available information & communication systems, as discussed later. Such kind of diagnosis has been applied to real-time systems [22, 34]. For such systems the collection and processing of data streams must not disrupt the real-time properties. A thorough analysis of how to effectively manage these data, fulfilling the requirements, is in [13].

In our past research, we addressed issues related to goals and constraints of a diagnostic sub-system based on the concept of threshold, which must be able to: *i*) understand the nature of errors occurring in the system, *ii*) judge whether and when some action is necessary, and *iii*) trigger the recovery/reconfiguration/repair mechanisms/staff to perform the adequate actions to maintain the system in good health [13].

The novel contribution of this paper is a methodology and an architectural framework for handling multiple classes of faults (namely, hardware-induced software errors in the application, process and/or host crashes or hangs, and errors in the persistent system stable storage) in a COTS- and Legacy-based application by using an evidence-accruing fault tolerance manager to choose and carry out one of multiple fault recovery strategies, depending upon the perceived severity of the fault. A case study system has been implemented and

benchmarked using a performability benchmark. In particular, the study focused on the best threshold values for the number of faults needed before triggering specific recovery mechanisms. The effectiveness of the suggested approach is evaluated through combined use of direct measurements on the system prototype and analytical modeling. The use of a fault injection on a real system prototype allowed us to derive realistic fault models (by tracking error propagation through individual system layers), and to extract relevant system parameter values (which were used to populate an analytical model of the system).

The rest of the paper is organized as follows. Section 2 provides an overview of relevant previous work. Section 3 describes the conceptual architecture of the replication framework, i.e. the main components and their interactions, emphasizing the fault treatment approach that we propose. Section 4 specializes the conceptual architecture of the replication framework taking into account the characteristics of a specific application which we used as a case study. Section 5 describes the Stochastic Activity Network (SAN) model that was used to evaluate the system. Section 6 presents the results on parameter tuning and overall system dependability that were obtained via combined use of direct measurements and analytical modeling. Section 7 concludes the paper with final remarks and lessons learned.

2 Related Research

This section provides an overview of the research conducted in the field of fault tolerant distributed systems which include COTS components and Legacy systems. The analysis clearly indicates that most projects proposed replication-oriented architectural frameworks focusing on error processing, but addressed fault-treatment only to a limited extent.

A great deal of research has been conducted on providing support for dependability to existing distributed applications. These proposals differ from one another in many aspects, including the nature of the fault tolerance mechanisms (hardware, software, or a combination of the two), and the level of transparency to the application level (application aware/unaware approach).

Many projects have been undertaken, both in the academia and in the industry, which provide fault tolerance to CORBA applications, by incorporating fault tolerance facilities in the ORB and/or in additional software layers. AQuA [3], Eternal [31], IRL [5], and

OGS [12] are examples of such projects. AQuA provides replication groups, which communicate using connection groups [3]. The Eternal system adds fault tolerance to CORBA applications by replicating objects and incorporating a number of mechanisms to maintain replica consistency [31]. The IRL relies on a multi-tier scheme and on a set of protocols, mechanisms and services which allow a CORBA system to handle object replication [5]. OGS provides an active replication scheme implemented as a CORBA service, which is based on a group communication facility [12]. In addition to the proposals related to CORBA, several research studies have been conducted on providing support for dependability to existing applications via distributed architectural frameworks which do not rely on any specific middleware product. Among these, it is worth mentioning Chameleon [4], FRIENDS [8], and DELTA-4 [10]. Chameleon provides fault tolerance through a wide range of software implemented error detection and error recovery mechanisms for both applications and Chameleon entities. The FRIENDS system provides mechanisms for building fault tolerant applications through the use of libraries of meta-objects. The Delta-4 project provides an open, fault-tolerant, distributed computing architecture.

In [7], the authors present a middle-tier based architectural framework for leveraging the dependability of legacy applications, transparently to the clients. The conceptual architecture of such a framework is -to a large extent- independent of the specific characteristics of the target application, as well as the enabling technologies. A prototype was deployed using CORBA technology.

Despite such a large body of research, distributed systems made of COTS components and Legacy systems exhibits some characteristics which need further investigations. In particular, the analysis clearly indicates that most proposals, although effective in leveraging the dependability of the target applications, addressed fault-treatment only to a limited extent. In particular, accurate fault diagnosis, which allows more effective fault treatment actions, can play a key role for further improving the dependability of COTS- and Legacy-based applications. Diagnostic techniques applied to such systems have to carefully assess component status or the extent of component damage, so to carefully identify the need for reconfiguration, and for treatment actions. One-shot collection of a syndrome, typical of traditional diagnostic models, is not possible; data on components behavior have to be collected, and filtered over time. In the literature, some of such approaches have been proposed mainly for the identification of the nature of hardware faults

(transient, intermittent or permanent). These can be broadly classified in two categories: i) techniques designed to support human intervention [18, 19], to be applied off-line to error logs, and ii) algorithm-based, automatic mechanisms to be used on-line. Techniques in the latter category are all based on a notion of thresholding the error frequency to diagnose components hit by a permanent fault [20, 21, 22, 23]. A more extended description of the two categories and of the various proposed solutions is in [9]. Among the heuristics based on the concept of threshold, the α-count family of mechanisms [21, 9] appears to be particularly interesting for our purposes, due to the clear and simple mathematical characterization and to the thorough analysis already conducted.

Very briefly, the α-count processes information about erroneous behavior of each system component, giving a smaller and smaller weight to error signals as they get older. A score variable α_i is associated to each not-yet-removed component i to record information about the errors experienced by that component. α_i is initially set to 0, and accounts for the L-th judgement as follows:

- $\alpha_i(L) = \alpha_i(L-1) + 1$ *if channel i is perceived as faulty during execution L,*
- $\alpha_i(L) = K \cdot \alpha_i(L-1)$ $(0 < K < 1)$ *if channel i is perceived as correct during execution L.*

When $\alpha_i(L)$ reaches a given threshold α_T, component i is diagnosed as failed and a signal is raised to trigger further actions (error processing or fault treatment). Proper tuning of the values of parameters K and α_T is crucial for the effectiveness of the mechanism. The optimal tuning of these parameters depends on the expected frequency of errors and on the probability of correct judgements of the error signaling mechanism. The analysis performed in [9] has clearly shown the trade-offs between delay and accuracy of the diagnosis and thoroughly discussed ways to tune these parameters for maximizing performance.

3 Replication Framework and Fault Treatment Strategy

In this section, we describe the conceptual architecture of the replication framework, i.e. the main components and their interactions, emphasizing the fault treatment approach that we propose. The role of individual components is -to a large extent- independent of the specific characteristics of the target application (i.e., the COTS- and Legacy-based application which is replicated), whereas crucial system settings (such as the values of the thresholds and the number and the nature of the recovery actions) must take into account

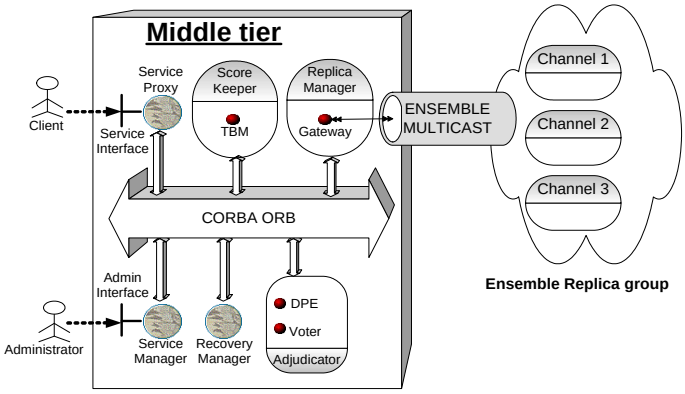

Figure 1: Overall System Architecture

application and environment factors, including fault assumptions and failure propagation mechanisms. Section 4 describes the deployment of the proposed architecture with respect to a specific case study application.

The architectural framework, depicted in Figure 1, is a three-tier system. The first tier consists of a client which uses the services provided by the third tier, consisting of a replicated COTS- and Legacy-based application (individual replicas are referred to as channels in the rest of the paper). The middle tier is in charge of handling distribution and fault tolerance related activities. The objective is to contrast the effects of faults hitting the back-end channels. This goal is achieved by replicating the legacy and COTS based application according to a Triple Modular Redundancy (TMR) scheme. Since the replicated legacy and COTS components are identical, they could exhibit common mode failures. In order for replication to be effective, a key issue, that is limiting the probability of common mode failures, must thus be addressed. To this end, diversity has to be enforced when the legacy systems and COTS components are integrated in the replication framework. This goal is best achieved if different mechanisms/settings/components are enforced at multiple architectural levels. Section 4.3 illustrates possible measures with respect to our case study application. The middle tier is in charge of handling distribution and fault tolerance related activities, and in particular: error detection, data filtering, error diagnosis, error masking, and fault treatment. It mediates the service by hiding replication to the users and providing management facilities (distribution of service requests, voting upon individual results, and redundancy management). In particular, the Middle Tier handles state modifications by broadcasting them to all active replicas. Connectivity between the

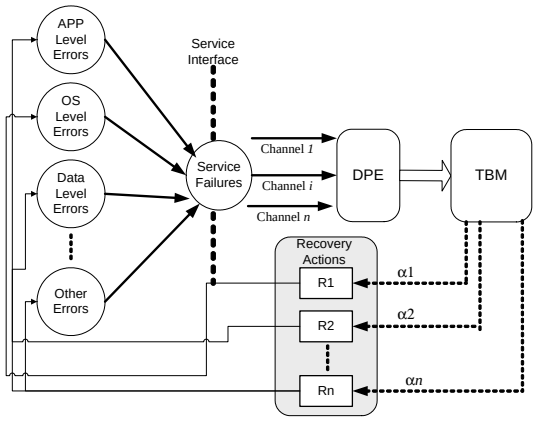

Figure 2: Error accumulation and recovery actions

individual components is provided by CORBA [11], specifically VisiBroker version 4.1. In the following the main components of the middle-tier are briefly described.

Adjudicator - The Adjudicator consists of two main components; i.e., the Voter and the Data Processing & Exchange (DPE). The Voter is in charge of selecting a presumably correct result out of those provided by the channels (error masking). It may support several adjudication strategies, but in this study it is configured to perform majority voting. Based on the results of voting, the adjudicator provides error detection information to the DPE. The DPE filters the error detection data produced by the voter, and delivers it to the Threshold Based Mechanism (TBM) component.

Score Keeper - The Score Keeper includes the TBM, which is in charge of performing diagnostic activities (fault diagnosis) and of triggering recovery actions, individually tailored to specific kinds of errors (fault treatment). As to diagnosis, the TBM monitors the status of the channels and decides whether it is useful to keep individual channels online or it is more convenient to perform some recovery actions. This is a crucial activity, since excluding replicas or performing heavy and costly recovery actions, at the very first occurrence of an error is too a simplistic approach, which may well have a negative impact on the performability of the overall system. The specific algorithm implemented by the TBM is called α-count, and it is described in detail in [9]. Multiple instances of α-count monitor each channel and are used to choose among a number of alternative recovery actions. This is illustrated in figure 2.

Recovery Manager – The Recovery Manager is in charge of performing the recovery actions triggered by the Score Keeper. The basic idea is to trigger the least costly and

most effective recovery action based on an estimate of the nature of the fault affecting the channel, in order to limit error accumulation and ultimately contrast the propagation of failures from the back-end to the service interface. Figure 2 illustrates the abstract and conceptual way errors do accumulate in one channel and how the specific recovery actions (detailed in section 4.3) are triggered by the components of the diagnostic subsystem. Note that, in many cases, given two recovery actions R_i and R_j (R_j has a greater cost than R_i), the execution of R_j also fixes inconsistent states which could be cured by R_i. Due to this consideration, recovery actions are triggered according to a least costly first strategy. This is achieved by piloting the actions using multiple α-counts for each channel.

Replica Manager – The Replica Manager coordinates and supervises all activities necessary for ensuring that individual replicas are consistent. To this end, it uses the facilities provided by the `Gateway` object, which handles data exchanges between the Voter and the COTS-based application. The Gateway performs two key activities: i) purifies the data from application-specific and platform-related dependencies, thus avoiding that system failures occur due to interaction problems (interaction faults are typically caused by mismatches or incompatibilities between the legacy applications and the COTS platforms and software components), and ii) provides reliable multicast support. Limiting the probability of interaction faults is a key issue.

Service Manager – The Service Manager is in charge of configuring all other components. It provides functions to customize the behavior of individual objects (such as the specific adjudication strategy which must be used for building the reply to be sent to the client), and to set system configuration parameters (such as the number of threads in the thread pools).

Service Proxy – The Service Proxy encapsulates the services provided by the legacy application and exports them via the Service Interface, thus making such (enhanced) services available to the clients.

4 Deployment Scenario with Respect to a Case Study Application

In this section, we specialize the general architecture of the replication framework, which was illustrated in the previous section, taking into account the characteristics of a specific application which we used as a case study. In particular, a Failure Mode Error Analysis

was conducted in subsection 4.2 which allowed to identify effective fault treatment actions, i.e. recovery actions tailored to specific classes of faults affecting the target application.

4.1 Case Study Application

The application used as a case study is a distributed one, which encompasses a legacy component consisting of legacy code written in C, which uses a COTS DBMS, namely PostgreSQL for stable storage facilities, as shown in figure 3. The main objectives of the resulting system are:

- to make the application available to remote users which are given the possibility to access the services (perform query and updates operations) through a CORBA infrastructure;
- to make it a long running service, available for long time of continued (apart from outages due to recovery and maintenance) operations;
- to leverage the dependability of the service, beyond the level originally provided by the legacy component.

The DBMS is seen as a black box which exposes an X/Open XA compliant interface. The application runs on a Unix-like kernel, on top of a commodity PC or a workstation. Services can be grouped in two main categories, namely *Queries* and *Updates*. The legacy application reads data from the database to the application level cache, which is implemented as a linked list of nodes which stores temporary data (result sets) to/from the database, as illustrated in Figure 3. The nodes of such a list consist of an `el` field which holds the stored value, and a `ptr` field which points to the next node in the list. The application allocates new memory as it needs some. The address of the first node of the list is stored in a pointer variable. Such a pointer is allocated in the process stack area, whereas the list nodes are allocated in the process heap area. The data base physical files are stored on disk.

4.2 Faults and Failure modes

The following types of faults have been considered:

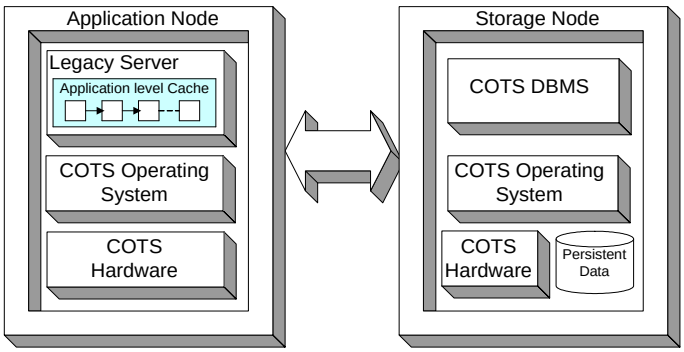

Figure 3: Case-study legacy application

- Hardware faults - These are faults stemming from instabilities of the underlying hardware platform, which manifest as errors at the software level [14]. We limit our attention to intermittent and transient faults, since these are by far predominant [15];

- Software aging faults– Software aging is a phenomenon, usually caused by resource contention, which can lead to server hangs, panic, or crashes. Software aging mechanisms include memory bloating/leaks, unreleased file locks, accumulation of unterminated threads, shared-memory-pool latching, data corruption/round off accrual, file-space fragmentation, thread stack bloating and overruns [1]. Recent studies [24] have shown that software aging is a major source of application and system unavailability.

Logic level faults, i.e. erroneous coding of the application's functional specifications, are not considered. This is not a simplistic assumption. Indeed, this choice is motivated by the fact that logic level faults in COTS components are rapidly detected (and fixed) due to the availability of a large number of installations, which provide a large amount of field data about component failures. As to legacy components, although a large installed base is not available, logic level faults have been detected and fixed during the years long operation. In fact, in the typical scenario, legacy software has been thoroughly tested and debugged, and the vast majority of such faults, which manifest as a violations of the application's functional specifications, have been detected and fixed over the years. Conversely, software aging faults may still be present in the code base. As a result, the legacy program may exhibit memory leakage problems. A memory leak can go undetected for years if the application and/or the system is restarted relatively often (which might well have been

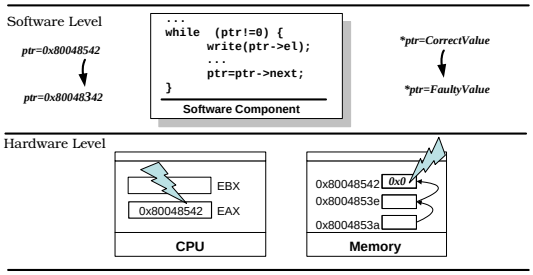

Figure 4: Propagation of Hardware Faults to Application Level.

the case of the legacy application). Communication errors are not considered either. Such errors occur when data gets corrupted while traversing the network infrastructure. Indeed, unless "leaky" communication protocols are adopted, it is fairly unlikely that this kind of errors remains undetected and is not handled by the network protocol facilities (a leaky protocol is a protocol that allows corrupted information to be delivered to the destination).

A thorough analysis has been performed to understand how hardware faults and software aging faults propagate from lower levels of the system to the application level. The approach taken for such a Failure Mode Error Analysis (FMEA) has been to conduct a fault injection campaign. An additional -and a fundamental- benefit of the measured based campaign has allowed to extraction of values for populating the parameters of the analytical model of the system which has been built to facilitate an end to end analysis of the system behavior.

Propagation of hardware faults to the software level is illustrated in Figure 4. The Figure illustrates how a transient hardware fault hitting a memory location or a CPU register may lead to the corruption of the nodes of the application level cache of the case study legacy component. Since these faults manifest at the application level as wrong values of the `el` or `ptr` field, we were able to evaluate the effects of hardware faults by injecting faults at the software level. Injections were performed using the NFTAPE tool [16] and the GNU debugger (GDB).

We performed the following experiments:

1. *Data corruption.* The injection consisted of modifying the value of the $i-th$ `el` field. At the application level, this kind of fault resulted in a corrupted entry being written to the database (in the case of an update) or exposed to the system external interface (in the case of a query). In the case of an update, the error thus propagated

to the stable storage.

2. *Un-aligned Pointer.* The injection modified the value of the $i-th$ `ptr` field with a new value in the process address space. The effect of this injection was that the application returned one or more random records before it eventually crashed. Again, in the case of an update, the error propagated to the stable storage. More precisely, the sublist starting from record i-th became erroneous, being some of its records not up to date.

3. *Aligned Bridging Pointer.* The injection modified the value of the $i-th$ `ptr` field with a new value which is the address of the $(i+k)-th$ node. Although the application did not crash, this kind of fault resulted in a query or update which operated on a wrong number of records. At the application level, this kind of fault resulted (in the case of an update) in a truncated list being written to the physical database (i.e. in possibly several items being deleted from the database) or (in the case of a query) in a truncated list being read from the database.

4. *Aligned Looping Pointer.* The injection modified the value of the $i-th$ `ptr` field with a new value which is the address of the $(i-k)-th$ node $(0 < k < i)$. In other words, the performed injection induced a loop in the list. This caused the application to loop infinitely as soon as a query or an update was issued. It is worth noting that the error did not propagate to the stable storage. However, it led to an application hang due to the fact that the application entered an infinite query/update loop.

As to software aging faults, injections were performed using GNU debugger (GDB) scripts, which forced the target application to behave as a "resource hog", i.e. to request system resources (such as memory, file locks, and thread mutexes) which were never released. We performed the following experiments:

1. *Memory leak.* The injection forced the application to skip memory deallocation instructions. This resulted in a larger and larger amount of memory being allocated to the faulty application. Initially, this led to an overload condition for the hosting node and manifested as a performance fault, that is a timeout failure of the application. Eventually, the Operating System denied the allocation of further resources

to the application. This resulted in a crash failure of the application, since a signal was generated by the Operating System and the application was killed.

2. *Blocked thread.* Two kinds of injections were performed: i) we forced the application to skip mutex unlock functions, and ii) we forced a loop in a thread's code. Both experiments resulted in progressive depauperation of system resources, which led to performance faults.

4.3 Experimental Testbed and Settings

4.3.1 Recovery Actions

The FMEA described in subsection 4.2, as well as research conducted in the field of software aging and rejuvenation [1], has suggested three recovery actions for the legacy subsystems, each one tailored a specific class of errors:

1. Restart of the application ($R1$) - This action can cure inconsistent application level states (such as corrupted data structures), but it has no effect on errors which have propagated to the OS and the system stable storage.

2. Reboot of the host computer ($R2$) - A reboot restarts the operating system and all the service software (obviously, in such a case also the application software is restarted). This action can thus fix erroneous OS as well as service software states (such as locks due to badly handled race conditions). Again, it has no effect on errors which have propagated to the system stable storage.

3. Restoration of the data base ($R3$) - Since multiple copies of the data exist, attempts can be done to correct errors in the system stable storage. Details about the restoration procedure are provided later in this section.

Whereas the Restart of the application and Reboot of the host computer are self-explanatory, the Restoration deserves a brief explanation. The restoration procedure is as follows: the recovery manager reads binary data from the two replicas which are not been recovered, and compares the flows. If comparison is successful, bits are copied to the recovering instance. If a disagreement is detected, a third value is read from the recovering replica. Hopefully, it is possible to determine the correct value via majority voting. If this is not the case, we assume the system has failed, since no valid data is available. It must

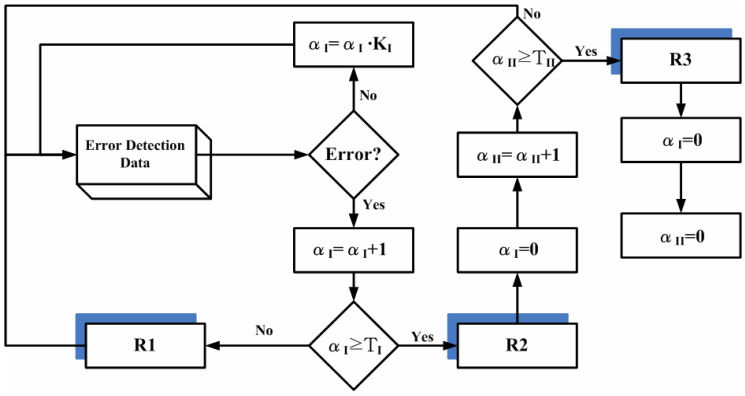

Figure 5: Fault-treatment logic

be remarked that the restoration procedure just described is of course only one of many possible algorithms, and it is not claimed here that this is the best alternative (the focus of this paper is not on database restoration techniques).

We explicitly note that R_2 (Application Restart) is in essence an escalation of R_1 (hardware reboot), while R_3 (Database recovery) is helpful for different failure modes.

4.3.2 Fault Treatment Logic

The resulting fault-treatment logic, which is illustrated in figure 5, makes use of two α functions and triggers recovery actions according to a least costly first strategy. Any error detected by the voter is signaled to the first α-count, which increases the score, whereas each success is used to decrement the score. In the case of an error, if the threshold is not reached, $R1$ is performed. When instead the threshold is reached, $R2$ is executed. The score of the first α-count is also reset and an error signal is sent to the second α-count, which increments its score. The score of the second α-count is never decremented (this choice is consistent with the assumption that normal operation does not correct corrupted data). Finally, when the threshold in the second α-count is reached, $R3$ is performed, and the scores of both α-counts are reset.

To provide reliable multicast support to the `Gateway` object, the Ensemble group communication facility [28] was used. Ensemble was attached to the `Gateway` object (at the one end) and to the server replicas (at the other end). The former task was straightforward: it was sufficient to link the Ensemble library to the Gateway code. The latter task was quite more complex. In fact, the legacy application came with a TCP/IP socket

based interface. TCP calls were intercepted and redirected to Ensemble. To this end, a virtual device driver was integrated in the kernel of the node which hosted the legacy server. This approach is to a large extent, i.e. apart from implementation level technicalities, independent of the specific characteristics of the operating system of the hosting node. For a thorough description of this technique, with respect to the Sun Solaris operating system, please refer to [29]. For improved performance, three threads, namely the `GatewayThreads`, parallelize the activity of the `Gateway` object.

4.3.3 Diversity

In order to limit the probability of common mode failures the following measures were adopted:

- At the hardware level, nodes with different CPUs (specifically, a PowerPC, a SPARC Ultra, and a Pentium CPU), equipped with different amounts of RAM (specifically 256, 512, and 1024 MBytes) were used.
- At the OS level, use was made of different versions of the Linux kernel (namely version 2.5 for the PowerPC, 2.2 for the SPARC, and 2.4 for the Pentium), and –most importantly– the kernels were compiled using different values for *usage limits* configuration parameters. In particular, it was made sure that none of the following parameters had identical values on any two machines: `RLIMT_CPU`, i.e. the maximum CPU time allowed for the process; `RLIMT_FSIZE`, i.e. the maximum file size allowed; `RLIMT_DATA`, i.e. the maximum heap size; `RLIMT_STACK`, i.e. the maximum stack size; `RLIMT_RSS`, i.e. the number of page frames owned by a process; `RLIMT_NPROC`, i.e. the maximum number of processes that a user can own; `RLIMT_NOFILE`, i.e. the maximum number of open files; `RLIMT_AS`, i.e. the maximum size of process address space.
- At the application level, individual nodes were configured with diverse background loads, i.e. we launched a diverse set of services on each node.

It must be remarked that the measures taken do limit the occurrence of common mode failures, since they have a direct impact on the failure modes of individual channels, as described in section 4.2.

5 Model of the System

This section describes the model defined for conducting an analysis of the dependability (or better performability, as will be motivated later) of the system. The model has been obtained as a set of interconnected Stochastic Activity Networks (SAN) [33] and then solved by simulation using the MOBIUS tool [26]. A complete description of SAN can be found in [33]. Very briefly, SAN are a variant of Stochastic Petri Nets (SPN), with a graphical representation consisting of places, timed and instantaneous activities, input and output gates. Activities are equivalent to transitions in SPN. The amount of time to complete a timed activity may be exponentially or non-exponentially distributed. Cases can be associated to activities (represented graphically as circles on the right side of an activity) and permit to model uncertainty upon completion of an activity. The use of gates permits a greater flexibility in specifying enabling conditions and completion rules than simple SPN.

Instead of developing unnecessary complex and large detailed models of the channels (accounting for fine-grain components like processes, data structures, OS layer, HW layer, etc.), we opted for modelling a channel as a relatively simple component, consisting of an application object, an OS component, and a database object. This approach allows to represent the three distinct kinds of errors which were identified in 4, namely Application Level errors (erroneous states of the application software), OS Level errors (erroneous states of the kernel and basic services), and Data Level errors (erroneous data which is stored in the data base).

Moreover, intermittent application or OS software aging errors have been assumed to have an increasing rate according to a lognormal (or Weibull) distribution [25], since this is consistent with the fact that the extent of the damage of the channels increases with time (if no recovery action is taken). In addition, transient hardware faults are assumed to have an exponential rate, characterized by the alternation of periods where normal fault rate is observed, and period with abnormal, higher rate. The duration of a period follows an exponential law (with normal periods quite longer than abnormal ones). Hardware faults manifest as application level errors as follows: the effects of hardware faults last for a number of service requests with a geometric distribution, i.e. at each service request there is a constant probability p_c of removing the effect of the fault. The restart

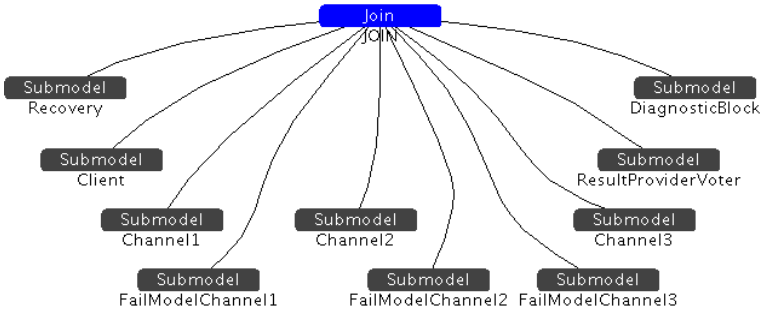

Figure 6: Composed Model of the System

of the application removes the effect of the hardware faults at the application level. The composed model of Figure 6 represents the hierarchical model of the system. It consists of ten logically distinct SANs (`Recovery, Client, Channel1, Channel2, Channel3, FailModelChannel1, FailModelChannel2, FailModelChannel3, ResultProviderVoter`, and `DiagnosticBlock`), connected together through common places by the `Join1` operation.

The submodel `Recovery` mimics system behavior as different types of recovery actions are taken. During recovery, the system does not serve requests. The `Client` submodel represents service requests, the status of the replies to the clients (correct, detected erroneous, undetected erroneous), and the number of replicated servers which are online. `Channel1`, `Channel2`, and `Channel3` represent the three channels and the associated GatewayThread processes. Submodels `FailModelChannel1`, `FailModelChannel2` and `FailModelChannel3` represent the failure behaviour of each channel.

The SAN `ResultProviderVoter` mimics the activities of the Gateway (which receives the result sets from the GatewayThreads and delivers them to the Voter), and of the Voter. The SAN `DiagnosticBlock` represents the behavior of the TBM.

In the remainder of this section three sub-models are described in detail.

Figure 7 depicts the SAN `Recovery`. The activity `Recovery` represents the deterministically distributed time of recovery, depending on the type of recovery action. For example, the time for reconstructing the database depends on the number of records in the database, represented by the number of tokens in places nRecords1_1 and nRecords1_0 (for the first channel), nRecords2_1 and nRecords2_0 (for the second channel), and nRecords3_1 and nRecords3_0 (for the third channel).

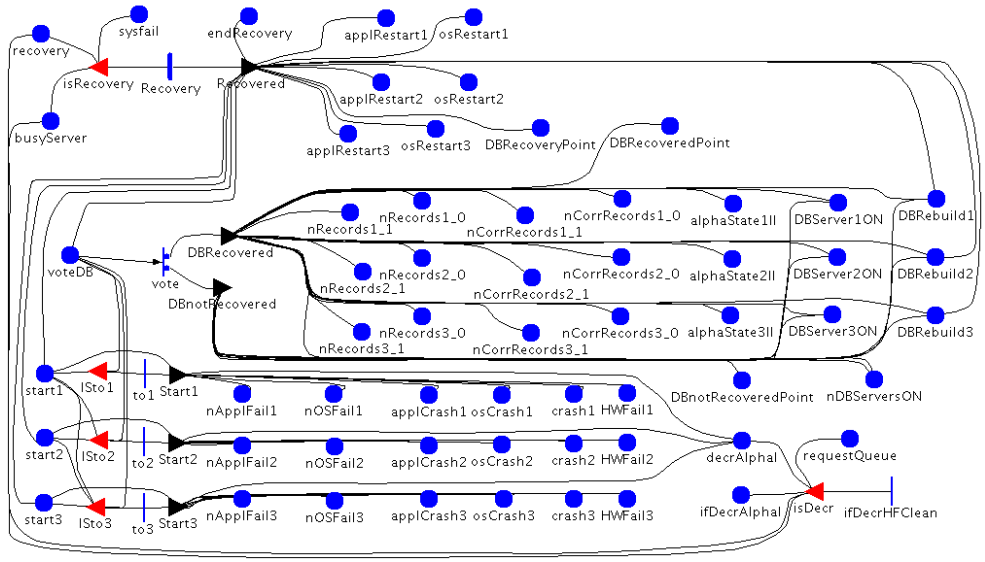

Figure 7: SAN "Recovery"

The activity Recovery is triggered by the `DiagnosticBlock` (which triggers the recovery by putting a token in the place `Recovery`) and the Client models (which remove a token from the place `busyServer` when the current request has been served). The `C` code in the output gate `Recovered` enables the marking changes due to the restoration of the data base or due to the restart of the application and of the OS of the three channels. The output gates `DBRecovered` and `DBnotRecovered` represent the marking changes for modeling the success or the failure of the restoration of the database, respectively. The two cases of the activity vote represent the probabilities of success/failure of the restoration of the database. These probabilities are a function of the number of correct records in the database, which is given by the number of tokens in places nCorrRecords1_1 and nCorrRecords1_0 (for the first channel), nCorrRecords2_1 and nCorrRecords2_0 (for the second channel), and nCorrRecords3_1 and nCorrRecords3_0 (for the third channel). The output gates `Start1`, `Start2`, and `Start3` represent changes in the marking to model the application and the OS restart in the three channels. Places `DBRecoveryPoint`, `DBRecoveredPoint`, and `DBnotRecoveredPoint` represent the event of occurrence, of success, and of failure of the restoration of the database, respectively. The termination of recovery actions results in a token being punt to `decrAlphaI`. This activates the `DiagnosticBlock` model (not shown), which is in charge of decrementing α-count values. To reduce the simulation time, this operation is not modeled upon delivery of every single signal from the DPE.

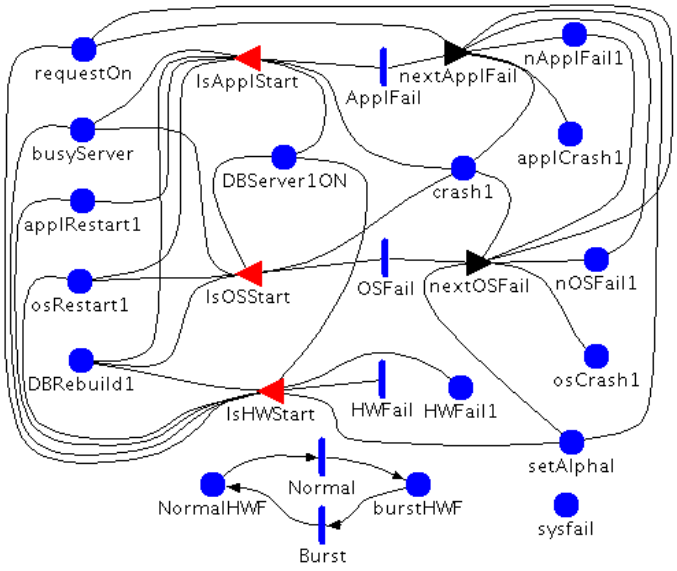

Figure 8: SAN "FailModelChannel1"

Instead, we consider time slices corresponding to a reduction in the value of α-count equal to `deltaAlphaI`. This parameter has been assumed equal to 0.1 in all simulations. If one chooses a greater value for this parameter, the simulation would be speeded up, at the cost of a worse approximation of the α-count mechanism. The behavior of the system during individual intervals (such as the number of requests which are served) is modeled using the average values of the request arrival time and of the service execution time. The input gate `isDecr` activates the submodel when failures occur (this is represented by a token in place `ifDecrAlphaI`).

Figure 8 depicts the SAN `FailModelChannel1`. The Lognormal and Weibull distributed activities `ApplFail` and `OSFail` represent the times to failure of the application and of the OS, respectively. After the i-th failure represented by the number of tokens in the places `nApplFail1` (for the application) or `nOSFail1` (for the OS), the time to the next failure is reduced by using a distribution with a mean equal to the original one divided by 2^i. When the number of tokens of the places `nApplFail1` or `nOSFail1` becomes greater than or equal to a given threshold N_{crash}, the application or the OS crash, respectively. `ApplFail` and `OSFail` are restarted with the original distributions after each restart of the application or of the OS, respectively.

The exponential activities `Normal` and `Burst`, represent the alternation of normal periods, whose expected duration is indicated by the parameter T_N, where the transient fault

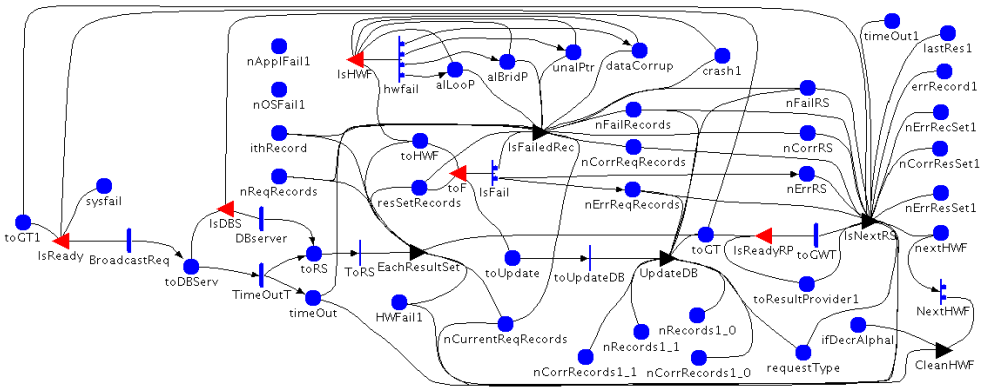

Figure 9: SAN "Channel1"

occurs with rate λ_N, and of abnormal periods, having expected duration T_B, characterized by a higher rate λ_B. The rate of the exponential activity `HWFail`, representing the time to failure of the application caused by an hardware fault, is λ_N or λ_B depending on the marking of the places `NormalHWF` and `burstHWF`. After the restoration of the database all the places are reset and the activities are restarted.

Figure 9 depicts the SAN `Channel1`. The activity `BroadcastReq`, which represents the deterministic broadcasting time, is activated by the Client if there is no token in the place `toGT`. Since the result of a reply can be very large, it is delivered in chunks of data (result sets). The current size of each result set is represented by the place `resSetRecords`. The activity `DBServer` represents the retrieval and the updating of the result, it is a timed activity deterministically distributed. This time is a function of the number of tokens in the places `nApplFail1` or `nOSFail1`. After the i-th failure of the application or of the OS, the time for providing the remaining result sets is multiplied by the factor 2^i, being i the number of errors. The deterministically distributed activity `TimeOutT` represents the timeout for the delivery of a result set. Upon completion of the `DBServer` activity, the output gate `eachResultSet` sets the number of current records (represented by the place `nCurrentReqRecords`) and, if an hardware fault has occurred (there is one token in the place `HWFail1`), it enables the instantaneous activity `hwfail` by inserting a token in place `toHWfail`. The four cases of activities `hwfail`, representing the probabilities of the fault events described in section 4.2 (Data corruption - Case 1, Un-aligned Pointer - Case 2, Aligned Bridging Pointer - Case 3, and Aligned Looping Pointer - Case 4) are as follows:

$$P(Case1) = \frac{N_{rr}(L_{rec} - L_{addr})}{(N_{rr}L_{rec} + L_{addr})},$$
$$P(Case2) = \frac{(1 - P(Case1))(L_{mem} - N_{rr} - 1)}{L_{mem} - 1},$$
$$P(Case3) = \frac{(1 - P(Case1))(N_{rr} - i_e + 1)}{L_{mem} - 1},$$
$$P(Case4) = \frac{(1 - P(Case1))(i_e - 1)}{L_{mem} - 1},$$

where, N_{rr} is the number of records for each request, L_{rec} is the number of bytes for each record, L_{addr} is the length of an address (in bytes), and L_{mem} is the number of addresses for a memory,i_e is the index of the corrupted pointer.

The probability of Case 1 is computed as the ratio of the probability that an error hits a specific record (which is proportional to the size of the memory area containing data for that record) to the probability that the error falls in the memory area of the application-level cache (which is proportional to total size of the application level cache). The probability of Case 2 is the probability that a specific pointer is corrupted by a fault and points to a non valid memory address (thus leading to a crash). This is given by the ratio of all possible memory addresses diminished by the total number of valid memory addresses ($L_{mem} - N_{rr} - 1$) to the total number of non valid memory addresses. The probability of Case 3 is the probability that a corrupted pointer points to an item which follows in the record list. In the computation we have also considered the probability that the corrupted pointer points to `null`. Formula for Case 4 represents the probability that a corrupted pointer points to an item which precedes it in the record list. For each result set the instantaneous activity `IsFail` generates the number of correct and erroneous records (`nCorrReqRecords` and `nErrReqRecords`) and the number of records (`nFailRecords`) affected by a failure occurred in the meantime. Correct and erroneous records are randomly chosen by the two cases of the activity `IsFail` by using the number of records and correct records in the DB, represented by the places `nRecords1_1`, `nRecords1_0` and `nCorrRecords1_1`, and `nCorrRecords1_0`, respectively. A failure occurs if a token has been inserted in places `timeOut`, `crash1`, `dataCorrup`, `unalPtr`, `alBridP` or `alLooP` and the number of tokens in the place `ithRecord` represents the index of the corrupted pointer described in section 4.2, in the case of fault occurrence. Then, the output

gate UpdateDB records relevant information regarding the system stable storage. As soon as the activity toGT completes, a result set is provided to the ResultProvider (the output gate IsNextRS inserts one token to the place toResultProvider1 and reduces the number of tokens to nCurrentReqRecords, representing the current number of records have not yet delivered to the client). A new result set (if it exists, i.e. there is at least one token in the place nCurrentReqRecords) can be retrieved (IsNextRS inserts a token to the place toGT1). Places nFailRS, nCorrRS, and nErrRS represent the number of failed, correct, and erroneous records, respectively, which are read from the DB for each result set. These places are reset when the simulation of the handling of a new result is started. The presence of one token in place lastRes1 means that the system is delivering the last result set of the current request. The two cases of the activity NextHWF represent the probabilities that the effect of an hardware fault is removed (token removed from the place HWFail1).

6 System Evaluation

This section describes the results obtained via combined use of direct measurements on the system prototype and the evaluation of the model of the system. Our analysis consists of two main parts. The first part shows how to best tune the parameters of the α-counts. The second part reports some measures of system performability obtained exercising the SAN model. The concept of performability has been introduced in the seminal work of J. Meyer [27]. Since then it has attracted widespread attention and has been the topic of a vast amount of literature which cannot be fully reported here. It is a unified measure proposed to deal simultaneously with performance and dependability. performability evaluation involves specifying a performance (reward) variable (which has the generic meaning of what a system accomplishes during its use) and determining a reward model for the performance variable, i.e. a reward structure which associates reward rates with state occupancies and reward impulses with state transitions [27]. Obviously, the accomplishment levels and the reward structures depend on the system and the measures of interest. In the following we define the measures specific for our system.

6.1 Performability Metrics and Assumptions

The possible outcomes of a request to a channel are:

- Success - for queries the channel returns the correct value(s) within the timing con-

straints (before the Voter timeout expires, for updates, the databases are updated correctly;

- Timing Errors - The channel returns no value (before the Voter timeout expires);
- Value Errors - The channel returns a wrong value. More precisely: *i*) it returns a value different from what is actually stored in the physical database; *ii*) it stores to the physical database a value different from the input; *iii*) it does not perform the requested operation.

Thus, the events that contribute to define the performability model for the system performing a mission of finite duration t are the following:

- Success - The channel returns the correct value(s). $N_S(t)$ denotes the number of successes in the interval $[0, t]$.
- Timing query failure - A query fails due to a timeout, but the mission does not fail because of this event, since correct results to the same query can be obtained once the system has undergone recovery actions. $N_{TF}(t)$ is the number of failed queries (due to a timeout) in the period $[0, t]$.
- Unavailable system - the result of query cannot be delivered, since the system is not available (it is engaged in *recovery* activities). This event is not a mission failure. $N_R(t)$ indicates the number of queries which cannot be delivered in the period $[0, t]$ due to recovery activities.
- Recovery failure (*rfail*) - Failure of the restoration of the database, due to erroneous copies of a same record stored in at least two databases. This event makes the system to halt, thus failing to completing the mission.
- Query value failure (*qfail*) - The system fails to respond to a query by providing a wrong value. This outcome is due to erroneous copies of a same record of the result stored in at least two databases. This event represents a mission failure. (the difference with the previous case is that in the previous case the system halts once it is aware of the unrecoverable corrupted state, while in the present case a wrong output is emitted).

- Mission success (*msucc*) - The system terminates successfully the mission at time t.

In performing system evaluation, a time dependent performability measure $Y(t)$ has been identified as an appropriate indicator to evaluate our proposal, defined as:

$$Y(t) = \begin{cases} N_S(t)G - N_{TF}(t)C_{TF} - N_R(t)C_R, & \textit{if msucc,} \\ N_S(t)G - N_{TF}(t)C_{TF} - N_R(t)C_R - C_{RF}, & \textit{if rfail,} \\ N_S(t)G - N_{TF}(t)C_{TF} - N_R(t)C_R - C_{QF}, & \textit{if qfail,} \end{cases}$$

where a reward G is accumulated for each correct query, a penalty C_{TF} is paid for each timing query failure, a penalty C_R is paid for each query that cannot be provided during a recovery, penalties C_{RF} or C_{QF} are paid in the case of database recovery failure or query value failure, respectively.

From the definition of the performability variable $Y(t)$, the expected effectiveness of the system $E[Y(t)]$ can be derived as:

$$E[Y(t)] = E[N_S(t)]G - E[N_{TF}(t)]C_{TF} - E[N_R(t)]C_R - P_{rfail}(t)C_{RF} - P_{qfail}(t)C_{QF},$$

where $P_{rfail}(t)$ is the probability of observing the recovery failure event (rfail) within the interval [0, t] and $P_{qfail}(t)$ is the probability of observing the query value failure event (qfail)within the interval [0, t].

It is assumed that software and hardware faults times of the channels are statistically independent and for each fault activation a single bit flip occurs. Service requests form a Poisson process with rate λ_r. We assume that the maximum number of records involved in a service request is L_Q^{max} for a query and L_U^{max} for an update, and that the distribution of the number of records of a query L_Q is a modified geometric, normalized (truncated) with respect to L_Q^{max}, with parameter p_r, defined as: $P(L_Q = i) = \frac{p_r(1-p_r)^i}{1-(1-p_r)^{L_Q^{max}}}$. The distribution of the number of records involved in an update is defined in a similar way.

6.2 Evaluation Strategy

In order to analyze and evaluate the proposed architecture and to tune the relevant parameters for the α-counts, we adopted an approach based on combined use of modelling and prototype-based measurements. This approach appears as the most promising one for large complex systems [17]. Fault injection experiments and performance measurements were performed on the prototype to obtain realistic values for the parameters of the an-

alytical model. Consistently with the basic assumptions made earlier, both the network and the CORBA infrastructure and services have been considered reliable. The focus of our fault injection champaign has been on faults affecting the channels mainly to observe how faults and errors propagate in the system, as discussed in 4.2. Fault injection was conducted using the NFTAPE tool [16]. We used different machines for the channels and different workloads.

The time needed to perform the recovery actions and to service a request were derived from the analysis of the experimental data. The values for these parameters are summarized in Table 1 where the range of the values observed is reported.

Parameter	Description	Range
T_1	Time to restart the application	$0.1 - 0.5$
T_2	Time to restart the machine	$130 - 460$
T_3	Time to reconstruct the database (about 10GB)	$10000 - 30000$
T_4	Time to serve a request (at sustained rate)	$0.05 - 0.15$

Table 1: Parameter values obtained from measurements on the prototype [sec]

Table 2 reports the main parameters of the SAN model and their reference values. These values, together with the higher extremes reported in Table 1, were used as default values in the evaluation of the SAN model. In the analysis a period of 5 years has been considered. The expected times to first occurrences of application and OS errors have been set to 15 and 13.7 days, respectively. Then, since N_{crash} is set to 10, the crash of the application or of the OS have a mean of 30 and 27.4 days, respectively. The rates of the hardware faults during normal and abnormal periods are 1 every 105 days and 1 every 2 minutes, respectively. The expected durations of the normal and abnormal periods is 315 days and 6 minutes. The ratio between the occurrences of the application, OS and hardware faults is 42%, 46% and 12% (such values are in accordance with experimental studies conducted by other authors [32, 25, 15]). Whenever possible the reference values have been chosen from measurements, field experience or published results, other have been selected driven by reasonableness and authors experience. [SILV] questa frase si mette o no?????? In any case the validity of the evaluations performed, aiming at comparing alternatives and at showing that tuning is possible through studies of sensitivity, does not depend on the precision of the reference values used.

Parameter	Description	Default value
t	Duration of the period of the analysis	$1.5768E{+}8$
λ_r	Rate of input requests	0.1
p_Q	Probability of query input request	0.6
p_U	Probability of update input request	0.4
L_Q^{max}	Maximum number of records replied to a query request	50
L_U^{max}	Maximum number of records affected in an update request	30
p_r	Parameter of the modified geometric normalized distribution representing the number of records of a reply	0.1
L_{IQ}	Size of the input queue	100
t_{out}	Timeout for the delivery of a result set	15
L_{RS}	Number of records of a result set	10
K_I	Decrease ratio of the 1st α-count	0.99
T_I	Threshold of the 1st α-count	3
K_{II}	Decrease ratio of the 2nd α-count	1
T_{II}	Threshold of the 2nd α-count	6
N	Number of records	10^7
μ_A, μ_O	Scale parameters of the lognormal distributions of the times to errors of application and OS	12
σ_A^2, σ_O^2	Shape parameters of the lognormal distributions of the times to errors of application and OS	$4.2, 4$
N_{crash}	Number of software errors to crash of the application or of the OS	10
λ_N	Rate of the hardware faults during normal periods	$1.10231E$-7
T_N	Expected duration of the normal periods	27215640
λ_B	Rate of the hardware faults during abnormal periods	$8.33E$-3
T_B	Expected duration of the abnormal periods	360
p_c	Probability of removing the effect of the hardware failure at each service request	0.1
L_{rec}	Number of bytes for each record	1024
L_{addr}	Size of an address (bytes)	4
L_{mem}	Number of addresses of the memory	2^{32}
G	Gain accumulated for each correct query	1
$G(t)$	Gains accumulated at time t (in absence of faults) for G=1	9460800
C_R	Penalty paid for each query during a recovery	15
C_{TF}	Penalty paid for each failed query that can be recovered	$10C_R$
C_{RF}	Penalty paid in the case of DB recovery failure	$C_{QF}/200$
C_{QF}	Penalty paid when a query fails and the result cannot be recovered	$20G(t)$

Table 2: Parameters and their default values used in the evaluation (time in seconds)

6.3 Parameter setting and tuning

A good performance of the system can be reached if one properly understands how frequently and under which system conditions the restoration procedure should be scheduled. The procedure is triggered by the second α-count. The fact that the records can be corrupted but cannot be corrected by a service request is represented by setting K_{II}=1. The recovery follows a majority voting approach and, if the database is not correctly recovered (upon completion there are still erroneous records) the system halts with a failure. The probability that the recovery procedure reaches its goals, i.e., that a correct version of the database can be reconstructed is evaluated as a function of the amount of corrupted records existing in the three replicas. We assume: *i*) a uniform distribution of erroneous data, *ii*) that corrupted replicas of the same record are perceived as different, and *iii*) (conservatively) that all the replicas contain the same number of corrupted records. The assumption iii) is acceptable in some cases, but does not hold in others. In any case, determining the probability of failure recovery of a database assuming that all channels have the same maximum number of corrupted records constitutes a worst for the estimated measure. In the general case such probability is lower, as showed in Figure 10.

Under assumptions *i*) and *ii*), a good approximation of the probability p_k^e that there are k erroneous records after executing the recovery procedure is obtained using a binomial distribution:

$$p_k^e = \binom{N}{k} q^k (1-q)^{N-k},$$

where q is the probability of failure of the recovery of a generic record, defined as:

$$q = q_1 q_2 q_3 + q_1 q_2 (1-q_3) + q_1 (1-q_2) q_3 + (1-q_1) q_2 q_3,$$

where $q_i = \frac{N_i^e}{N}, i = 1, 2, 3$, being N_i^e is the number of erroneus records of the i-th channel.

Figure 10 plots the probability of failure of the recovery procedure $1 - p_0^e$ as a function of the number of corrupted records, under assumption *iii*), that is all the channels have the same maximum number of corrupted records n^{max}, and in a less pessimistic case where two channels have $n^{max}/2$ corrupted records. Given a desired probability of failure for the recovery procedure, one can now relate the number of corrupted records to the threshold of the second α-count which is used to trigger the recovery procedure and to the threshold of the first α-count which is used to trigger the restart of the host computer.

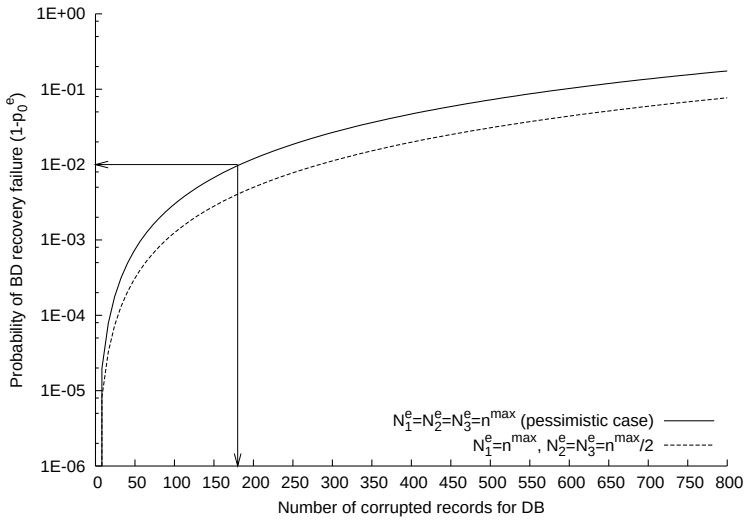

Figure 10: Probability of failure of the recovery procedure as a function of the number of corrupted records

Figure 11 reports the number of erroneous records of a replica of the database as a function of T_I for different values of T_{II} as estimated by simulation with our SAN model of the system. Figure 11 shows that, in the settings chosen, the number of corrupted records ranges from about 3 for low values of T_I and T_{II} to about 800 when T_I and T_{II} go to 6. In more details, at low values of T_I the number of corrupted records grows markedly as T_I increases; for $T_I > 1.5$ these variations become smaller. The influence of T_I increases as T_{II} increases. The figure shows that choosing a desired value for $1 - p_0^e$ results in specific constraints on the number of the corrupted records. As an example, to get values of $1-p_0^e \leq 10^{-2}$ the recovery has to be performed when at most 180 records are corrupted. Figure 11 show that values of $T_{II} = 2$ and T_I in the range $[1, 2.8]$ satisfy the condition that the number of corrupted records is in the range $[0, 180]$. Note that many different combination of values for T_I and T_{II}, can satisfy the condition.

6.4 Performability Results

In the following, we analyze the perfomability results obtained for different values of the α-count parameters and of the cost parameters. Figure 12 reports the expected performability of the system in 5 years, as a function of T_I for several values of T_{II}. Additionally $N_S^{max}(t)$ (defined as the maximum number of correct queries that can be performed in absence of faults in a period $[0, t]$) has been plotted. $N_S^{max}(t)$ represents the performance in the nominal case and is the theoretical upper bound for $Y(t)$.

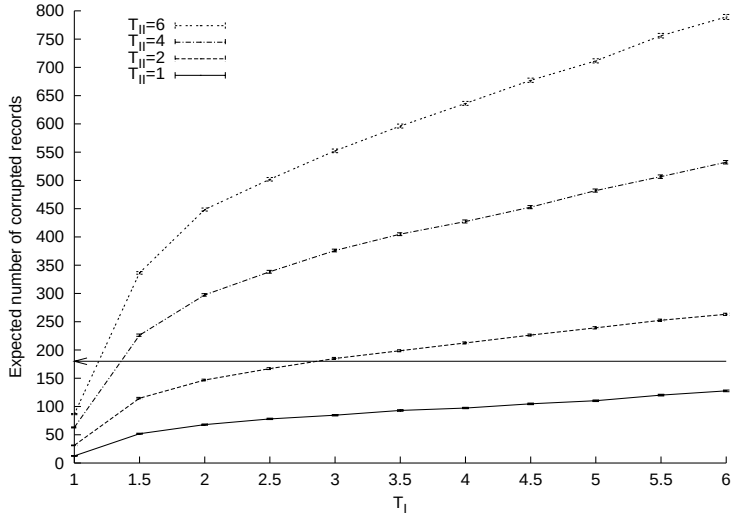

Figure 11: Number of corrupted records as a function of T_I for different values of T_{II}

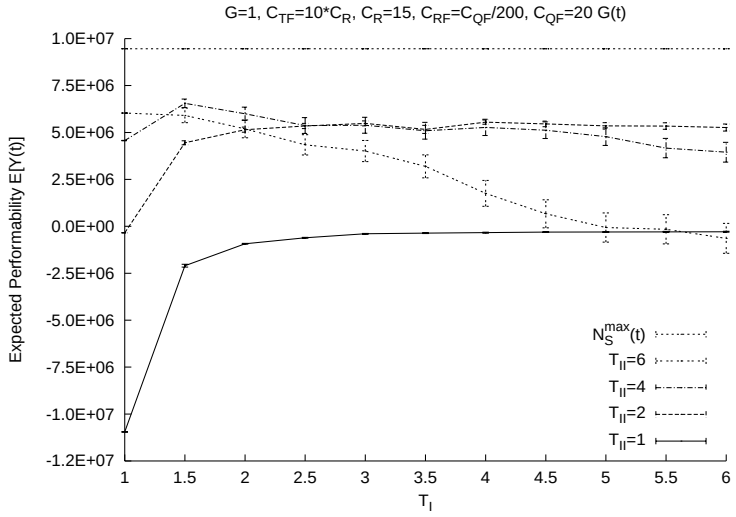

Figure 12: Expected performability of the system in 5 years, as a function of T_I for several values of T_{II}

The figure shows that the setting $T_I = 1$ and $T_{II} = 1$, which represents a system which does not incorporate the proposed diagnostic mechanisms (in such a system, a database restoration would be attempted upon occurrence of each failure) is the one which delivers the worst performability. The figure shows that distinguishing among the recovery actions brings benefits to the obtained performability. More precisely, the combination $T_I = 1$ and $T_{II} > 1$, in which R1 and R2 are always executed together while R3 is executed only when the second threshold is reached, gives better performability than the case when the three actions are executed all together. The same holds when R2 and R3 are executed

together and R1 is kept separated, that is when $T_I > 1$ and $T_{II} = 1$. The curve marked $T_{II} = 1$ in Figure 12 is increasing for increasing values of T_I. From the figure it is evident that the proposed strategy brings significant benefits and can take further advantages from a careful usage of the identified recovery actions. In the scenario considered the best expected perfomability is reached in correspondance of the setting $T_I = 1.5$ and $T_{II} = 4$. Additionally, also simple-to-determine, non optimal choices for the α-count parameters bring extremely significant improvements of performability, as compared to the case $T_I = 1$ and $T_{II} = 1$ (i.e. the system without our diagnostic mechanisms).

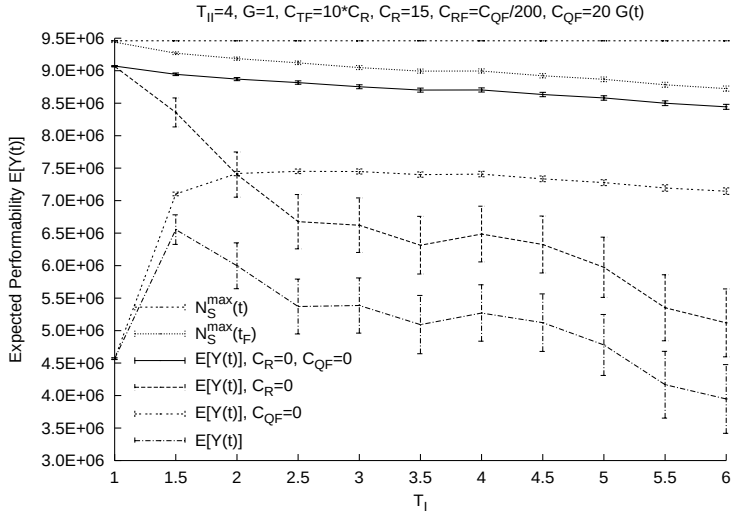

Figure 13: Impact on the performability of the main factors contributing to it as a function of T_I, with $T_{II} = 4$

Figure 13 reports the impact of the main factors on the performability of the system in 5 years, as a function of T_I, for $T_{II} = 4$. The curves reported are (from the top):

- $N_S^{max}(t)$ as defined earlier,
- $N_S^{max}(t_F)$, defined as the expected maximum number of correct queries that can be performed in absence of faults in a period $[0, t_F]$ with $[t_F]$ the expected time to failure of the system,
- The expected performability $E[Y(t)]$ for $C_R = 0$ and $C_{QF} = 0$. This equals the expected number of successes in the interval $[0, t]$, since the cost of all the other events is set to 0.
- The expected performability $E[Y(t)]$ for $C_R = 0$. In this case only the cost associated

to system failure is considered.

- The expected performability $E[Y(t)]$ for $C_{QF} = 0$. Here only the cost due to unavailability during recovery actions is accounted for.

- The expected performability $E[Y(t)]$ including all the costs.

The figure allows to appreciate the contribution of the various factors to the performability. It shows several upper bounds and the two main factors that reduce the performability: the cost of failure during restoration and the cost of query value failure.

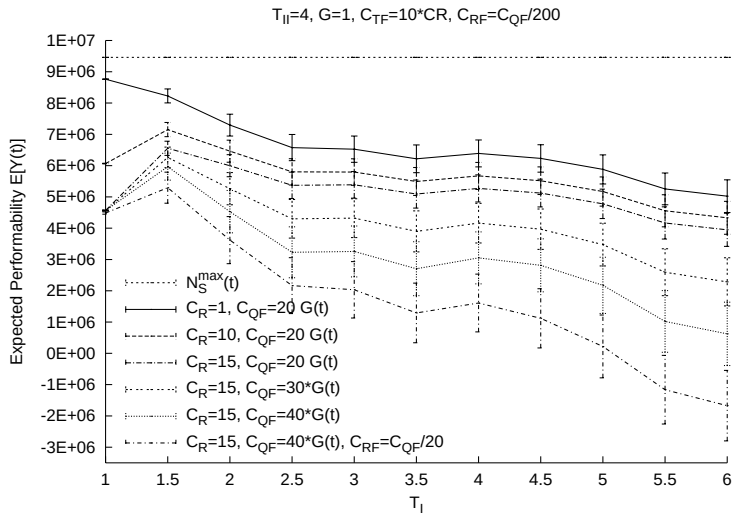

Figure 14: Expected performability of the system in 5 years, as a function of T_I for several values of C_R and C_{QF}, with $T_{II} = 4$

Figure 14 reports the expected performability of the system in 5 years, as a function of T_I for several values of C_R and C_QF, with $T_{II} = 4$. The curves allow to appreciate the interplay of the different costs associated to system events and operations.

Figure 15 shows the expected performability of the system in 5 years, as a function of T_I for several values of K_I, and $T_{II} = 4$. The figure shows that given a valued for K_I there is one corresponding best value for T_I. In fact, each curve in the figure has its maximum in correspondence of a different value for T_I. In the case we have considered the absolute best combination is obtained for $K_I = 0.9999$ and $T_I = 3$.

This analysis allowed us to improve our understanding of the dynamics ruling the system behavior and gave us hints on how the parameters of the α-counts have to be set for the system to obtain good performability. The analysis also showed that it is

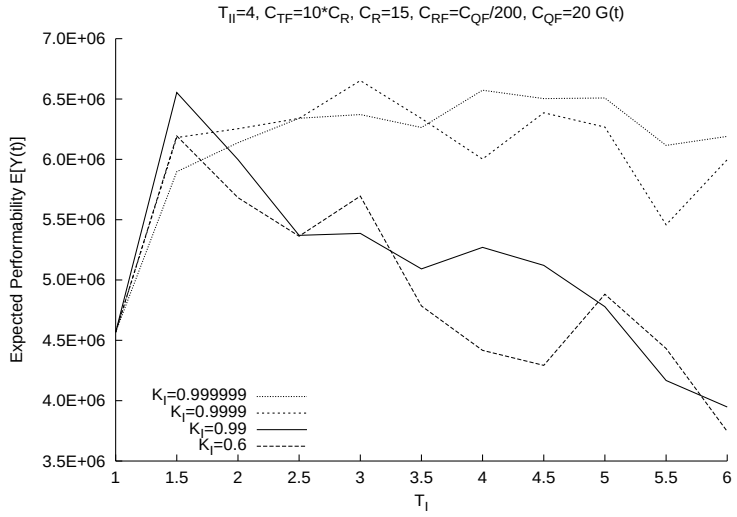

Figure 15: Expected performability of the system in 5 years, as a function of T_I for several values of K_I, and $T_{II} = 4$

very useful to distinguish among the identified recovery actions, which have to be applied individually. It appears instead less important to have a very precise determination of the threshold values: the differences in the obtained performability levels do not seem to be particularly relevant. In any case the benefit of the proposed diagnostic and treatment strategy is evident: the case $T_{II} = 1$, $T_I = 1$ has been the one where the system performed by far the worst in all the settings we considered.

7 Conclusions

The novel contribution of this paper is a methodology and an architectural framework for handling multiple classes of faults (namely, hardware-induced software errors in the application, process and/or host crashes or hangs, and errors in the persistent system stable storage) in a COTS- and Legacy-based application by using an evidence-accruing fault tolerance manager to choose and carry out one of multiple fault tolerance strategies, depending upon the perceived severity of the fault. A case study system has been implemented and benchmarked using a performability benchmark. In particular, the study focused on the best threshold values for the number of faults needed before triggering specific recovery mechanisms. The effectiveness of the suggested approach is evaluated through combined use of direct measurements on the system prototype and analytical modeling. The use of a fault injection on a real system prototype allowed us to derive re-

alistic fault models (by tracking error propagation through individual system layers), and to extract relevant system parameter values (which were used to populate an analytical model of the system). A SAN model of the overall system was developed and thoroughly evaluated. The model consisted of ten sub-models joined together. We performed several evaluations by simulation in order to account for non exponential events.

The application used as a case study consists of legacy code, written in C, which uses a COTS DBMS, for persistent storage facilities. The application runs on Linux, on top of a commodity personal computer. A careful Failure mode effect analysis has been performed by fault injection to understand how faults and error propagate, thus identifying the most promising countermeasures. Special care was devoted to assess the status or the extent of the damage in the channels, through a diagnostic subsystem based on the concept of α-count, so to carefully calibrate the use of the fault treatment and system reconfiguration actions identified: Action 1 (application restart), Action 2 (host restart), and Action 3 (data base restoration). The analysis shows: i) how to set proper values for the parameters, and ii) the efficacy of the system which calibrates different recovery actions.

ACKNOWLEDGEMENTS

This work has been partially supported by: the Italian Ministry for Education, University and Research (MIUR), within the framework of the FIRB Project "Middleware for advanced services over large-scale, wired-wireless distributed systems (WEB-MINDS); the Regione Campania, within the framework of the "Centro di Competenza Regionale ICT" and "Telemedicina" projects; the National Research Council, within the framework of "Strumenti, Ambienti e Applicazioni Innovative per la Societá dell'Informazione", SOTTOPROGETTO 4.

References

[1] K. J. Cassidy, K C. Gross, and A Malekpour, Advanced Pattern Recognition for Detection of Complex Software Aging Phenomena in Online Transaction Processing Servers, in proc. of International Conference on Dependable Systems and Networks, 2002.

[2] P. Narasimhan, and P.M. Melliar-Smith, State Synchronization and Recovery for Strongly Consistent Replicated CORBA Objects, in proc. of The 2001 International Conference on Dependable Systems and Networks, 2001.

[3] C. Sabnis, W.H. Sanders, D.E. Bakken, M.E. Berman, D.A. Karr, M. Cukier, AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects, in proc. of The IEEE 17th Symposium on Reliable Distributed Systems, 1998.

[4] Z.T. Kalbarczyk, R.K. Iyer, S. Bagchi, K. Whisnant, Chameleon: a Software Infrastructure for Adaptive Fult Tolerance, IEEE Trans. on Parallel and Distributed Systems, vol. 10, pp. 560–579, 1999.

[5] R. Baldoni, C. Marchetti, M. Mecella, A. Virgillito, An Interoperable Replication Logic for CORBA Systems, in proc. of The 2nd International Symposium on Distributed Object Applications 2000 (DOA00), 2000.

[6] B. Natarajan, A. Gokhale, S. Yajnik, and D.C. Schmidt, DOORS: TowardsHigh-performance Fault-tolerant CORBA, in proc. of International Symposium on Distributed Objects and Applications (DOA'00), 2000.

[7] D. Cotroneo, N. Mazzocca, L. Romano, S. Russo, Building a Dependable System from a Legacy Application with CORBA, Journal of Systems Architecture, vol. 48, pp. 81–98, 2002.

[8] J.C. Fabre, T. Perennou, A metaobject architecture for fault-tolerant distributed systems: the FRIENDS approach, IEEE Transactions on Computers, vol. 47, pp. 78–95, 1998.

[9] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, F. Grandoni, Threshold-Based Mechanisms to Discriminate Transient from Intermittent Faults, IEEE Transactions on Computers, vol. 49, pp. 230–245, 2000.

[10] D. Powell, G. Bonn, D. Seaton, P. Verissimo, F. Waeselynck, The delta-4 approach to dependability in open distributed computing systems, in Proc. of the 18th International Symposium on Fault Tolerant Computing Systems (FTCS 18), 1988.

[11] O.M. Group, Fault-Tolerant CORBA Specification, v1.0, OMG, http://www.omg.org, document ptc/00-04-04 2001.

[12] P. Felber, R. Guerraoui, A. Schiper, "The Implementation of a CORBA Object Group Service", in *Theory and Practice of Object Systems (TAPOS)*,Wiley&Sons, Vol. 4, No. 2, 1998

[13] L. Romano, S. Chiaradonna, A. Bondavalli, D. Cotroneo, Implementation of Threshold-based Diagnostic Mechanisms for COTS-based Applications, in proc. of The 21st IEEE Symposium on Reliable Distributed Systems (SRDS 2002), Osaka, Japan, 2002.

[14] K.K. Goswami, R.K. Iyer,Simulation of Software Behavior Under Hardware Faults, in Proc. of the 23rd Annual International Symposium on Fault-Tolerant Computing, 1993.

[15] R.K. Iyer, D. Tang, Experimental Analysis of Computer System Fault Tolerance", in chapter 5 of Fault-Tolerant Computer System Design, D.K. Pradhan, Prentice Hall Inc., 1996.

[16] D. Stott, P. H. Jones, M. Hamman, Z. Kalbarczyk, R. K. Iyer, NFTAPE: networked fault tolerance and performance evaluator, in proc. of International Conference on Dependable Systems and Networks, 2002.

[17] DBench Consortium, Measurements, Deliverable ETIE1, IST-2000-25425 Dependability Benchmarking (DBench), 2002.

[18] R. K. Iyer, L. T. Young, P. V. K. Iyer, Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data, IEEE Transactions on Computers, Vol. C-39, pp. 525-537, 1990.

[19] T.T. Y. Lin and D. P. Siewiorek, Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis, IEEE Transactions on Reliability, Vol. 39, pp. 419-432, 1990.

[20] P. Agrawal, Fault Tolerance in Multiprocessor Systems without Dedicated Redundancy, IEEE Transactions on Computers, Vol. C-37, pp. 358-362, 1988.

[21] A. Bondavalli, S. Chiaradonna, F. di Giandomenico, F. Grandoni, Discriminating fault rate and persistency to improve fault treatment, in proc. of Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing, 1997 (FTCS-27), pp. 354–362.

[22] G. Mongardi, "Dependable Computing for Railway Control Systems," in Proc. DCCA-3, Mondello, Italy, 1993, pp. 255-277.

[23] N. N. Tendolkar, R. L. Swann, Automated Diagnostic Methodology for the IBM 3081 Processor Complex, IBM J. Res. Develop., Vol. 26, pp. 78-88, 1982.

[24] Yennun Huang, Chandra M. R. Kintala, Nick Kolettis, N. Dudley Fulton: Software Rejuvenation: Analysis, Module and Applications. FTCS 1995: 381-390

[25] R. Mullen, The Lognormal Distribution of Software Failure Rates: Origin and Evidence, in proc. of The Ninth International Symposium on Software Reliability Engineering, Paderborn, Germany, 1998.

[26] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, P. G. Webster, The Mobius framework and its implementation, IEEE Transactions on Software Engineering, 28(10), pp. 956–969, oct. 2002

[27] John F. Meyer: On Evaluating the Performability of Degradable Computing Systems. IEEE Trans. Computers 29(8): 720-731 (1980).

[28] Ken Birman, Robert Constable, Mark Hayden, Christopher Kreitz, Ohad Rodeh, Robbert van Renesse, Werner Vogels,The Horus and Ensemble Projects: Accomplishments and Limitations, in Proceedings of the DARPA Information Survivability Conference & Exposition (DISCEX '00), 2000.

[29] D. Cotroneo, A. Mazzeo, L. Romano, S. Russo, Implementing a CORBA-based architecture for leveraging the security level of existing applications, 8th International Symposium on Distributed Objects and Applications (DOA 2002), Lecture Notes in Computer Science Series, LNCS 2519, Springer Verlag, 2002.

[30] A. Bondavalli, S. Chiaradonna, D. Cotroneo, L. Romano, A Fault-Tolerant Distributed Legacy-based System and Its Evaluation, 1st Latin American Symposium on Dependable Computing (LADC 2003), Lecture Notes in Computer Science Series, LNCS 2847, Springer Verlag, 2003, pp. 303-320.

[31] L.E. Moser, P.M. Melliar-Smith, P. Narasimhan, L. Tewksbury and V. Kalogeraki, "The Eternal System: An Architecture for Enterprise Applications", in *Proc. of International Enterprise Distributed Object Computing Conference* ,University of Mannheim, Germany (September 1999), pp. 214-222.

[32] R. Chillarege, S. Biyani, J. Rosenthal, Measurement of failure rate in widely distributed software Fault-Tolerant Computing, in proc. of Twenty-Fifth International Symposium on Fault Tolerant Computing Systems (FTCS-25) , pp. 424-433, June 1995.

[33] W. H. Sanders and J. F. Meyer. Stochastic activity networks: Formal definitions and concepts. In E. Brinksma, H. Hermanns, and J. P. Katoen, editors, *Lectures on Formal Methods and Performance Analysis*, volume 2090 of *LNCS*, pages 315–343. Springer-Verlag, 2001.

[34] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabéjac, and A. Wellings. GUARDS: a generic upgradable architecture for real-time dependable systems. *IEEE Transactions on Parallel and Distributed Systems, Special Issue on Dependable Real-Time Systems*, 10(6):580–599, 1999.