A Fault-Tolerant Distributed Legacy-based System and Its Evaluation

A. Bondavalli[1], S. Chiaradonna[2], D. Cotroneo[3], and L. Romano[3]

[1] Dipartimento di Sistemi e Informatica, Universitá di Firenze,
Via Lombroso 6/17, 50134 Firenze, Italy
a.bondavalli@dsi.unifi.it

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

[3] 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 for improving the dependability of complex COTS and legacy-based systems. For long-lived applications, such as most of those being constructed nowadays via integration of legacy subsystems, fault treatment is a very important part of the fault tolerance strategy. 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 component. 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, handling replicated legacy-based subsystems. Replication and voting are used for error detection and masking. An experimental prototype deployed over a COTS-based LAN is described and has allowed a dependability analysis, via combined use of direct measurements and analytical modeling.

Keywords: Software Implemented Fault Tolerance, Fault Diagnosis, Fault Treatment, Legacy systems, Threshold-based mechanisms, CORBA Architectures

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) and legacy systems. The resulting systems 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, as well as attacks and intrusions, i.e. they must provide some level

of (reduced) functionality also in the event of faults. The integration of COTS components and legacy subsystems in a wider infrastructure is a challenging task, for a variety of reasons, and in particular:

- COTS components are relatively unstable and unreliable [3]. Nevertheless, to reduce system development time, the use of unmodified COTS components is mandatory in modern distributed systems;
- Legacy systems were designed as disjoint components and they were not supposed to interoperate. As a consequence, the integration environment might stimulate legacy components in unexpected ways, thus leading to potential failures. This is a well-known problem to integration test professionals. Indeed, the task of the integration test is to check that components or software applications, e.g. components in a software system or - one step up - software applications at the company level - interact without error. Many examples from field experiences can be found at `http://www.sqs.de/english/casestudies/index.htm`.

For these reasons, without effective architectural solutions, survivable infrastructures consisting of COTS and legacy-based applications, are virtually impossible to obtain. In order to be effective, fault tolerance strategies in such systems need to be revisited. More precisely, mechanisms and strategies to implement fault tolerance functions have to be tuned, to account for the many differences between COTS and legacy-based systems and traditional safety-critical systems. Recently some proposals have been made, which allow to build dependable systems by integrating COTS and legacy components [4], [5], [6], [7], [8], [9], [10]. These proposals mainly concentrate on error processing, either by NMR-style replication or by organizing distributed membership of replicas. However, little attention has been paid to the problem of maintaining system health and preserving tolerance capabilities.

This paper focuses on diagnosis and fault treatment and proposes their integration with mechanisms for error detection and processing. Fault treatment consists of fault diagnosis, and recovery/reconfiguration [11]. Although diagnosis has been extensively studied, distributed COTS and large legacy-based systems raise 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 knowledge and control over the system as a whole, as well as over individual components, since for most components and subsystems the internal design is not known. Second, COTS and legacy components are heterogeneous, whereas the targets of traditional diagnosis 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. It is thus not practical (or possible at all), as soon as an error is observed, to declare the entire component failed and proceed to repair and replacement (repair implies at least very costly recovery and re-integration, replacement may not be possible at all).

For the above discussed reasons, in a COTS and legacy-based infrastructure diagnosis must be able to assess the status or the extent of the damage in individual components, so to carefully identify the most appropriate fault treatment and system reconfiguration actions and when they have to be applied. To this aim, it is foremost that data about error symptoms and failure modes be carefully gathered and processed. One shot-diagnosis is thus 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 to many fields (e.g. diagnosis and telecommunications). All these mechanisms take their decisions based on the analysis of streams of data, consisting of sequences of observations.

We present a complete architecture for improving the dependability of complex COTS and legacy-based systems. We emphasize aspects related to fault treatment. We had to face several key issues, and in particular:

- Limiting the probability of common mode failures - The replicated legacy systems were identical and thus they could exhibit common mode failures. Since we could not modify the legacy systems, we enforced diversity at different architectural levels when we integrated the legacy systems in our replication framework. More details about the specific measures that we have taken are provided in subsection 3.2.
- Limiting the probability of interaction faults - These are typically caused by mismatches or incompatibilities between the legacy applications and the COTS platforms and software components. Research has demonstrated that in order for an error detection process to work properly, data from individual replicas must be conditioned before comparisons are made [19].

We detail the description of the architectural solutions we have implemented with respect to a case study. We also evaluate the effectiveness of the suggested approach via combined use of direct measurements on the system prototype and analytical modeling. We use a real prototype to populate the model with realistic parameter values. The rest of the paper is organized as follows. Section 2 introduces previous relevant work and illustrates the system we use as a case study. Section 3 details our complete fault tolerance strategy highlighting fault treatment and describes our implementation. Section 4 describes our approach to the analysis, and the models we built. Section 5 reports the results we obtained so far, while section 6 concludes the paper.

2 Preliminaries

2.1 Background

A great deal of research has been conducted on providing support for dependability to existing applications via distributed architectural frameworks. These projects differ from one another under 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). Among the others, it is worth mentioning [6], [7], [9], [10], [13], as well as the FT CORBA initiative [14]. All these projects focus on error processing, but they address fault-treatment only to a limited extent. There are also several commercial products which claim to incorporate fault tolerant facilities (such as J2EE [2] and Microsoft .NET [1], to name a few). As to our personal experience, in [9] we presented a middle-tier based architectural framework for leveraging the dependability of legacy applications in a way transparent to the clients. Such a framework can be deployed on top of any CORBA infrastructure. The fault tolerance mechanisms used include error detection, different forms of error processing, graceful degradation, and re-integration. A system prototype was developed and tested over a distributed heterogeneous platform. A preliminary analysis of such a prototype clearly indicated that a more effective fault treatment support was needed to significantly improve the dependability level attained by the system. Thus, we started investigating diagnosis in COTS- and legacy-based applications. More precisely, 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/repair mechanisms/staff to perform the adequate actions to maintain the system in good health [15]. This led to the definition of a diagnostic subsystem scheme which consists of two distinct components. One, the Threshold-Based Mechanism (TBM), implements a threshold-based algorithm, the other, the Data Processing & Exchange (DPE), is in charge of conveying to the former the data from the detection sub-system, possibly performing some processing. The two blocks (TBM and DPE) might well be distributed in actual implementations of the system. Typically, DPE functions would be co-located with the error detection sub-system. The specific algorithm implemented by the TBM is called alpha-count, and is described in [12]. Among the heuristics based on the concept of threshold, the alpha-count family of mechanisms appears to be particularly interesting for our purposes due to the clear and simple mathematical characterization and to the thorough analysis already conducted. In the following, we briefly describe how the alpha-count mechanism works. The alpha-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*
- $\alpha_i(L) = K * \alpha_i(L-1)$ $(0 < K < 1)$ *if channel i is perceived as correct during execution L*

When $\alpha_i(L)$ becomes greater than or equal to 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). The effectiveness of the mechanism depends on the parameters K and α_T. The optimal tuning of these parameters depend on the

expected frequency of errors and on the probability c of correct judgements of the error signaling mechanism. The analysis performed in [12] has clearly shown the trade-offs between delay and accuracy of the diagnosis and thoroughly discussed ways to tune these parameters for maximizing performance.

2.2 The application used as a case study

By legacy application, we mean a software program for which re-engineering or maintenance actions are either impossible or prohibitively costly. The application used as a case study is a multi-tier application, consisting of legacy code, written in C, which uses a COTS DBMS, namely PostgreSQL for stable storage facilities. The application runs on a Unix-like kernel, on top of a commodity PC or a workstation. The data base physical files are stored on disk. Services can be grouped in two main categories, namely *Queries* and *Updates*. The legacy applications reads data from the database to the program dynamic memory. The application allocates new memory when it needs some. Records are stored in a linked list like data structure. The DBMS also caches data.

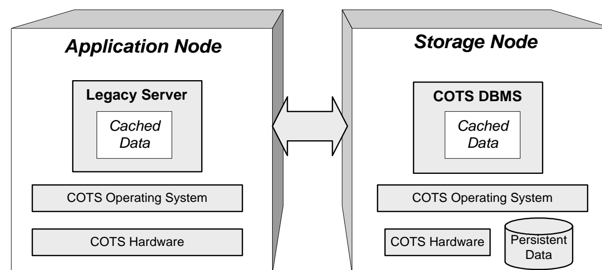

Fig. 1. Case-study legacy application

Since we could introduce a satisfactory degree of diversity (more details about this in section 3.2) in the running environments of our legacy subsystems to make the probability of common mode failures very low, our objective is to contrast the effects of independent faults hitting the back-end servers. The application was made more dependable by replicating the legacy application according to a TMR scheme. In particular we consider the following type of faults:

- Hardware-induced faults - These are faults stemming from instabilities of the underlying hardware platform [16]. We limit our attention to intermittent faults, since these are by far the predominant cause of system errors [17];
- Software errors - These are faults related to flaws in the application and/or in the software infrastructure. Examples are errors in the design and/or implementation process of the application and/or of the Operating System, or to process and/or host crashes or hangs due to inconsistent application

and/or OS level state, or to unavailability of resources due to overload conditions. We distinguish between "naive" software faults, such as trivial bugs, and "'subtle"' software faults, such as memory leaking problems in the application or system resource exhaustion. We limit our attention to the latter kind of faults, since we assume that the former kind of faults has been detected and fixed during the years long operation of the legacy system. This is not a simplistic assumption. In fact, in the typical scenario, legacy software has been thoroughly tested and debugged, and the vast majority of naive bugs has thus been detected and fixed over the years. However, more subtle software errors may still be present in the code base. As an example, errors related to incorrect use of memory may well have gone undetected (it is likely that the programmer has requested the allocation of a certain amount of memory, but he has failed to deallocate it). 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 the case of the legacy application). However, if the new deployment scenario requires that the legacy system be launched as a long running application, these subtle problems would eventually manifest.
- Faults in the physical data-base - Corrupted data has been written to the system stable storage.

We do not consider communication errors. Such errors occur when data gets corrupted while traversing the network infrastructure. Indeed, unless "leaky" communication protocols are adopted, this kind of errors is fairly unlikely to happen (a leaky protocol is a protocol that allows corrupted information to be delivered to the destination).

The above discussed faults manifest as failures of the legacy application as follows. Hardware faults may corrupt the linked list structure of the application cache. This may happen in two ways, which result in errors of different severity. A first problem occurs when a data item in the list gets corrupted. This is the less severe kind of error, since at the application level it results in a corrupted entry being written to the database or exposed to the system external interface. A second problem occurs when a pointer gets corrupted. This is a more severe error, since it manifests as a multiple error. One possibility is that the faulty pointer points to a null value. At the application level, this error results in a truncated list being written to the physical database (i.e. in possibly several items being deleted from the database). Another possibility is that the faulty pointer points to a wrong item in the list. This error would result in possibly several items being added/deleted to/from the database. Yet another possibility is that the the faulty pointer points to an invalid address. This error typically results in a signal being generated by the Operating System and in the application being killed.

Software faults may lead to resource exhaustion. As an example, memory leaks may result in a huge amount of memory being allocated to the faulty application. At first, this would result in an overload condition for the hosting node, with potential performance faults which would manifest as timeout failures

of the application. Eventually, the Operating System would deny the allocation of further resources to the application. Again, this would typically result in a signal being generated by the Operating System and in the application being aborted.

3 Fault Tolerance and System Prototype

In this section, we describe the complete fault tolerance strategy that we propose. In particular, we highlight the fault treatment logic (i.e. alpha count mechanisms, and data exchange techniques) and describe its implementation applied to our case study (i.e. the architecture of the distributed legacy based system and the roles played by individual components).

3.1 Fault tolerance approach

Our architecture is based on a middle-tier to manage three replicas of the legacy database and COTS DBMS running on the back-end nodes (channels), used in a TMR fashion. Thus, a voter in the middle tier assures masking of one replica failure and conveys to our diagnostic subsystem information on errors detected. As we already mentioned, 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 dependability of the overall system. Thus, the middle-tier is also in charge of diagnosing the status of the channels, and of deciding i) when to perform recovery actions, and ii) which recovery action is the most appropriate. Based on the fault assumptions made, we identified three recovery actions for the legacy subsystems:

1. Restart of the application - This action can fix inconsistent application level states, but it is inefficient against errors in the system stable storage;
2. Reboot of the host computer - This action restarts the operating system and all the service software; it can thus fix inconsistent OS states as well as service software states;
3. Restoration of the data base - This action aims at correcting errors in the physical data stored on disks. Restoration is conducted as follows: the recovery manager reads binary data from the two other replicas 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.

The restoration procedure that we have described is of course just one of many possible algorithms, and we do not claim it is the best alternative (the focus of this paper is not on database restoration techniques). Two instances of alpha-count monitor each channel. These are used to choose among the three alternative recovery actions. Fault-treatment logic is as follows. Any error detected

by the voter triggers the first recovery action (application restart) and is sent to the first alpha-count which increases the score, whereas each success is used to decrement the score. When the threshold is reached, recovery action number two (host restart) is executed, the score is reset to zero and an error signal is sent to the second alpha-count, which increments its score. The score of the second alpha-count is never decremented (assuming that normal operation does not correct corrupted data). Finally, when the threshold in the second alpha-count is reached, the third recovery action is performed (database restoration) and the scores of both alpha-counts are reset to zero.

3.2 Prototype

The overall organization of the system prototype is depicted in Figure 2 and its components are described in the following. This architecture consists of 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 (channel).

Since the final effects of faults (i.e. the application failure modes) depend on several factors, which include the specific characteristics of the computing node which hosts the application, when we integrated the legacy system replicas of individual channels in the TMR architectural framework, we enforced diversity at various architectural levels of the deployment environment. By doing so, we were able to lower the occurrence of common mode failures. More precisely, the nodes in the third tier run different versions of the Linux Kernel (2.2, 2.4.18, and 2.4.19), have different amounts of RAM (0.5 GB, 1 GB, and 256 MB), and are configured so to launch a different set of services at startup.

The second tier mediates the service by hiding replication to the users and providing proper management facilities (distribution of service requests, voting upon individual results, and redundancy management). In particular, the Middle Tier handles updates by broadcasting them to all active replicas. The connectivity between the individual components is provided by CORBA [14], specifically, VisiBroker version 4.1.

Gateway – In order to replicate the COTS-based application, we had to wrap it with a multicast enabled component. To this goal, we integrated in the Gateway the reliable multicast support provided by the Ensemble group communication facility [23]. We had to attach Ensemble to the `Gateway` object (at one end) and to the server replicas (at the other end). The former task was straightforward: we just had 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. We had to intercept TCP calls, and redirect them to Ensemble. In order to do so, we developed a virtual device driver within the kernel of the node which hosted the legacy server. For a thorough description of this technique, please refer to [24]. The Gateway is in charge of all data exchanges between the Voter and the specific COTS-based application. This entails addressing all synchronization-related and format-related issues, since research has demonstrated that, in order for the voting process to work properly, data

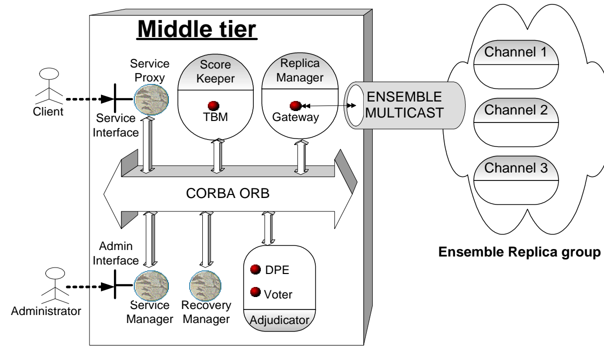

Fig. 2. Overall System Architecture

must be conditioned before comparisons are made [19]. The Gateway purifies the data from application-specific and platform-related dependencies, thus avoiding that system failures occur due to interaction problems.

Service Manager – The Service Manager component is in charge of configuring all other components. It provides functions to customize the behaviour 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). System configuration is performed via the Admin Interface.

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.

Adjudicator – The Adjudicator component incorporates both the voter and the DPE. The voter is in charge of selecting a presumably correct result out of those provided by the channels. It may support several adjudication strategies but here it is configured to perform TMR-based majority voting. Based on the results of voting, the adjudicator provides error detection information to the DPE. The DPE is in charge of packing and delivering error detection data to the threshold based mechanisms (TBM) component.

Score Keeper – The Score Keeper computes the scores. It updates the alpha-count pair which is associated to each channel. It receives data produced by the Voter and filtered by the DPE.

Recovery Manager – The Recovery Manager is in charge of performing the recovery actions triggered by the Score Keeper.

4 Dependability Analysis

In order to analyze and evaluate our proposal and to tune the relevant parameters for the alpha-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 [20].

4.1 Prototype settings

Fault injection experiments and performance measurements were performed on the prototype to populate the analytic model with realistic parameter values. Since our focus is faults which affect the channels, both the network and the CORBA infrastructure and services were considered reliable. Thus, we only injected faults to the channels. Fault injection was conducted using the NFTAPE tool [18]. The details of the fault injection campaign are not discussed here, due to lack of space. We used different machines for the channels and different workloads. From the analysis of the experimental data, the time needed to perform the recovery actions and to service a request, summarized in Table 1, were derived.

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 (1GB)	$1000 - 3000$
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]

4.2 System Models

The system has been modeled using Stochastic Activity Networks (SAN) [22]. We did not develop a detailed model of the channels,i.e., a model consisting as a large set of fine-grain components (e.g. processes, data structures, OS layer, HW layer, etc.), since this would have resulted in an unnecessarily large size for the model. Instead, we considered a channel as a relatively simple component, consisting of an application object, an OS component, and a database object, as detailed later in this section. Also, instead of modeling the whole variety of faults that we have discussed in section 2.2, we considered fault effects as they manifest as application failures. Conceptually, this is equivalent to using a fault dictionary to abstract from the component level to the application level. We assume intermittent application failures with an increasing failure rate according to a lognormal distribution [21] , since this hypothesis is consistent with the fact that the extent of the damage of the channels increases with time (if no recovery action is taken). The channel failure modes which we considered are:

– 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.

The composed model of Figure 3 represents the hierarchical model of the system behavior. It consists of ten logically separate SANs (Recovery, Client, GTDBServer1, GTDBServer2, GTDBServer3, FailModelDBServer1, FailModelDBServer2, FailModelDBServer3, ResultProviderVoter, and DiagnosticBlock), joined through common places by the Join1 operation. The SAN Recovery mimics system behavior as different types of recovery actions are taken. During recovery, the system does not serve requests. The SAN 'Client' represents the service requests, the status of the replies to the clients (correct, detected erroneous, undetected erroneous) and the number of replicated servers which are still online. The SANs GTDBServer1, GTDBServer2, GTDBServer3 represent the

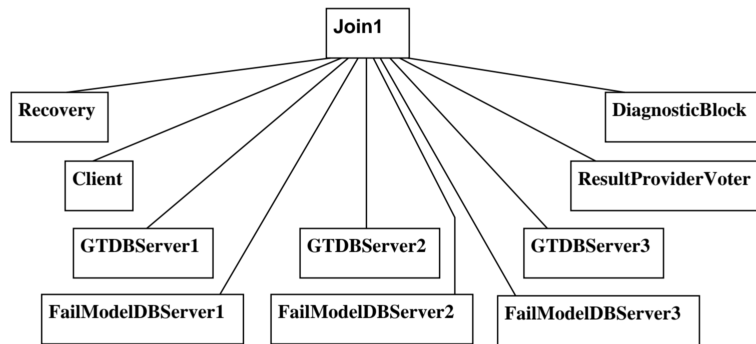

Fig. 3. Composed Model of the System

three replicas of the DB server and the GatewayThread processes (used to parallelize the activity of the Gateway. The SANs FailModelDBServer1, FailModelDBServer2, FailModelDBServer3 represent the failure behaviour of each channel. The SAN ResultProviderVoter represents the time and the actions of the ResultProvider (which receives the result sets from the GatewayThreads and delivers them to the Voter), and the Voter. The SAN DiagnosticBlock represents the behavior of the TBM.

For the sake of brevity, in the following only two of the sub-models are described in detail. Figure 4 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 the place nRecords1, nRecords2, nRecords3. The activity Recovery is enabled by the DiagnosticBlock (which triggers the recovery by putting a token in the place recovery) and the Client models (which remove a token by the token 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 DataBase or due to the restarting of the application and the OS of the three channels (after restoration all the OS of the channels are restarted). The

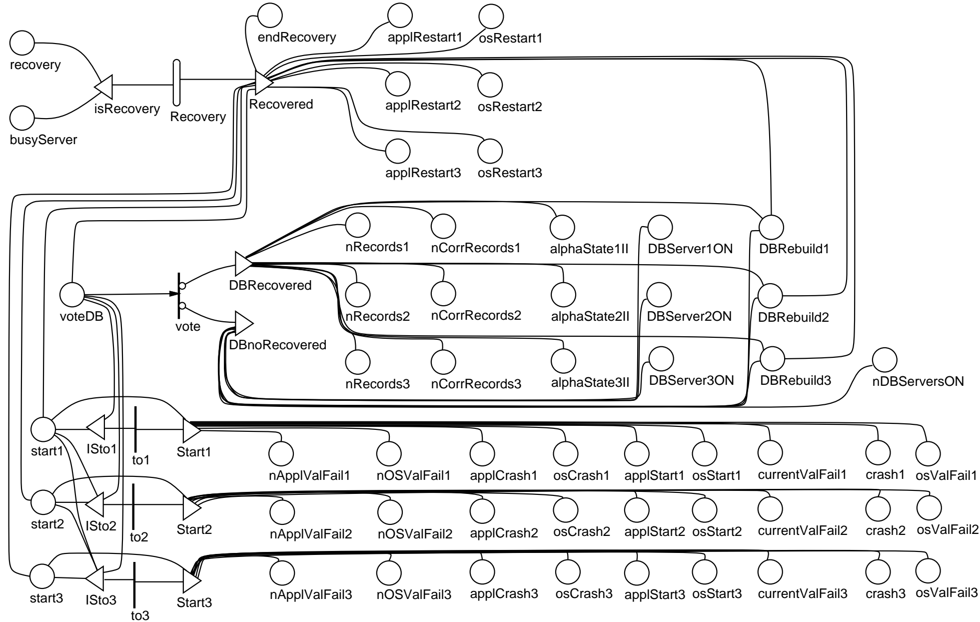

Fig. 4. SAN 'Recovery'

output gates DBRecovered and DBnoRecovered represent the marking changes for modeling the success or the failure of the restoring of the DataBase, respectively. The output gates Start1, Start2, Start3 represent the marking changes for modeling the restarting of the application and the OS of the three DB servers. Figure 5 depicts the SAN 'FailModelDBServer1'. The Lognormal (or Weibull) distributed activities ApplFail and OSFail represent the times to failure of the application and of the OS, respectively. The two associated cases represent the probability of crash (case 1) and of value failure (case 2). After the i-th failure, the time to next failure is reduced by using a distribution with a mean equal to the original one divided by i. ApplFail and OSFail are restarted with the original distributions after each restart of the application or of the OS, respectively.

5 System Evaluation

This section describes the results obtained so far on parameter tuning and overall system dependability via combined use of direct measurements on the system prototype and analytical modeling. Table 2 reports the main parameters of the SAN model and their default values. These, together with the ones reported in Table 1, which were extracted from direct measurements, were used to populate the analytical model. For all parameters, the higher extreme of the range measured has been used as the reference value for that parameter in the evaluation of the SAN model. Our analysis consists of two main parts. The first part shows

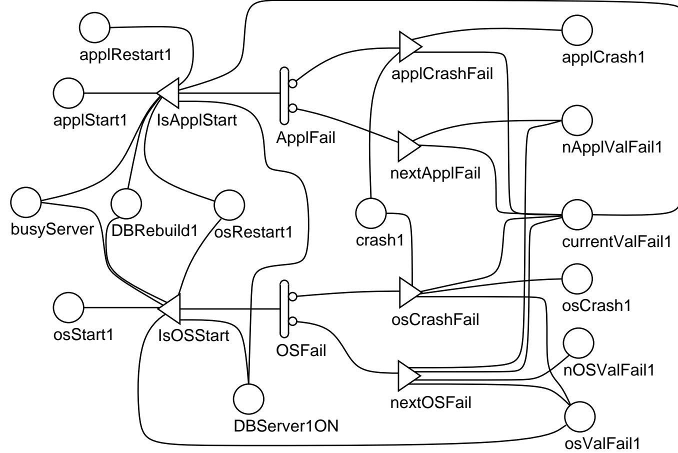

Fig. 5. SAN 'FailModelDBServer1'

how to derive indications about tuning the parameters of the alpha-counts in order to reach good performance of the fault treatment strategy. The second part reports some measures of system availability obtained using the SAN model.

5.1 Tuning of the parameters

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 alpha-count and since the records can be corrupted, but cannot be corrected by a service request, it must be 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 this 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. Under these assumptions, a good approximation of the probability p_k that there are "k" erroneous records after executing the recovery procedure is obtained using a binomial distribution:

$p_k = \binom{N}{k} q^k (1-q)^{N-k}$, where:
$q = q_1q_2q_3 + q_1q_2(1-q_3) + q_1(1-q_2)q_3 + (1-q_1)q_2q_3$
$q_i = \frac{N_i^e}{N}$
N_i^e=number of erroneus records of the Channel_i

Figure 6(a) plots p_0, i.e. the probability of correctly restoring all the records of the database, for $N = 10000$. Having in mind the desired probability of suc-

Parameter	Default value
KI (1st alpha-count)	0.99
TI (1st alpha-count)	$2, 3$
KII (2nd alpha-count)	1
TII (2nd alpha-count)	3
Probability of Application Crash	0.75
Probability of OS Crash	0.75
N: nr. of records in the Database	10000
Request rate $[s^{-1}]$	0.005
Probability of an Update Request	0.2
Probability of a Query Request	0.8
μ_A, μ_O (par. of the Lognormal)	14
α_A, α_O (par. of the Lognormal)	0.6
mean (of the Lognormal) [days]	16.7
variance (of the Lognormal) [days]	12651129

Table 2. Parameters and their default values used in the evaluation

cessfully restoring one replica of the database, one can now relate the number of corrupted records to the threshold of the second alpha-count which is used to trigger the recovery procedure. Figure 6(b) reports the number of erroneous records of a replica of the database as a function of T_{II} as estimated with our SAN model of the system. Figure 6(b) shows that, in the settings chosen, the

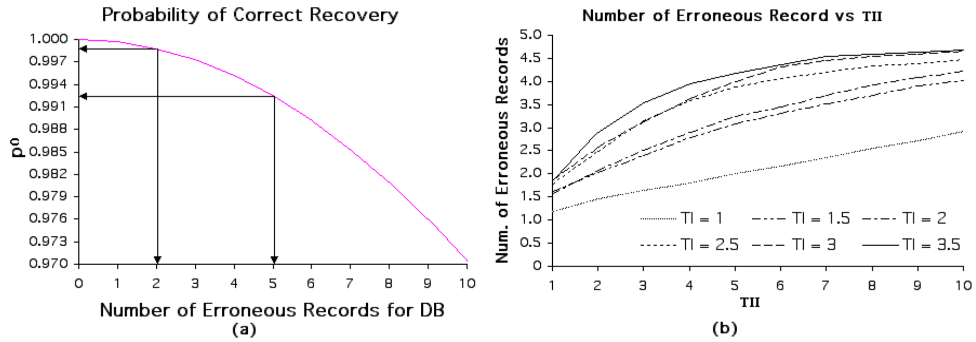

Fig. 6. (a) Probability of success of the recovery procedure as a function of the number of corrupted records, (b)Number of corrupted records as a function of the T_{II} of the second alpha-count for different values of T_I (KI=0.995)

number of corrupted records is always very low ranging from about 2 for low values of T_{II} to 5 when TII goes to 10. In more details, for the same T_{II} the higher is TI the higher the number of corrupted records. Figure 6(a) shows that

in the range 2-5 of the number of corrupted records the probability of correct recovery varies from .999 to .992. However this should not be a concern: chances of correct recovery are higher than .99.

5.2 Availability

We considered a time frame of one year, and we measured the cumulated time in which the system is available for providing services i.e., the availability. Due to our will to account for many non exponential events (e.g., the failure process modeled as a lognormal and the duration of the recovery procedures modeled as constants), the SAN models were solved by simulation with UltraSAN [22]. Figure 7 reports the expected availability of the system in a one year period as a function of the value of the parameter KI of the first Alpha-count for several values of T_I the threshold of the first alpha-count. T_{II} the threshold of the second alpha-count is fixed to 3. The availability is computed accounting both for the time spent in doing recovery and for a potential complete failure of the system.

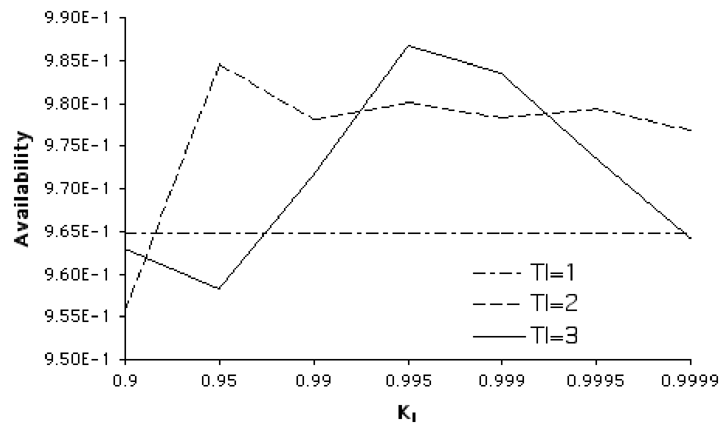

Fig. 7. Expected availability of the system in one year, as a function of KI the parameter of the first alpha-count for several values of TI (TII=3)

The figure shows that setting T_I =1 gives a constant availability: in this case the value of K_I does not play any role. Instead when T_I assumes values greater than 1 the impact of K_I on the availability can be noticed. Given a value for T_I one can notice that for increasing values of K_I the availability first increases until it reaches a maximum and then becomes worse. The utility of the first alpha-count and of the distinction between the first two recovery actions can be noticed by considering that the best combination of T_I and K_I gives better availability than the case with T_I =1.

Figure 8 reports the expected unavailability of the system in a one year period as a function of the value of the second threshold T_{II} for three different values

(1, 2 and 3) of T_I the first threshold and K_I fixed to 0.99. The unavailability is computed accounting for both the time spent in doing recovery and the outage due to complete failure of the system.

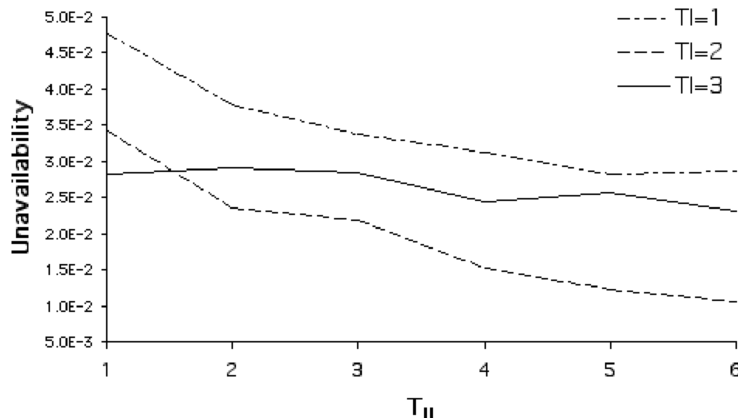

Fig. 8. Expected availability of the system in one year, as a function of TII (the threshold of the second alpha-count) for TI=1, 2 and 3 assuming KI=0.99

The point T_{II}=1, T_I=1 shows the unavailability of the system without our fault treatment mechanisms: all the recovery actions are performed at each error detection. The figure shows that the system improves for growing values of T_{II}. The best availability is obtained with $T_I = 2$. In this scenario setting $T_I = 3$ appears to be not very rewarding, the unavailability remains approximately constant, probably because too many errors affect the database before restoration is performed. Overall the figure shows that with a good tuning of the parameters using our fault treatment strategy one can reduce 5 times the unavailability from .05 to .01.

6 Conclusions and Future Work

This paper described a fault-tolerant distributed legacy-based system which has been implemented as a middle-tier based architectural framework to leverage the dependability of an existing application. 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. Three different kinds of faults in the application have been considered: i) hardware-induced software errors in the application; ii) software errors (i.e., process and/or host crashes or hangs); and iii) errors in the physical database (i.e., corrupted data in the system stable storage). The system is organized

as a TMR and failure data collected by the voter is provided to the diagnostic subsystem, consisting of a filtering unit and a threshold-based component (TBM). The TBM uses filtered failure data to update three alpha-count pairs, one for each replica of the back-end servers. The two alpha-counts composing individual pairs are used to discriminate among alternative recovery actions. We considered three possible recovery actions: Action 1 (application restart), Action 2 (host restart), and Action 3 (data base restoration). In order to evaluate our system i.e., the effectiveness of the proposed fault-treatment strategy and to tune the parameters of its mechanisms, we developed a SAN model of the overall system consisting of ten sub-models joined together. We performed several evaluations by simulation in order to account for non exponential events. Values for the model parameters were extracted from direct measurements on the system prototype. The analysis, although still preliminary, 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 projects: FIRB "WEB-MINDS: Wide-scalE, Broadband, MIddleware for Network Distributed Services", "Strumenti, Ambienti e Applicazioni Innovative per la Societá dell'Informazione" , SOTTOPROGETTO 4, and Legge 449/97 Progetto "SP1 Reti Internet: efficienza, integrazione e sicurezza".

References

1. Microsoft Corporation (2002), NET Framework Reference, http://msdn.microsoft.com/netframework/techinfo/documentation/default.asp
2. B. Shannon (2002), Java 2 Platform Enterprise Edition Specification, v1.4, http://java.sun.com/j2ee
3. J. Arlat, J.-C. Fabre, M. Rodrguez, F. Salles, Dependability of COTS Microkernel-Based Systems, IEEE Transactions on Computers, 2002 (Vol. 51, No. 2)
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.

9. 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.
10. 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.
11. A. Avizienis, J.C. Laprie, and B. Randell, Fundamental Concepts of Dependability, LAAS, Technical Report n.ro 01145, Tolosa (France), Technical Report n.ro 01145 2001.
12. 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.
13. 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.
14. O.M. Group, Fault-Tolerant CORBA Specification, v1.0, OMG, http://www.omg.org, document ptc/00-04-04 2001.
15. 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.
16. 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.
17. 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.
18. 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.
19. D.E. Bakken, Z. Zhan, C.C. Jones, D.A. Karr, Middleware support for voting and data fusion, presented at DSN01- IEEE International Conference on Dependable Systems and Networks, Gotenburg, Sweden, 2001, pp. 453–462.
20. DBench Consortium, Measurements, Deliverable ETIE1, IST-2000-25425 Dependability Benchmarking (DBench), 2002.
21. 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.
22. W. H. Sanders and J. F. Meyer, A Unified Approach for Specifying Measures of Performance, in Dependable Computing for Critical Applications, vol. 4 of Dependable Computing and Fault-Tolerant Systems, A. Avizienis, H. Kopetz, and J. C. Laprie, Eds.: Springer Verlag, 1991, pp. 215-237.
23. 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.
24. 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.