A Recovery-Oriented Approach for Software Faults Diagnosis in Complex Critical Systems

Gabriella Carrozza
SESM S.c.ar.l., Finmeccanica company, Italy

Roberto Natella
Università degli Studi di Napoli Federico II, Italy

ABSTRACT

This paper proposes an approach to software faults diagnosis in complex fault tolerant systems, encompassing the phases of error detection, fault location, and system recovery. Errors are detected in the first phase, exploiting the operating system support. Faults are identified during the location phase, adopting on a machine learning approach; this phase then triggers the proper recovery action for the occurred fault - actuated in the third phase. Feedback actions are also adopted in the location phase to improve detection quality over time. A real world application from the Air Traffic Control field has been used as case study for evaluating the proposed approach. Experimental results, achieved by means of fault injection, show that the diagnosis engine is able to diagnose faults with high accuracy and at a low overhead.

Keywords: Fault Diagnosis, Machine Learning, Software Dependability

INTRODUCTION

Hardware and software technologies are progressing fast, dramatically increasing the complexity of modern computer systems. Even in the context of critical scenarios, we are witnessing a paradigm shift from stand-alone and centralized systems toward large-scale and distributed infrastructures; simple monolithic programs are being superseded by modular software architectures, typically based on Off-The-Shelf (OTS) software items. This allows organizations to be competitive by reducing development costs and the time to market. Testing and verification activities along with fault tolerance techniques are massively used to satisfy dependability requirements. The key for achieving fault tolerance is the ability to accurately detect, diagnose, and recover from faults during system operation.

The great research effort striven in fault tolerant systems has provided good results with respect to hardware-related errors. Recent examples are (Serafini, Bondavalli, & Suri, 2007) (Daidone, Di Giandomenico, Chiaradonna, & Bondavalli, 2006) (Bondavalli, Chiaradonna, Cotroneo, & Romano, 2004). However, it is well known that many systems outages are due to software faults (Gray, 1985). These are bugs lying into the code, hence they are permanent in nature. This means that, if a program contains a bug, any circumstances that cause it to fail once will always cause it to fail. This is the reason for which software failures are often referred to as “systematic failures”

(Littlewood & Strigini, 2000). However, the *failure process*, i.e., the way the bugs are activated, is not deterministic since (i) the sequence of inputs cannot be predicted, hence it is not possible to establish which are the program's faults (and failures), and (ii) software failures can be due to environmental conditions (e.g., timing and load profile) which activate a given fault rather than another one. For this reason, it is said that software faults can manifest transiently. By failure we intend the software modules/components failure in which the fault has been activated. This can be viewed as fault from the whole system point of view (Joshi, Hiltunen, Sanders, & Schlichting, 2005). Activating conditions which cause a software fault to surface into a failure have been recognized to be crucial in (Chillarege, et al., 1992), where they are defined as "triggers" and where software bugs are grouped into orthogonal, non overlapping, defect types (Orthogonal Defect Classification, ODC). Software faults which manifest permanently, also known as *Bohrbugs*, are likely to be fixed and discovered during the pre-operational phases of system life cycle (e.g., structured design, design review, quality assurance, unit, component and integration testing, alpha/beta test), as well as by means of traditional debugging techniques. Conversely, software faults which manifest transiently, also known as *Heisenbugs*, cannot be reproduced systematically (Huang, Jalote, & Kintala, 1994), and they have been demonstrated to be the major cause of failures in software systems, especially during the system operational phase (Oppenheimer & Patterson, 2002) (Sullivan & Chillarege, 1991) (Chillarege, Biyani, & Rosenthal, Measurement of Failure Rate in Widely Distributed Software, 1995) (Xu, Kalbarczyc, & Iyer, 1999).

Focus in this work is on recovery oriented software fault diagnosis in complex fault tolerant systems. Little attention has been paid so far to this problem, which plays a key role in maintaining system health and in preserving fault tolerance capabilities. Previous studies on software diagnosis aimed to identify software defects from their manifestations through off-line and/or on-site analysis (Tucek, Lu, Huang, Xanthos, & Zhou, 2007). They aim to discover bugs in the code, by using static/dynamic code screening, in order to perform more effective maintenance operations. Thus, they are not able to catch Heisenbugs, in that environmental factors which caused them are not easy to be localized into the code.

In this work, the aim of diagnosis is twofold. First, starting from outward symptoms we are interested in identifying what are the execution misbehaviors which caused failure occurrence, and where these misbehaviors come from, in order to trigger proper recovery actions. This is crucial in complex, modular and distributed systems, for which the overall failure can be avoided by confining, and masking, the individual failures of the parts (nodes, components, processes). Second, we aim to provide information about manifested symptoms which are useful for off-line maintenance activities.

The massive presence of OTS items, whose well-known dependability pitfalls do not hold industries back from their usage in critical systems, further exacerbates the diagnosis problem. In fact, faults can propagate in several ways and among several components, depending on a complex combination of their internal state and of the execution environment. Actually, the failures which result from unexpected faults, known as *production run failures* or *field failures* (e.g., crashes, hangs and incorrect results), are the major contributors to system downtime and dependability pitfalls (Tucek, Lu, Huang, Xanthos, & Zhou, 2007). They cause the system failure modes to be not known at design time, and to evolve over time. To face this problem, we

believe that the detection has to be included into the diagnosis process, differently from most existing approaches. As it has been demonstrated in (Vaidya & Pradham, 1994) with respect to distributed systems recovery from a large number of faults, by combing detection and location adaptively, the number of diagnosed faults increases over time at a low additional cost.

Addressed issues

This work defines a recovery-oriented diagnosis approach, to (i) locate the cause of a system's component failure at run time, and (ii), trigger proper recovery actions, based on an estimate of the fault nature, in the form of fault tolerance via fault masking techniques.

Several issues arise when designing this approach, which have not been addressed before. First, the presence of software faults hampers the definition of a simple, and accurate, mathematical model able to describe systems failure modes (hence, pure model based techniques become inadequate). Second, due to the presence of OTS components, low intrusiveness in terms of source code modifications is desirable. Third, diagnosis has to be performed on-line, i.e., a fault has to be located as soon as possible during system execution and with lack of human guidance. The reason for this is twofold. On the one hand, it is to fulfill strict time requirements, for system recovery and reconfiguration, in the case of a fault. On the other hand, it is to face system complexity with respect to ordinary system management and maintenance operations, whose manual execution would result in strenuous human efforts and long completion times.

Paper's contributions

This paper proposes a novel approach for on-line software fault diagnosis in complex and OTS based critical systems. To the best of authors' knowledge, this is the first proposal which addresses on-line software diagnosis issues of complex and fault tolerant systems, and evaluates in the context of a real world industrial application. Indeed, differently from most of the previous work which proposed off-line/on-site diagnosis approaches aiming to locate bugs in the source code, and sometimes the environmental conditions, we address the diagnosis problem during the system operational phase. Thus, two are the key aspects:

- *low intrusiveness*: since we are addressing OTS-based systems, it is required that target applications is not instrumented at all
- *holistic diagnostic approach*: Detection (D), Location (L), and Recovery (R) have been integrated into one diagnosis process.

More specifically, the approach features:

- A detection strategy in charge of detecting application errors by exploiting OS support, exhibiting high accuracy and low overhead. To face the partial ignorance about the failure modes, the detection strategy is pessimistic and relies on the anomaly detection paradigm. To the best of authors' knowledge, this paradigm has not been applied yet for fault/error detection in critical systems. Experimental results reveal that this approach is promising.

- A novel fault location strategy in charge of (i) locating the cause of system components/modules failures, and (ii) of triggering the most effective recovery action on the basis of the detection output. More important, it is also in charge of solving detection falls and of improving detection quality over time by means of feedback actions.
- The design of a recovery dictionary in charge of associating automatically the most effective recovery mean to the occurred fault. This is also the main objective of (Joshi, Hiltunen, Sanders, & Schlichting, 2005). Even if it is close in spirit to our work, that work aims to optimize the sequence of recovery actions to be applied, whereas we apply only the best one, by using monitors we implemented by our own.

SYSTEM MODEL AND ASSUMPTIONS

The target systems are assumed to be complex software systems, deployed on several nodes and communicating through a network infrastructure. Each node is organized in software layers and it is made up of several *Diagnosable Units* (*DUs*), representing *atomic* software entities, at it is shown in Figure 1.

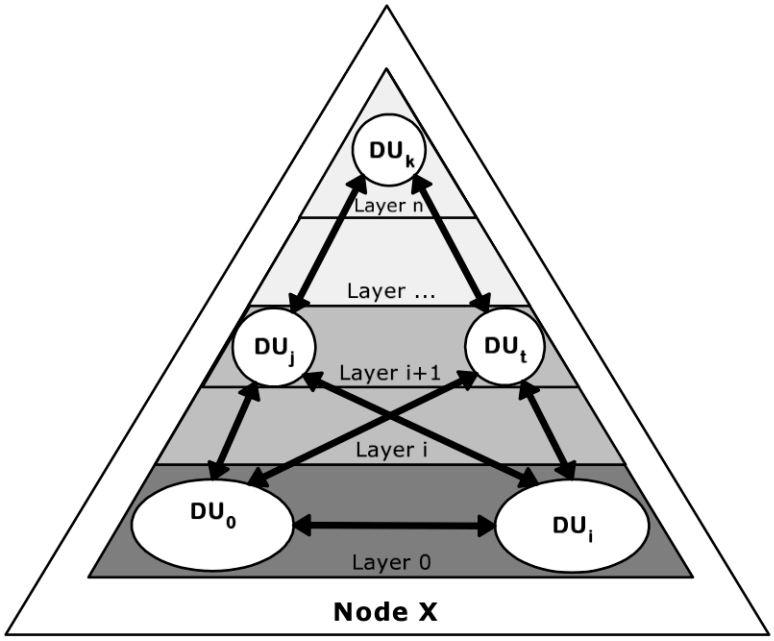

Figure 1: System's node model

In most of the cases, the layered structure of each node encompasses the Operating System (OS), the Middleware and the User Application levels. Since the focus is on software faults, such a structure has been adopted, thus excluding the underlying hardware equipment. Such an assumption sounds reasonable since modern systems are equipped with redundant and highly reliable hardware platforms which are developed and extensively tested in house, especially in the case of mission and safety critical systems. This means that hardware related faults will be not diagnosed by the proposed DLR framework.

DUs are assumed to be processes. This means that a process is the smallest entity which can be affected by a fault and for which it is possible to diagnose faults, as well as to perform recoveries. Of course, the bug which caused the process to fail can be located within an OTS library or module which is being executed in the context of the process; additionally, propagations can occur among different nodes and layers. Look at Figure 2, where the process P_1 experiences a failure due to the component *C*. However, the failure is actually located into the *D* library, which is running in the context of a different process, P_2 and the bug propagates to *C* through *y*, e.g., due to an erroneous input from *D* to *y*. According to a recovery-oriented perspective, addressing the process as the atomic entity of the system, it is enough to identify the cause which induced the failure of the process, within the context of the process itself. In other words, if a recovery action exists in charge of recovering the failed process by only acting on it, it is unnecessary to go back through the propagation chain out of the context of the process. With respect Figure 2, the failure of P_1 will be attributed to *y*, which is the last link in the propagation within the P_1 context.

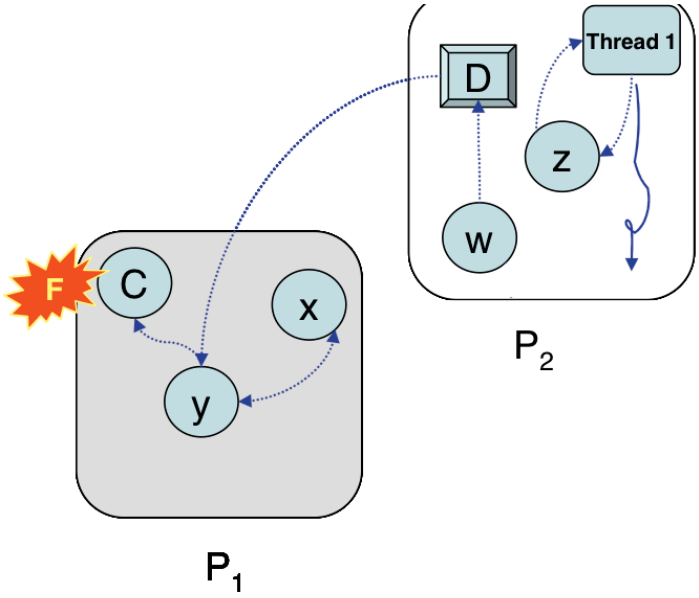

Figure 2: Diagnosis at process level.

Once the root cause has been identified the proper recovery action has to be selected. Hence, the final output of diagnosis consists of a couple of vectors *(D, R)*. The former associates the failed node, by means of the IP address, to the failed process which is identified by the Process ID (*PID*). The latter, instead, associates the experienced failure (f) to the recovery action to be initiated (r). Schematically

$$D = (IP_{failednode}, PID_{failedprocess})$$
$$R = (Failure_f, Recovery_r)$$

The diagnosis output provides information about the failed process, rather than about the component which caused the failure. This information would not be interesting for the final users. However, it could be helpful for bug fixing and fault removal.

Failure modes

Crash, hangs and workload failures are encompassed by the proposed approach. A process crash is the unexpected interruption of its execution due either to an external or an internal error. A process hang, instead, can be defined as a stall of the process. Hangs can be due to several causes, such as deadlock, livelock or waiting conditions on busy shared resources. As for workload failures, they depend on the running application. Workload failures can be both value (e.g., erroneous output provided by a function) or timing failures.

Since the target systems are distributed on several nodes, and since faults can propagate, the set of the failures to be encompassed is given by $FM=F \times DUs$, i.e., by the product set of the failure types and of all the *DUs* (i.e., the processes).

Recovery actions

Since the focus of this paper is on diagnosis, we assume that some basic Fault Tolerant services are provided by the middleware layer. For instance, in our industrial case study the used middleware implements the standard Fault Tolerant OMG CORBA service (OMG, 2001). Thus, the middleware is able to manage server replication, including issues related to state transfer which usually follow system reconfiguration procedures. The proposed DLR framework encompasses two classes of recovery actions:

- **System Level Recovery**, i.e., actions which aim to repair a failed process by acting at system level. These actions are intended for dealing with crashes and hangs, and they can be more or less costly depending on the size of the system, as well as on the number of processes involved into the failure. Encompassed actions are system reboot, application restart and process kill. Once one of these actions has been performed, the FT middleware service will be able to restore the application.
- **Workload Level Recovery**, i.e., action which aim to repair application failures. These actions are intended for dealing with workload failures, hence a knowledge of the application semantic is required.

THE OVERALL APPROACH

Figure 3 gives an overall picture of the proposed approach, representing how it works from the fault occurrence till system recovery.

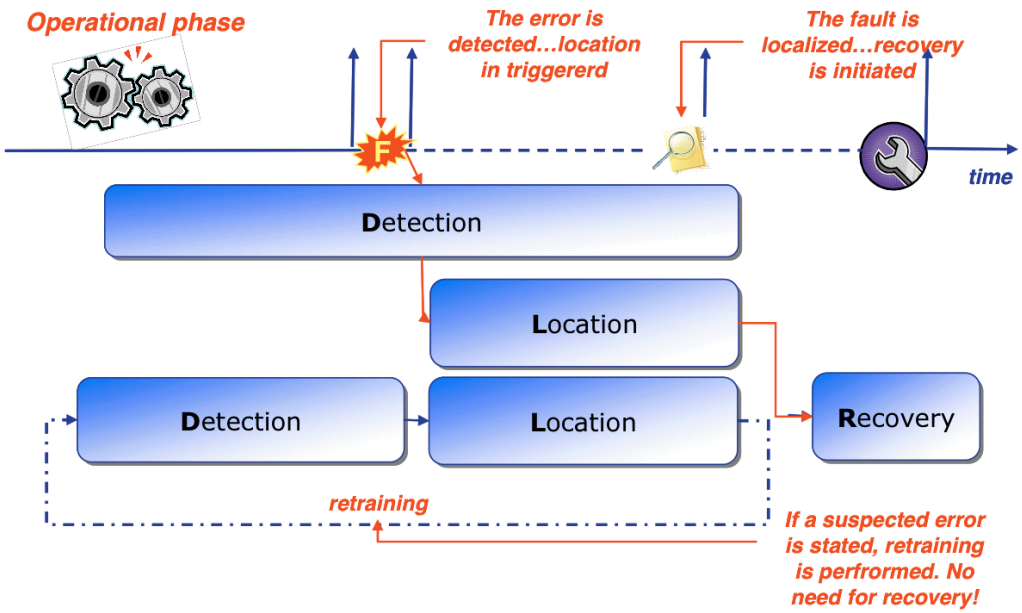

Figure 3: A time snapshot of the overall DLR approach.

During the operational phase of the system, a monitoring system performs continuous detection on each DU, exploiting Operating System (OS) support. Once a failure (*F*) occurs, an alarm is triggered. This initiates the Location phase, aiming to identify the root cause of the failure. Once the Location has been completed, the Recovery phase is started in order to recover the failed process(es) and to resume normal activities. The task of detection consists of the alarm triggering when a given process fails. Since if a process fails it is not assured that the system will fail as well, a process failure is conceived as an error for the overall system. For this reason, the task of detection is in fact *error detection* in the context of this work.

The overall approach is based on the machine learning paradigm, as in many previous papers focusing on diagnosis (Yuan, et al., 2006) (Zheng, Lloyd, & Brewer, 2004) (Brun & Ernst, 2004). The main reason for this is the presence of field failures, which cannot be known at design/development time. Indeed, such a paradigm makes the DLR engine, and all of its components, able to learn over time. Thus, field failures influence the design of the entire engine, from detection to recovery.

As for detection, an error is defined as a corruption in the state of a *DU*, which can impact in turn on the state of the system. An alarm is triggered each time an anomaly is encountered in system behavior; this is achieved by means of anomaly detection, i.e., all the conditions which deviate from *normal* behaviors are labeled as *errors*. This is quite a pessimistic detection strategy. In fact, not all the anomalies correspond to actual errors, i.e., this way errors can be signaled even when the system is behaving correctly but that working condition has not been recognized as normal. On the one hand, such a pessimistic strategy leads to a non negligible amount of false positives, in that alarms are likely to be triggered which do not correspond to actual errors. On the other hand, it allows to minimize the number of anomalous conditions which are misinterpreted as normal behaviors, thus going unnoticed. This is crucial in the context of critical systems in that unsignaled errors are in fact false negatives which may have

catastrophic effects. It is worth noting that reducing false positives, i.e., improving detection accuracy, at design time has been the primary requirement of traditional detection system.

Once an alarm has been triggered, the Location phase is initiated to identify its root cause. Along with the aim of pinpointing the actual fault, this phase has also to remedy detection accuracy falls. More precisely, during this phase the presence of an actual fault has to be established, since false positives are likely to be triggered by the detector. This means that the location module behaves ``distrustfully'' to compensate the pessimistic detection. This is achieved via the machine learning paradigm, which underlies this phase in the form of classification. We separate faults into smaller classes, and give criteria for determining whether an occurred fault is in a particular class or not.

Each fault class is represented as a set of features, i.e., a set of measurable properties (which are inferred from OS and produced logs) of the observed *DU*. We adopt a pattern recognition module which gathers observed features during system operation to be classified. Features are determined experimentally; the relation between features and faults is therefore learned (or trained) experimentally as well, and then stored to form an explicit knowledge base. Faults can be identified by comparing the observed values of the features with the nominal ones.

Starting from manifestations (i.e., the errors), the location module has to infer the presence of a fault and to associate it to a class. To design the fault classes properly, three circumstances has to be considered:

1. *SUSPECTED ERROR (SE)*: the triggered alarm was not the manifestation of an actual fault, i.e., the detection module triggered a false positive. In this case, there is no need neither for location nor for system recovery;
2. *ERROR*: a fault actually occurred that the location module is able to identify. In this case, recovery actions have to be associated to the fault and initiated as soon as possible;
3. *UNKNOWN FAULT*: the triggered alarm was actually due to a fault which cannot be identified during the location. This is the tricky case of a fault which is *unknown*, i.e., a fault which never occurred before. In this case, the system has to be put in a safe state, and further investigations are needed which can potentially require human intervention.

The location capability of uncovering false positives allows to improve the detection accuracy. Actually, this is the aim of the feedback branch, namely “retraining”, depicted in Figure 3: once an alarm has been labeled as a SE, the detection module is upgraded consequently. This allows a reduction of the number of false positives over time, as it will be shown in the following sections.

Recovery actions to be initiated in the case of an ERROR have been associated to the fault classes. This is to perform recovery actions which are tailored for the particular fault that occurred. Since the approach is intended for operational systems, two main phases are encompassed. During the first phase, the DLR engine is trained in order to build a *starting knowledge*. This is leveraged during the second phase, which is in fact the system operational phase.

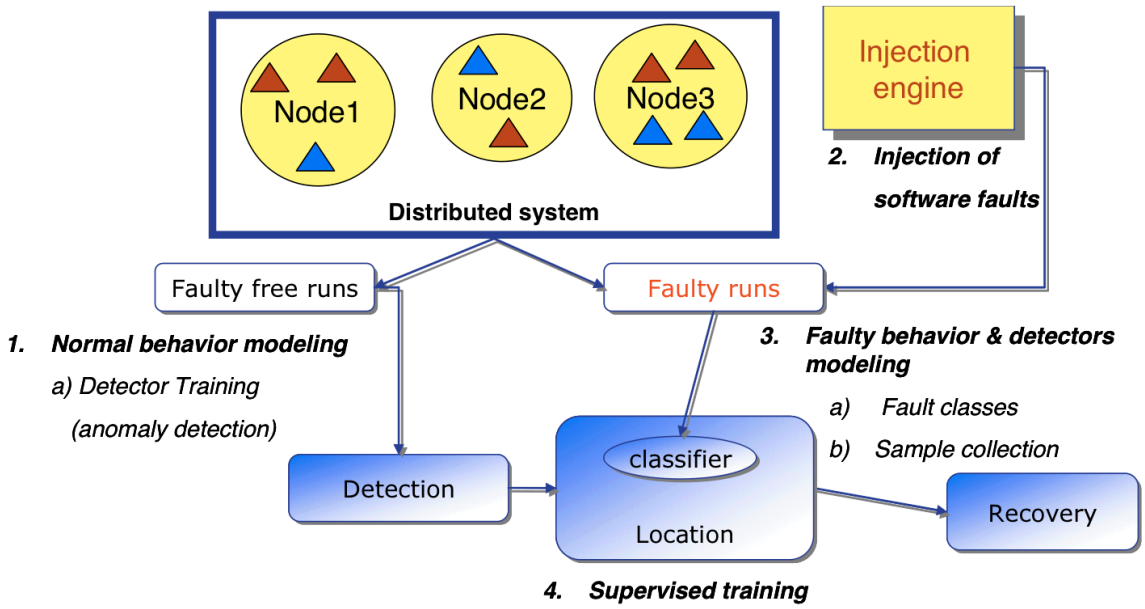

Figure 4: Training of the DLR engine

Figure 4 depicts the training process. In order to train the detection module, which performs anomaly detection, faulty free executions (i.e., correct executions) of the system have to be run in order to model its normal behavior (1). Conversely, fault injection (2) is required in order to allow (i) the definition of the fault classes (3.a) and (ii), the collection and analysis of fault related data to model system behavior in faulty conditions (3.b). DUs running into the system, which are depicted as little triangles in the figure, are the injection target. Once faults have been injected, the supervised training of the location classifier is performed (4), as detailed in section *location*. At the end of the training phase, both the detection and the location classifiers can rely on a starting knowledge about the target system. On the one hand this can be exploited during the operational phase. On the other hand, the base knowledge has to be improved during the system lifetime adaptively in order to take fields failures into account.

DETECTION

Good detection systems have to exhibit low overhead, i.e., they do not have to compromise system resources, as well as low rates of missing and wrong detections. This is in order to reduce the effects of a not handled fault, and the number of false alarms as well. These requirements represent the most challenging issues when dealing with detection in several fields. As for example, for intrusions detection within networked systems, performance overhead increases with the traffic volume, and users behavior is difficult to characterize (i.e., licit users can behave in unexpected manner thus being misinterpreted as attackers). Anomaly detection is the most common and effective way to detect attacks (Becraft & Lee, 1993) (Forrest, Hofmeyr, Somayaji, & Longstaff, 1996). However, to the best of authors' knowledge it has not been used yet for error detection in critical, dependable, systems.

In this work we use anomaly detection for error detection. Simply, if the monitored application deviates from the normal behavior (i.e., faulty free executions), it is likely to be faulty. Normal behavior is modeled in two steps. First, the identification of representative and synthetic parameters; second, a training phase during which these are traced and characterized. Since it is influenced by several factors, such as the hardware configuration and usage application profile, the training phase should be repeated (manually or automatically) to take into account their significant variations. Once the normal behavior has been modeled, the application is let run on the field and it is continuously monitored. When deviations are encountered, i.e., when parameters values differ from the modeled ones, the location phase is initiated aiming at pinpointing the root cause of the error, as well as at uncovering and fixing detection falls, as explained in the previous section.

In fact, *DUs* are the finest grain for detection, i.e., the above described detection process is applied to all the application *DUs*. Although the detection process can be applied to several kinds of *DUs* in principle, the way the detection is actually performed depends on their nature. As stated in section II, in this work *DUs* are assumed to be OS processes, also seen as "collection" of several threads. This is a very common abstraction used by computer systems. The behavior of a process can be effectively described through its interactions with the OS resources, and with all the other running processes as well. For this reason, we decided to trace these interactions by means of several *monitors*. Since target systems are OTS based, *monitors* require neither any internal knowledge of *DUs* nor their source code availability, i.e., *DUs* are considered as black boxes.

Parameters and detection criteria

The following OS data, which can be to collected and analyzed with a low computational overhead, are monitored to describe *DUs* behavior:

1. SIGNALS, i.e., notifications produced by the OS when providing services to the application (e.g., numeric return codes returned by system calls, UNIX signals). Erroneous OS notifications are logged (e.g., return codes different than zero, which represent exceptional conditions and are relatively rare);
2. TIME EVENTS, i.e., the timestamp of interesting events for the application execution (e.g., when a given resource becomes available, such as a semaphore). A log entry is produced each time these events do not occur within a given interval, i.e., timeouts are exceeded;
3. THROUGHPUT, i.e., the amount of data exchanged by OS processes through I/O facilities (e.g., the throughput of network and disk devices). Upper and lower bounds are associated with the I/O throughput; throughput is periodically sampled, and a log entry is produced when bounds are exceeded.

Architecture and strategy

The architecture of the detection subsystem implemented in this work is depicted in Figure 5. It is a modular system made up of several, simple, *monitors* which are combined to provide a

detection alarm. This is in order to get the most from each of them, in that monitors exhibit different performances in terms of coverage (i.e., the ability to detect an actual fault) and accuracy (e.g., the ability to avoid false alarms). Additionally, by combining monitors' responses the total number of false positives (e.g., a timeout can be exceeded due to a particular overload condition which is not an actual fault), can be minimized.

Being *P* the number of processes within the monitored application, monitors are associated to each thread t_j of the *P* processes. Hence they account for a total of *P* x t_{jk} ∀ *k=1..n*, where *n* is the number of monitored parameters. Monitors evaluate a triggering condition periodically (T is the period). An alarm generator (α_i) collects the output of all the monitors related to a given parameter *p* for the *i*-th process. A counter *n* is incremented by α_i if the monitored thread verifies the triggering condition. The normal behavior of a process (and of a thread as well), is modeled by associating a range of licit values to each alarm generator, specifically $r_i = [r_i^-, r\_i^+]$.

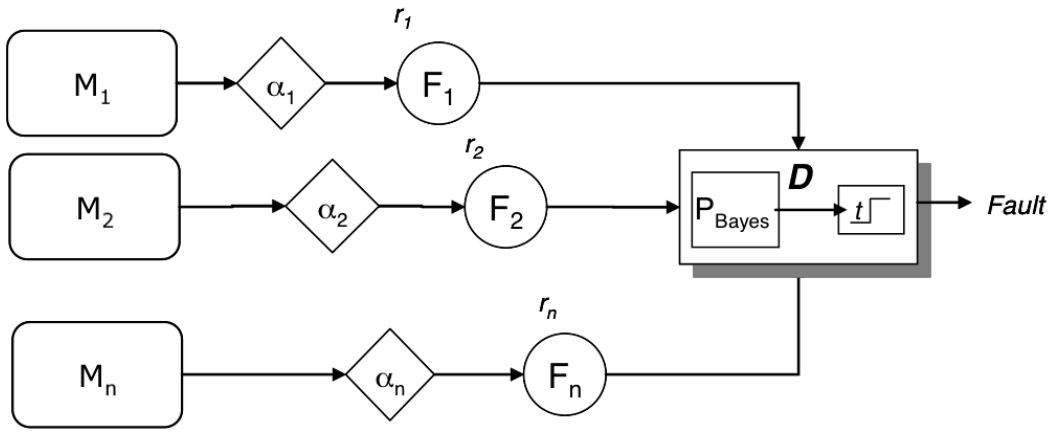

Figure 5: Detection architecture.

In practice, if the monitored value *n* for a given parameter *p* does fall outside r_i within a temporal window *T*, an alarm is triggered. Hence, the output of each α_i is a binary variable defined as:

$$F_i = \begin{cases} 1 & \text{if } n \notin r_{in} \text{ in T} \\ 0 & \text{otherwise} \end{cases}$$

The range r_i is tuned during the preliminary training phase. The number of events is periodically sampled: the minimum and the maximum values which are experienced during faulty-free runs constitute the limits of the range.
The Bayes' rule has been chose as the combination rule, hence the probability of a fault is achieved by:

$$P(F \mid \underline{a}) = \frac{P(\underline{a} \mid F)P(F)}{P(\underline{a} \mid F)P(F) + P(\underline{a} \mid \neg F)(1 - P(F))}$$

An alarm is triggered, if the estimated *a posteriori* probability exceeds a given threshold. In the previous equation, *F* represents the event "faulty *DU*", and $\underline{a}$ is a vector containing the output of the alarm generators α_i: if $F_i = 1$ for *L* consecutive periods *T*, then $\underline{a}_i = L$, in order to take into account the alarm duration and to filter out "transient" false alarms (i.e., alarms triggered for only a short amount of time). Note that we assume that monitors do not fail (i.e., produce false alarms) at the same time. The joint probability distributions $P(\underline{a} \mid F)$ and $P(\underline{a} \mid \neg F)$, i.e., the probability of detection and the probability of false alarms respectively (Cardenas, Baras, & Seamon, 2006), have to be estimated during the training phase. The former can be estimated using fault injection, by evaluating the number of occurrences of the $\underline{a}$ vector under faults, over the total number of vectors collected during fault injection. Similarly, the latter can be estimated by counting the number of occurrences of $\underline{a}$ during faulty-free executions. Finally, the *a priori* fault probability *P(F)* has to be known. If field data are available about past fault occurrences, *P(F)* can be estimated using the ratio *T / MTTF* (MTTF stands for Mean Time To Failure), i.e., on the average, the *DU* becomes faulty once every *MTTF / T* detection periods. Otherwise, *P(F)* can only be gathered by literature, e.g., by using typical fault rates of complex software systems.

This detection approach reveals to be less intrusive that traditional techniques, such as those based on heartbeat, which also require extra code to be written at application level and can fail in the case of multithreaded applications. Additionally, the proposed approach is able to exploit OS information which would be not available if remote detection were performed: as for example, it allows to discern the nature of a process stuck (e.g., deadlock or I/O waiting).

Monitors

The detection system has been implemented to be compliant with a POSIX operating system. In particular, we developed it under the Linux OS, and the following monitors have been implemented for controlling the detection parameters:

Time-related monitors:

- *Waiting time on semaphores.* The delay between time in which a task \footnote{A thread in the Linux jargon} requests for a semaphore and the time the semaphore is actually acquired is measured, for each semaphore and task. An exceeded timeout can be symptom of a deadlock between the threads in a process, or between several processes on the same node.
- *Holding time on semaphores.* The delay between the time in which the task has acquired a semaphore and the time the semaphore is released is measured for each semaphore and task. An exceeded timeout can be due to a process blocked within a critical section.
- *Task schedulation timeout.* The delay between the preemption of a task (e.g., when its time slice is exhausted and the CPU is granted to another task), and the next schedulation of the same task is measured for each task. This way, blocked tasks due to indefinite wait can be detected. For example, the block can be due to a fault within the task, or to the stall of the overall application (hence not only to deadlocks).
- *Send/receive timeout on a socket.* The delay between two consecutive packets sent on a given socket (both from and to the monitored task) is measured, for each task and socket. This allows to detect stop and omission error of a given task.

Signals related monitors:

- *UNIX system calls, UNIX signals.* Applications use system calls to forward requests to the OS (e.g., access to hardware devices, process communication, etc.). In UNIX systems, each system call can return a given set of codes which reflect exceptions when the system call exits prematurely. Error codes may be both due to hardware faults and to application misbehaviors (e.g., unavailable file or socket is accessed). Similarly, signals are used by the OS to notify exceptional events which are not related to a system call (e.g., memory denied memory access).
- *Task lifecycle.* The allocation and the termination of a task and its descendants are monitored. In fact, when a task terminates, either voluntarily or forcedly, it is deallocated by the OS, and an error code is returned to the parent process; a common rule is to return a non null code in if there is an error.

Throughput monitor:

- *I/O throughput.* This monitor takes into account performance failures which may affect the application. In fact, when the application is running in degraded mode (e.g., due to resource exhaustion or overloading), it can be observed an anomalous amount of data (either too low or too high) produced or consumed by the running tasks. In order to keep the overhead low, we considered a simple detection algorithm based on upper and lower bounds for the I/O transfer rate. Bytes read from and written to disks, as well as bytes to and from the network devices were taken into account. Disk and network operations (both in input and output) within the kernel were probed, and the amount of bytes transferred within a second is sampled periodically (we refer to a sample as *X(t)*). The bounds applied to each metric have to be chosen conservatively (i.e., out-of-bound samples are infrequent during normal operation), in order to reduce the amount of false positives. A reasonable way to choose the bounds is to profile the task for a long time period, and to establish the bounds on first-order statistics (i.e., mean and standard deviation) of I/O samples. In this case, the detection algorithm can be described as follows:

$$y = \begin{cases} 1 & \text{if } X(t) > m_X \text{ and } \left|X(t) - m_X\right| > k^+\sigma_X \\ 1 & \text{if } X(t) < m_X \text{ and } \left|X(t) - m_X\right| < k^-\sigma_X \\ 0 & \text{otherwise} \end{cases}$$

were m_X and σ_X are the mean and the standard deviation of the profiled samples during the training phase, k^+ and k^- are constants preliminarily set by the user (greater constants will lead to more conservative bounds). In order to take into account bursty and idle periods, a threshold *C* is chosen such that an error log entry is produced only if *C* consecutive out-of-bound samples occurs; *C* can be set to the maximum length of bursty or idle periods occurred during the training phase.

Monitors, which are summarized in table 1, have been implemented by means of *dynamic probing*: the execution flow of the OS is interrupted when specific instructions are executed (similarly to breakpoints in debuggers), and ad-hoc routines are invoked to collect data about the execution (i.e., to analyze OS resource usage by monitored applications). Note that dynamic probing was only used for measurement and detection purposes, and no attempt is made to modify kernel and processes execution.

Table 1: Monitors running on the Linux Operating System for fault detection.

Monitor	***Triggering condition***	***Parameters***	***Domain***
UNIX system calls	An error code is returned	Window length	Syscalls × ErrCodes
UNIX signals	A signal is received by the task	Window length	Signals
Task schedulation timeout	Timeout exceeded (since the task is preempted)	Timeout value	$[0, \infty]$
Waiting time on semaphores	Timeout exceeded (since the task begins to wait)	Timeout value	$[0, \infty]$
Holding time in critical sections	Timeout exceeded (since the task acquires a lock)	Timeout value	$[0, \infty]$
Task lifecycle	Task allocation or termination	Window length	Lifecycle event
I/O throughput	Bound exceeded for C consecutive samples	Threshold C	$[0, \infty]$
Send/receive timeout on a socket	Timeout exceeded (since a packet is sent over a socket)	Timeout value	$[0, \infty]$

Detection features for location

Monitored data have been translated into a vector of real numbers (*features*) in order to be exploited by the location classifier. OS features are provided to the location classifier in order to provide it with insights into the alarms triggered by the detection. This is to allow the discrimination between false positives (i.e., *SE*) and actual faults. Features can be both binary (e.g., they represent the occurrence of an event, like an error of a system call) and real values (e.g., statistics about timeouts within the system, like tasks schedulation times within a DU). Selected features are summarized in table 2.

Table 2: Features gathered by OS monitors.

Monitor	***Number of features***	***Description***
UNIX system calls	1141	For each pair (syscall, error code), there is a binary feature (it is 1 if the pair occurred, 0 otherwise)
UNIX signals	32	For each signal, there is a binary feature (it is 1 if the signal occurred, 0 otherwise)
Task schedulation timeout	4	The mean, the standard deviation, the minimum, and the maximum waiting time for schedulation of DU's tasks
Waiting time on	4	The mean, the standard deviation, the minimum,

semaphores		and the maximum waiting time for a semaphore of DU's tasks
Holding time in critical sections	4	The mean, the standard deviation, the minimum, and the maximum holding time for a semaphore of DU's tasks
Task lifecycle	2	Binary features representing the occurrence of tasks newly allocated or deallocated, respectively
I/O throughput	1	Binary feature (it is 1 if the throughput exceeded a bound, 0 otherwise)
Send/receive timeout on a socket	2*4*number of nodes	For each node in the system, the mean, the standard deviation, the minimum, and the maximum time since last packet sent over sockets to that node, both in input and in output

LOCATION AND RECOVERY

The machine learning paradigm underlies the location phase in the form of classification. This approach has been used in several works trying to solve different problems, e.g., works focusing on document classification (Manevitz & Yousef, 2002) (Jagadeesh, Bose, & Srinivasan, 2005) or aiming to find latent errors in software programs (Brun & Ernst, 2004). The location classifier has been trained in a supervised way, by the means of the pseudo-algorithm in figure 6.

```
FOR each node i
        FOR all the processes j running on i
            FOR each fault location k into the code of (i,j)
                    failure= do_injection(fault,k,i,j)
    //do the injection , wait for a while then analyze failed processes
                IF (failure ==UNKNOWN)    {
                       ADD fault to the set of KNOWN faults
                       ASSOCIATE recovery mean to the injected fault
                ENDIF
                COLLECT data from detector related to the last D seconds
                    CREATE the entry (detector_ouput, failure)
                ADD the entry to the training set
            ENDFOR
    ENDFOR
 ENDFOR

FOR each collected entry
    do_supervised_training(classifier)
ENDFOR
```

Figure 6: Supervised training of the location classifier.

The basic idea is to associate recovery actions to each experienced fault, thus keeping the classifier aware of the most suitable recovery action to start in the case of actual faults. In fact,

for each DU in the system, injected faults are added to the training set, if unknown. This way, a base knowledge is built by exploiting human insights.

Support Vector Machines (SVM) have been used for performing classification. The high-performance algorithm they rely on has been commonly used across a wide range of text classification problems, and they demonstrate to perform effectively for handling large datasets (Joachims, 1998).

SVM classifiers have been mainly introduced to the aim of solving binary problems, where the class label can take only two different values, and which can be solved by discriminating the decision boundary between the two classes. However, real world problems often require to take more complex decisions, i.e., to discriminate among more than two classes, hence SVM have been extended for handling multi-class problems. Multi-Class SVMs (MCSVMs), can be achieved in two ways: (i) by combining several standard, one-class, SVM classifiers, and (ii) by formulating a single optimization criteria for the whole set of available data. The basic idea underlying the SVM classification is to find the maximum margin hyperplane which provides the maximum margin among the classes. For non-linearly separable problems, the original data are projected into a certain high dimensional Euclidean space by means of a kernel function (K). Classification results depend on the proper choice of the kernel function; the (gaussian) Radial Basis Function is a commonly used kernel function. Furthermore, in their original formulation, they do not provide any estimation of their classification confidence, hence they do not allow to leverage any a-priori information which, if available, can be crucial to integrate into the classification process to yield reliable results. A probabilistic SVM variant has been developed to face this issue, even for multi-class problems (MPSVM). It is able to provide a probability value indicating at what extent a given sample belongs to a class; hence, its output is in fact a vector of probabilities whose length equals the number of classes. A formal and thorough discussion about SVM mathematical basis is beyond the scope of this paper. The interested reader can exploit a substantial literature (Joachims, 1998) or (Vapnik, 1995) can be pursued for an in-depth description.

In this work we use MPSVM variant and we leverage output probabilities to properly diagnose Unknown Faults (see section III), as well as to unmask detection of false positives (i.e., SUSPECTED ERRORS in section III). More precisely, we introduce a notion of confidence, C, which is in fact the maximum element of the output probability vector provided by the classifier. A fault is claimed unknown if C is less than a given threshold, t. As for suspected errors, a special class of faults, “No Fault”, has been introduced: if the classifier is confident that a given fault belongs to this class, (i.e., $C \geq t$), a suspected error is stated. The monitor which triggered the alarm is retrained in this case, by modifying the joint probability distributions in the Bayes’ rule. Additionally, no recovery action has to be initiated. Of course, the choice of t impacts on the diagnosis quality, hence we performed a sensitivity analysis, as it will be detailed in the next section. Figure 7 shows diagnosis response with respect to the location output.

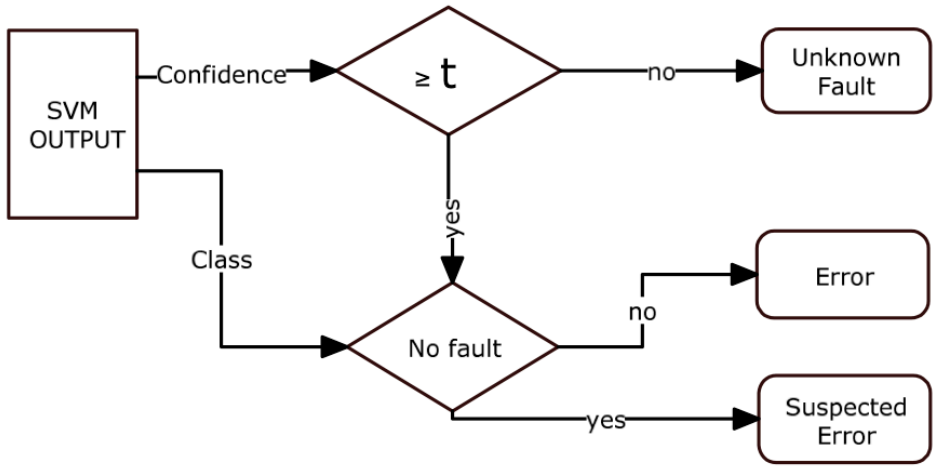

Figure 7: Diagnosis response with respect to location output.

EXPERIMENTAL FRAMEWORK AND RESULTS

Case study introduction

We evaluate the proposed *DLR* approach on a real case study from the Air Traffic Control (ATC) domain, within the framework of an industrial partnership with Finmeccanica group (COSMIC project, **http://www.cosmiclab.it**). The case study consists of a complex distributed application for Flight Data Processing. It is in charge of processing aircrafts data produced by Radar Track Generators, by updating the contents of Flight Data Plans (FDPs), and distributing them to flight controllers. The overall (simplified) architecture is depicted in figure 8; it is based on CARDAMOM (**http://cardamom.objectweb.org** (Corsaro, 2005)), i.e., an open-source CORBA middleware, for mission and safety critical applications which is compliant with OMG FT CORBA specifications (OMG, 2001). Furthermore, it makes use of OTS software items, such as the Data Distribution Service (DDS) implementation provided by RTI (**http://www.rti.com**) for publish-subscribe communication among components, and the ACE orb (**http://www.theaceorb.com**) on which CARDAMOM relies. The architecture is made up of several components:

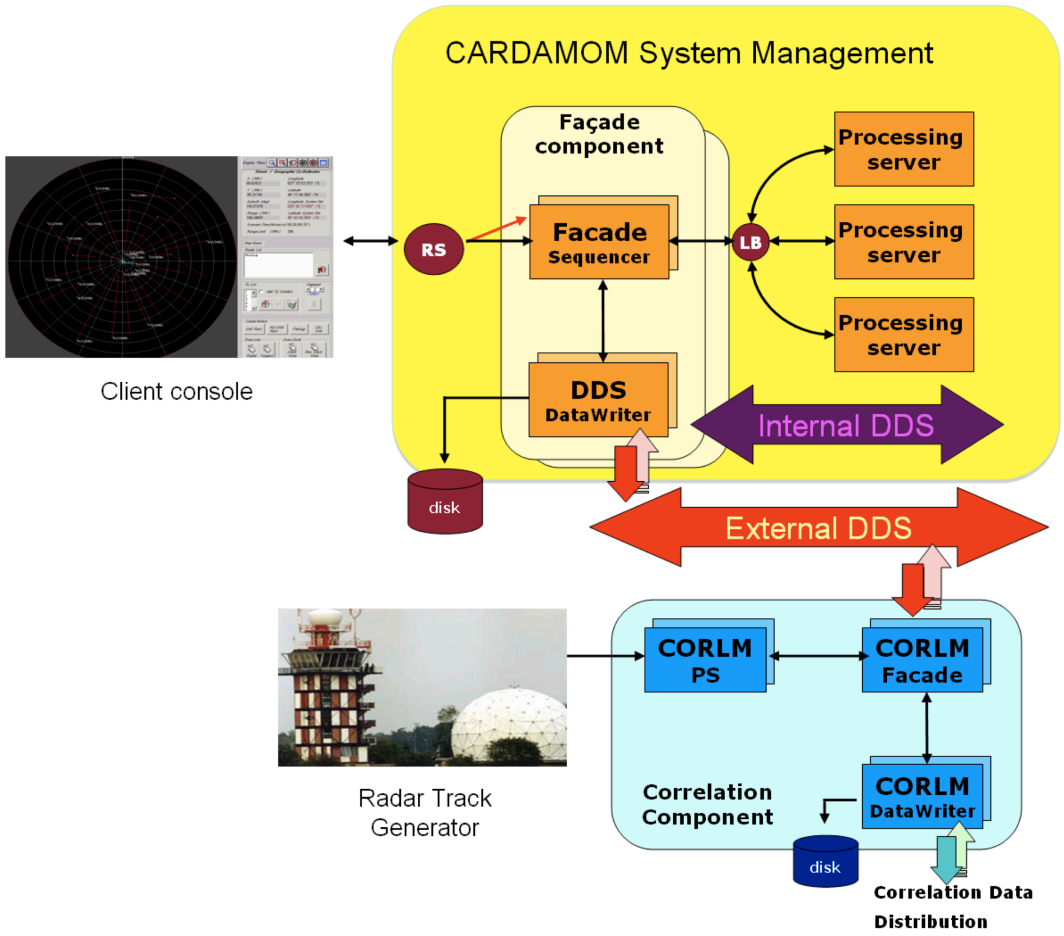

Figure 8: Case study architecture.

- *Facade component*, i.e., the interface between the clients (e.g., the flight controller console) and the rest of the system (conforming to the Facade GoF design pattern); it provides a remote object API for the atomic addition, removal, and update of FDPs. The Facade is replicated according to the warm-passive replication schema. It stores the FDPs along with a lock table for FDPs access serialization.
- *Processing Server* : it is in charge of processing FDPs on demand, by taking into account information from the Correlation Component and the FDPs published on the internal DDS. This component is replicated three times on different nodes, and FDP operations are balanced among servers with a round-robin policy.
- *Correlation component*, which collects flight tracks generated by radars, and associates them to FDPs, by means of Correlation Managers (CORLM in the figure 8).

The workload generates requests to the Facade component, both for flight tracks and FDP updates, in a random way and at a predefined average rate.

Objectives

Conducted experiments aim to demonstrate that:

- The detection approach is able to exploit several low-overhead and inaccurate monitors, by keeping low the false positive rate and the detection latency as well.
- The proposed location and recovery modules are able to correctly locate the root cause of a known fault within the system, and to trigger the proper recovery action in an *on-line* manner.
- The implemented DLR framework is able to partially discover unknown faults within the system. This is useful to trigger off-line maintenance (e.g., by alerting a human operator).

Evaluation metrics

According to (Cardenas, Baras, & Seamon, 2006), the following quality metrics have been used to evaluate detection approaches:

- *Coverage*: the conditional probability that, if there is a fault, it will detected. It is estimated by the ratio between the number of detected faults and the number of injected faults.
- *False positive rate*: the conditional probability that, if there is not a fault, an alarm will be issued. It is estimated by the ratio between the number of false alarms and the number of normal events monitored.
- *Latency*: the time between the execution of the fault-injected code, and the time of detection; it is an upper bound of the time between fault activation and the time of detection.
- *Overhead*: we consider the average execution time of remote methods implemented in the Facade remote object; in particular, we focus on the less and the most costly methods, in terms of execution time (respectively, *update_callback*, and *request_return*).

According to (Sebastiani, 2002) (Kim, Whitehead, & Zhang, 2008), the following metrics have been used to evaluate the location engine:

- *Accuracy*: the percentage of faults which are classified correctly, with respect to all activated faults. Letting A and B be two classes, it can be expressed as:
  $$A = \frac{TP_A + TP_B}{TP_A + FP_A + TP_B + FP_B}$$
- *Precision*: this metric is referred to individual classes; it represents the conditional probability that, if a fault is classified as belonging to class A, the decision is correct. It can be expressed as:
  $$P = \frac{TP_A}{TP_A + FP_A}$$
- *Recall*: this metric is referred to individual classes; it represents the conditional probability that, if a fault belongs to class A, the classifier decides for A.
  $$R = \frac{TP_A}{TP_A + FN_A}$$

In the previous equations, the quantities TP_A, FP_A and FN_A represent, respectively, the number of True Positives (i.e., the samples of A are classified as A), False Positives (i.e., the samples not of A are classified as A), and False Negatives (i.e., the samples of A are not classified as A).

Faultload

In order to evaluate the proposed fault detection and location techniques, we designed a realistic faultload based on the field data study conducted by (Duraes & Madeira, 2006). Faults have been injected in the source code of application-level components (i.e., the Facade and the processing servers), using the most common fault operators. Injected faults are detailed in table 3.

Table 3: Source-code faults injected in the case study application.

ODC DEFECT TYPE	***FAULT NATURE***	***FAULT TYPE***	#
Assignment (63.89%)	MISSING	MVIV - Missing Variable Initialization using a Value	8
		MVAV - Missing Variable Assignment using a Value	5
		MVAE - Missing Variable Assignment using an Expression	5
	WRONG	MVAV - Wrong Value Assigned to Variable	26
	EXTRANEOUS	EVAV - Extraneous Variable Assignment using another Variable	2
Checking (6.94%)	MISSING	MIA - Missing IF construct Around statement	2
	WRONG	WLEC - Wrong logical expression used as branch condition	3
Interface (4.17%)	MISSING	MLPA - Missing small and Localized Part of the Algorithm	2
	WRONG	WPFV - Wrong variable used in Parameter of Function Call	1
Algorithm (20.83%)	MISSING	MFC - Missing Function Call	13
		MIEB - Missing IF construct plus statement plus ELSE Before statement	1
		MIFS - Missing IF construct plus statement	1
Function (4.17%)	MISSING	MFCT - Missing Functionality	2
	WRONG	WALL - Wrong Algorithm (Large modifications)	1
Total			72

Fault types listed in the table are representative of the most common mistakes made by programmers. In particular, according to the Orthogonal Defect Classification, faults can be characterized by the change in the code that is necessary to correct it. Therefore, in order to emulate a software fault, we have to choose an adequate source code location, which is similar to ones containing faults from the field. Faults operators in (Duraes & Madeira, 2006) describe the rules to locate representative fault locations within source code. The operators were applied to components according to software complexity metrics, in order to choice the source locations containing the most of residual faults (Moraes, Duraes, Barbosa, Martins, & Madeira, 2007). The

most complex target components, in term of Lines Of Code (LOCs) and cyclomatic complexity, turned out to be the C++ classes implementing the Facade and Processing Server remote objects; we have injected, respectively, 56 and 16 source-code faults in them.

Before each fault injection campaign, source-code faults are randomly divided in two distinct sets, namely *training set* and *test set*. These are characterized by the same size, and the same number of source-code faults. Training sets are used to setup the detection and location techniques, and test sets are used to evaluate their effectiveness. Each fault injection experiment encompasses only one source-code fault at a time.

Adopted testbed

We used a cluster machine made up of 128 nodes. The system deployment consists of 9 machines (two Facade replicas, one for the CARDAMOM FT service, one for Load Balancing Service, three for the FDP processing servers, and 2 nodes are allocated to the Client and to CORLM component, respectively) wired by Gigabit LAN. In order to have more reliable results, and not be biased by hardware errors, we partitioned the cluster in 10 LANs. Thus, each experiment was lunched on the 10 partions, simultaneously. Results are then filtered and averaged. The hardware configuration of testing machines is made up of 2 Xeon Hyper-Threaded 2.8GHz CPUs, 3.6GB of physical memory, and a Gigabit Ethernet interface; machines are interconnected by a 56 Gbps switch, and they are equipped with the Linux OS with kernel v.2.6.25.

DLR in the case study

The DLR approach was applied to the considered case study, by defining the features and the classes used for fault diagnosis. In particular, the binaries and the libraries of both the application and OTS libraries (e.g., CARDAMOM, TAO) were inspected to extract potential error messages produced by them (using the *strings* UNIX utility). Several error messages were collected, and a dictionary of words was build on them. The total amount of features from all the monitored DUs and logs was 17171 (see table 4).

Table 4: Features used for diagnosis in the case study application.

Number of log file types	8
Number of monitored log files	16
Number of OTS libraries	87
Number of potential log messages	7691
Number of unique tokens within log messages	6043
Number of application keywords	33
Monitored processes by the OS	Façade, 3 Servers
Number of OS features (per process)	1250
Total amount of features	17171

The location classifier was trained using fault injection, and fault classes were identified using the proposed approach (see table 5). For each class, the root cause is represented by the component in which the fault was injected during the training phase. A proper recovery mean has been associated to each fault class.

Table 5: Diagnosis fault classes.

	FAULT TYPE	***FAULT LOCATION***	***RECOVERY***
Class 0	No fault	None	The system is correctly working.
Class 1	Crash	Façade	Activate the backup replica; a new backup replica is activated.
Class 2	Passive hang	Façade	Free all resources locked by semaphores, and kill the preempted transaction. The correctness of this recovery is due to the specific application properties (e.g., the FDP will be correctly updated by the next update operation); another recovery mean would be to kill the hung Façade and treat it as a crashed Façade.
Class 3	Crash	Server	Reboot the server process; add it to the load balanced group.
Class 4	Passive hang (at start time)	Façade	Reboot the whole application. The application start may fail because of transient faults, then the reboot may succeed on the second try. If the application still does not start, human intervention is requested.

Measurements

Detection:

As a basis for comparison, we first evaluated the performance of individual monitors. For each monitor, a sensitivity analysis has been made, letting parameter's value of each monitor vary within the range [1s, 4s] (see table 1). The best values for all detectors, with respect to the Facade and Server DUs respectively, are shown in tables 6 and 7. Different monitors achieve different performances in terms of coverage, since they are suited for different failure modes; actually, monitors are unable to achieve full coverage, except for the *SOCKET* monitor. Furthermore, performances vary with respect to the considered DU. As for example, in the case of the processing server, only crashes (i.e., class 3 in table 5) have been observed, hence no faults

have been identified by monitors controlling blocking conditions (e.g., wait for a semaphore). The reason for which all the monitors experience the same mean latency value, is that they have been triggered all together after the abortion of the processing server DU.

Table 6: Coverage, false positive rate, and latency provided by the individual detectors for the Façade DU.

Detector	***Parameter***	***Coverage***	***False positive rate***	***Mean Latency (ms)***
UNIX semaphores hold timeout	4 s	64.5%	36.08%	1965.65
UNIX semaphores wait timeout	2 s	67.7%	1.7%	521.18
Pthread mutexes hold timeout	4 s	64.5%	4.01%	469.51
Pthread mutexes wait timeout	-	0%	0%	-
Schedulation threshold	4 s	74.1%	3.25%	1912.22
Syscall error codes	1 s	45.1%	0.6%	768.97
Process exit	1 s	45.1%	0%	830.64
Signals	1 s	45.1%	0%	816.57
Task lifecycle	1 s	35.4%	0.05%	375.7
I/O throughput network input	3 s	77.3%	0.4%	4476.67
I/O throughput network output	3 s	77.3%	0.2%	2986.4
I/O throughput disk reads	3 s	70.9%	0.4%	4930
I/O throughput disk writes	2 s	67.6%	0.05%	6168.57
Sockets	4 s	100%	3.47%	469.58

Detector	***Parameter***	***Coverage***	***False positive rate***	***Mean Latency (ms)***
UNIX semaphores hold timeout	2 s	0%	3.61%	-
UNIX semaphores wait timeout	2 s	0%	2.28%	-
Pthread mutexes hold timeout	2 s	0%	4.44%	-
Pthread mutexes wait timeout	-	0%	0%	-
Schedulation threshold	1 s	0%	3.25%	-

Syscall error codes	1 s	100%	0.98%	522.5
Process exit	1 s	100%	0.005%	522.5
Signals	1 s	100%	0.005%	522.5
Task lifecycle	1 s	100%	0.22%	522.5
I/O throughput network input	3 s	100%	0.49%	522.5
I/O throughput network output	3 s	100%	87.35%	522.5
I/O throughput disk reads	3 s	100%	79.31%	522.5
I/O throughput disk writes	3 s	100%	77.77%	522.5
Sockets	2 s	100%	3.14%	522.5

Several monitors provided a small number of false positives, even if there were monitors which provided an unacceptably high false alarm rate. For this reason, it is important to filter false positives in order to include those monitors within the system (this is useful to increase the amount of covered faults, and to deliver more information to the location phase).

Table 8 shows the performances achieved by the joint detection algorithm. It can be seen that the joint detector is able to achieve the full coverage of all injected faults, while keeping low the false positive rate (it is comparable to the best rates in tables 6 and 7). Another benefit given by the joint detection is the much lower latency: in fact, when one of the individual monitors produce an anomalous value, the other detectors are immediately inspected for anomalies, providing a lower mean detection time.

Table 8: Coverage, accuracy, and latency provided by the joint detection approach.

	Façade	*Server*
Coverage	100%	100%
False positive rate	4.85%	6.86%
Mean Latency	100.26±135.76 ms	165.67±122.43 ms

Finally, the overhead of continuous monitoring DUs at the O.S. level has been measured, by varying the request rate from the client; figures 9 and 10 show the execution time observed with and without monitors. It should be noted that the overhead was lower that 10% in every case, even during most intensive workload periods.

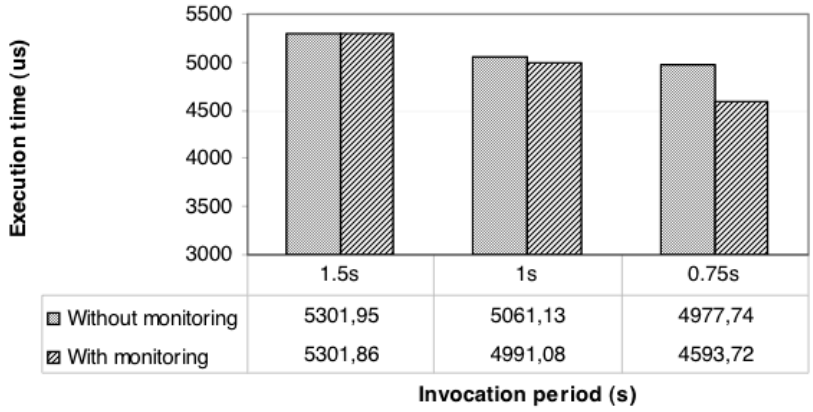

Figure 9: Overhead imposed to the execution of Façade's "update_callback" method.

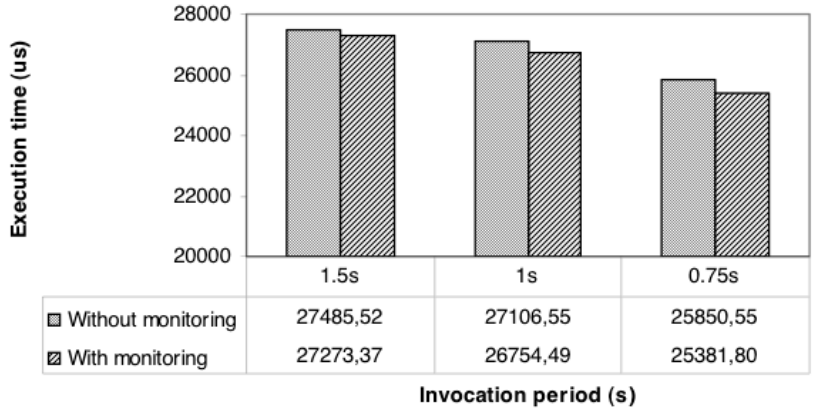

Figure 10: Overhead imposed to the execution of Façade's "request_return" method.

Location:

We evaluated the performance of the location phase with respect to both known faults (i.e., faults similar to ones observed during the training phase) and unknown faults (i.e., faults completely different than all known ones). First, we excluded faults belonging to class 4 from the training set, and evaluated location capability with respect to the remaining (known) classes, using a low confidence level ($C = 0.9$); in all the cases, the location was able to identify the correct fault class. Moreover, the location was able to identify all false positives produced by detection during faulty-free execution.

Next, faults belonging to class 4 were used for testing location; results are shown in table 9. It is shown that, although all known faults are correctly classified for $C = 0.9$, only a small amount of unknown faults were identified (represented by the Recall measure for unknown faults); in the most of cases, the locator wrongly classified an unknown fault as a known one. Therefore, we made a sensitivity analysis on the confidence level C, in order to discover the confidence level needed for the correct identification of unknown faults. It should be noted that an increase in the required confidence level for location, reduces the amount of known faults correctly identified; therefore, human intervention could be required even for known (but not trustfully classified)

faults. Nevertheless, it can be noted that, by increasing the confidence level, a better trade-off between identification of known and unknown faults can be achieved: a confidence level $C = 0.99$ or $C = 0.995$ still provides fully correct known fault classification, with a higher amount of unknown faults identified.

Table 9: Classification diagnosis evaluation, when deliberately excluding class 4 from the training (UNKNOWN). When a fault was classified as KNOWN, in all cases it was also correctly classified with respect to table 5.

Confidence	***ACCURACY***	***P(KNOWN)***	***R(KNOWN)***	***P(UNKNOWN)***	***R(UNKNOWN)***
0.9	60%	59.09%	100%	100%	5.26%
0.99	75.56%	70.27%	100%	100%	42.11%
0.995	77.78%	73.52%	96.15%	90.91%	52.63%
0.999	75.56%	80%	76.92%	70%	73.68%
0.9999	42.22%	n.a.	0%	42.22%	100%

After that, we included in the training set half the samples of class 4, becoming a known fault. Results with the respect to the confidence level are shown in table 10; known fault classification is still very high for more demanding confidence levels (i.e., $C = 0.9$, $C = 0.99$).

Table 10: Classification diagnosis evaluation, when including all 5 classes in the training. When a fault was classified as KNOWN, in all cases it was also correctly classified with respect to table 5.

Confidence	***ACCURACY***
0.9	100%
0.99	94.29%
0.995	94.29%
0.999	71.43%
0.9999	25.71%

In table 11, the mean time for detection data collection and classification are shown. It can been seen than the total amount of time required to diagnose a fault (the sum of mean detection, collection, and location times on the average) is about 1.2 seconds, which is reasonable for a large class of critical COTS-based systems.

Table 11: Time measurements for the location phase.

Mean time for data collection	84.4 ± 115.11 ms
Mean time for location	917.14 ± 23.63 ms

Finally, figure 11 shows the (cumulative) amount of false positives produced by joint detection during a long period of time. The location has been configured to retrain the detector which erroneously triggered the location, by updating the joint probability distribution $P(\underline{a} \mid \neg F)$. This produced a dramatic decrease of the false positives rate after less than an hour of execution, by filtering most common false positive patterns occurring during the detection phase.

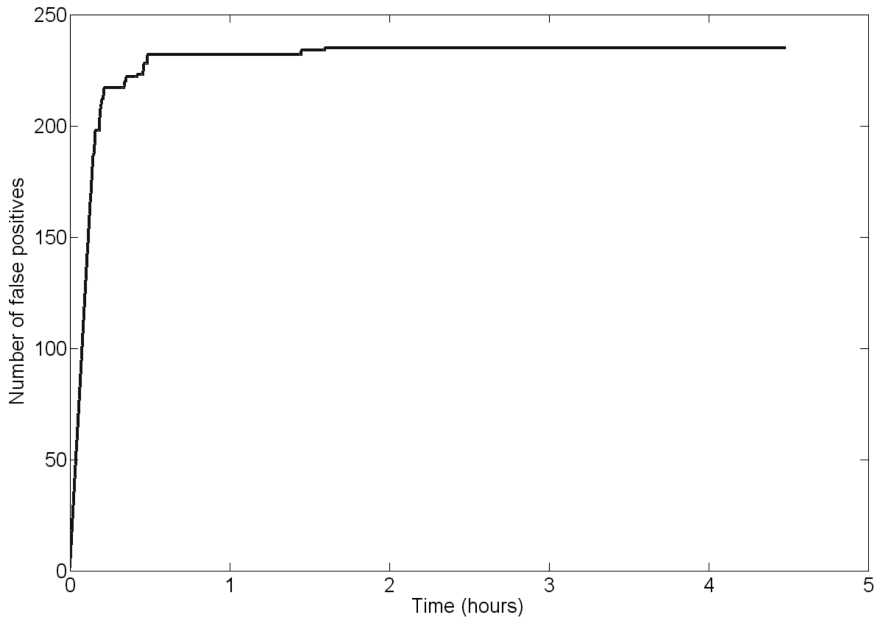

Figure 11: Cumulative number of false positives during time, using the location output to retrain detection joint probabilities.

RELATED WORK

The issue of diagnosis is being faced since a long time, maybe since computers came. The first attempt to formalize the problem is due to (Preparata, Metze, & Chien, 1967) which introduced system level diagnosis. The model they proposed in 1967 (also known as the PMC model) assumed the system to be made up of several units which test one another, and test results are leveraged to diagnose faulty units. Several extensions to this model have been proposed, even recently (e.g., (Vaidya & Pradham, 1994) where the safe system level diagnosis has been proposed).

In the last decade or so, there has being an increasing work focusing on diagnosis in order to face the problem by several perspectives and by using quite different techniques. For this reason, we tried to leverage existing solutions to similar problems, as well as to use approaches close to the ours which have been rather used to face different issues.

Similar approaches to similar problems (SASP)

The goal of identifying automatically the root cause of a failure is pursued in (Yuan, et al., 2006). Authors propose a trace-based problem diagnosis methodology, which relies on the trace of low level system behaviors to deduce problems of computer systems. Transient events occurring in the system (e.g., system calls, I/O requests, call stacks, context switches) are traced in order to (i) identify the correlations between system behaviors and known problems and (ii), use the learned knowledge to solve new (i.e., not known) problems. These goals are achieved by means of statistical learning techniques, based on SVMs, similarly to our work. The ultimate aim

that authors want to pursue is to make the problem identification fully automatic, thus eliminating human involvement. We have a different goal, in that we also aim to trigger recovery actions. Furthermore, the symptom of the problem needs to be reproduced before the root cause detection.

A decision tree based approach is presented in (Zheng, Lloyd, & Brewer, 2004) to diagnose problems in Large Internet Services. Similarly to what we do in this work, runtime properties of the system (they record clients requests) are monitored; automated machine learning and data mining techniques are used to identify the causes of failures. The proposed approach is evaluated by measuring precision and recall, similarly to what we do for evaluating diagnosis quality. However, our work differs from this one for what concerns with detection. In fact, detection is not encompassed in (Zheng, Lloyd, & Brewer, 2004): authors assume problems to have been already detected and they only concentrate on identifying the root cause, in order to trigger a fast recovery.

Similar approaches to different problems (SADP)

(Podgurski, et al., 2003) proposes an automated support for classifying reported software failures in order to facilitate the diagnosing of their root causes. The authors use a classification strategy which makes us of supervised and unsupervised pattern classification, as we do for location and detection respectively. Additionally, they also concentrate on the importance of features selection and extraction, as we do. However, the classification performed in this work aims to group failures which are due to the same cause and it is conceived as a mean for helping actual diagnosis. Conversely, we actually perform diagnosis by means of classification.

A very recent work which uses a machine learning approach based on SVM classification is (Kim, Whitehead, & Zhang, 2008). Its main goal is to predict the presence of latent software bugs in software changes (change classification). In particular, a machine learning SVM classifier is used to determine whether a new software change is more similar to prior buggy changes or clean changes. In this manner, change classification predicts the existence of bugs in software changes. We have in common with this work the classification problem, its formulation and the process of feature extraction.

Machine learning approach has also been used in (Brun & Ernst, 2004) for identifying program properties that indicate errors. The technique generates machine learning models of program properties known to result from errors, and applies these models to program properties of user-written code to classify the properties that may lead the user to errors. SVMs and decision trees are used for classification. The effectiveness of the proposed approach has been demonstrated with respect to C, C++, and Java programs. However it requires human labor to find the bugs, and the process is not fully automatic.

(Aguilera, Mogul, Wiener, Reynolds, & Muthitacharoen, 2003) address the problem of locating performance bottlenecks in a distributed system with only internode communication traces. They infer the causal paths through multi-tier distributed applications from message level traces, in order to detect the node causing extraordinary delay. They share with us the great attention which is paid to the presence of OTS items, as well as the fact that the

approach requires no modifications to applications and middleware. The major differences with our work are (i) the fact that they pay more attention to performance rather than on faults and (ii), the fact that they perform off-line diagnosis of the problem.

As for Bayesian estimation, a worth noting work to be referred is (Chang, Lander, Lu, & Wells, 1993) which addresses system diagnosis problems. It refers to comparison-based system analysis to deal with incomplete test coverage, unknown numbers of faulty units, and non-permanent faults. However, only one type of monitor is used in that work and also recovery is not encompassed.

Different approaches to similar problems (DASP)

Closely related to our work in goals is (Joshi, Hiltunen, Sanders, & Schlichting, 2005), which cares about automatic model driven recovery in distributed systems. Similarly to what we do, authors exploit a set of a limited coverage monitors whose output are combined in a certain way prior to trigger recovery actions. Additionally they also have a Bayesian Faults Diagnosis engine in charge of locating the problem, as well as to pilot a recovery controller that can choose recovery actions based on several optimization criteria. Similarly to the approach we propose, the approach proposed in (Joshi, Hiltunen, Sanders, & Schlichting, 2005) is able to detect whether a problem is beyond its diagnosis and recovery capabilities, and thus to determine when a human operator needs to be alerted. Despite of these common purposes, we take an opposite perspective in that we do not follow a model based approach since modeling the complex software systems we are addressing could be too difficult and inaccurate. Additionally, our work is different in several points. First, they propose incremental recovery actions whereas we directly start the best one action able to repair the system. Second, we always use the entire set of ``always-on'' monitors to detect errors instead of invoking additional monitors when needed. Third, we use fault injection to experimentally prove the effectiveness of the approach rather than for making a comparison with a theoretical optimum.

(Khanna, Laguna, Arshad, & Bagchi, 2007) face the problem of diagnosis in networked environments made up of black-box entities. This goal is achieved by (i) tracing messages to build a causal dependency structure between the components (ii), by tracing back the causal structure when a failure is detected and (iii), by testing components using diagnostic tests. Runtime observations are used to estimate the parameters that bear on the possibility of error propagation, such as unreliability of links and error masking capabilities. The work aims to provide diagnosis of the faulty entities at runtime in a non-intrusive manner to the application. Differently from this work, we do not build causal structure of the system since we do not make any assumption on the structure of the system itself. The main point in common is the fact that we pursue on-line diagnosis as well.

(Brown, Kar, & Keller, 2001) defines a methodology for identifying and characterizing dynamic dependencies between system components in distributed application environments, which relies on active perturbation of the system. This is in order to identify dependencies, as well as to compute dependency strengths. Even if discovering system dependencies automatically could be a good way for root cause analysis, it is assumed a deep knowledge of system internals. In particular, authors assume to completely know end-users interaction with the system (they use a

well known TPC benchmark). We take the opposite position in that we do not require such a knowledge. Furthermore, the Active Dependency Discovery approach which is defined in that work, reveals to be strongly intrusive and workload dependent.

A further worth referring work is (Chen, Kiciman, Fratkin, Fox, & Brewer, 2002), where the Pinpoint framework is defined. It employs statistical learning techniques to diagnose failures in a Web farm environment. After the traces with respect to different client requests are collected, data mining algorithms are used to identify the components most relevant to a failure. We share with that work the "learning from system behavior" philosophy. However, there is a difference in goals, since we want to detect and diagnose faults in order to determine the cause of the failure and trigger recovery action. Conversely, Pinpoint aims to recognize which component in a distributed system is more likely to be faulty. Fault injection is used also in (Chen, Kiciman, Fratkin, Fox, & Brewer, 2002) to prove the effectiveness of the approach. The major limitation of this approach are that (i) it is suitable only for small scale software programs, and (ii) it exhibit a significant logging. We differ from that work in two main points: (i) the Pinpoint framework is designed to work off-line and (ii), it is not a recovery-oriented approach.

Finally, on-site failure diagnosis is faced in (Tucek, Lu, Huang, Xanthos, & Zhou, 2007). The work aims to capture the failure point and conduct just-in-time failure diagnosis with checkpoint-reexecution system support. Lightweight checkpoints are taken during execution and rolls back are performed to recent checkpoints for diagnosis after a failure has occurred. Delta generation and delta analysis are exploited to speculatively modify the inputs and execution environment to create many similar but successful and failing replays to identify failure-triggering conditions. We discard a similar approach since Heisenbugs can be unreproducible this way: in fact, their conditions of activation are hard to identify (Grottke & Trivedi, 2007). Furthermore, long time is required (almost five minutes) to complete the process: this can be not tolerable for safety critical systems. Table 12 summarizes the related work.

CONCLUSION

In the context of leveraging the dependability of complex and fault tolerant software systems, the paper advocated the need of recovery oriented software fault diagnosis approach, which integrated detection, location, and recovery in one holistic diagnostic framework. This is different from most of the previous work which has been conducted on software failure diagnosis in the last few years in that target systems, in which we also conducted experiments, are very complex. Thus existing approaches, which require human involvement to discover the bug, are not suitable for field failures for a number of reasons. First of all, it is difficult to reproduce the failure-triggering conditions in house for diagnosis. Second, off-line failure diagnosis cannot provide timely guidance to select the best recovery action, i.e., a recovery action which is tailored for the particular fault that occurred.

The experimental campaign has been conducted in the context of a real-world Air Traffic Control system. Results demonstrated that:

- The detection approach is able to exploit several low-overhead and inaccurate monitors at the OS level, by keeping low the false positive rate and the detection latency as well;

- The proposed location and recovery strategies are able to correctly locate the root cause of a known fault within the system, and to trigger the proper recovery action in an on-line manner.
- The implemented DLR framework is able to partially discover unknown faults within the system. This is useful to trigger off-line maintenance (e.g., by alerting a human operator).

REFERENCES

Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., & Muthitacharoen, A. (2003). Performance Debugging for Distributed Systems of Black Boxes. *19th ACM Symposium on Operating Systems Principles*, (pp. 74–89).

Becraft, W., & Lee, P. (1993). An Integrated Neural Network/Expert System Approach for Fault Diagnosis. *Computers and Chemical Engineering , 17* (10), 1001-1014.

Bondavalli, A., Chiaradonna, S., Cotroneo, D., & Romano, L. (2004). Effective Fault Treatment for Improving the Dependability of COTS and Legacy-Based Applications. *IEEE Transactions on Dependable and Secure Computing , 1* (4), 223-237.

Brown, A., Kar, G., & Keller, A. (2001). An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. *IEEE/IFIP Symposium on Integrated Network Management*, (pp. 377–390).

Brun, Y., & Ernst, M. D. (2004). Finding Latent Code Errors via Machine Learning over Program Executions. *26th Conference on Software Engineering*, (pp. 480–490).

Cardenas, A. A., Baras, J. S., & Seamon, K. (2006). A Framework for the Evaluation of Intrusion Detection Systems. *IEEE Symposium on Security and Privacy*, (pp. 63–77).

Chang, Y., Lander, L. C., Lu, H. S., & Wells, M. T. (1993). Bayesian Analysis for Fault Location in Homogenous Distributed Systems. *12th Symposium on Reliable Distributed Systems*, (pp. 44–53).

Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., & Brewer, E. (2002). Pinpoint: Problem Determination in Large, Dynamic Internet Services. *IEEE/IFIP Conference on Dependable Systems and Networks*, (pp. 595–604).

Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., et al. (1992). Orthogonal Defect Classification-A Concept for In-Process Measurements. *IEEE Transactions on Software Engineering , 18* (11), 943-956.

Chillarege, R., Biyani, S., & Rosenthal, J. (1995). Measurement of Failure Rate in Widely Distributed Software. *25th Symposium on Fault-Tolerant Computing*, (pp. 424-433).

Corsaro, A. (2005). CARDAMOM: A Next Generation Mission and Safety Critical Enterprise Middleware. *Workshop on Software Technologies for Future Embedded & Ubiquitous Systems*, (pp. 73–74).

Daidone, A., Di Giandomenico, F., Chiaradonna, S., & Bondavalli, A. (2006). Hidden Markov Models as a Support for Diagnosis: Formalization of the Problem and Synthesis of the Solution. *25th IEEE Symposium on Reliable Distributed Systems* (pp. 245-256). IEEE.

Duraes, J., & Madeira, H. (2006). Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering , 32* (11), 849-867.

Forrest, S., Hofmeyr, S. A., Somayaji, A., & Longstaff, T. A. (1996). A sense of self for Unix processes. *IEEE Symposium on Security and Privacy*, (pp. 120-128).

Gray, J. (1985). *Why Do Computer Stop and What Can Be Done About It?* Tandem TR 85.7.

Grottke, M., & Trivedi, K. S. (2007). Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *IEEE Computer , 40* (2), 107–109.

Huang, Y., Jalote, P., & Kintala, C. (1994). Two Techniques for Transient Software Error Recovery. *Workshop on Hardware and Software Architectures for Fault Tolerance: Experiences and Perspectives*, (pp. 159–170).

Jagadeesh, R. P., Bose, C., & Srinivasan, S. H. (2005). Data Mining Approaches to Software Fault Diagnosis. *15th IEEE Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications*, (pp. 45–52).

Joachims, T. (1998). Text Categorization with Support Vector Ma- chines: Learning with Many Relevant Features. *10th European Conference On Machine Learning*, (pp. 137–142).

Joshi, K. R., Hiltunen, M. A., Sanders, W. H., & Schlichting, R. D. (2005). Automatic Model-Driven Recovery in Distributed Systems. *24th IEEE Symposium on Reliable Distributed Systems*, (pp. 25-36).

Khanna, G., Laguna, I., Arshad, F. A., & Bagchi, S. (2007). Distributed Diagnosis of Failures in a Three Tier E-Commerce System. *26th IEEE Symposium on Reliable Distributed Systems*, (pp. 185-198).

Kim, S., Whitethead, E. J., & Zhang, Y. (2008). Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering , 34* (2), 181-196.

Littlewood, B., & Strigini, L. (2000). Software Reliability and Dependability: A Roadmap. *ACM Conference on The Future of Software Engineering*, (pp. 175–188).

Manevitz, L. M., & Yousef, M. (2002). One-Class SVMs for Document Classification. *Journal of Machine Learning Research , 2*, 139-154.

Moraes, R., Duraes, J., Barbosa, R., Martins, E., & Madeira, H. (2007). Experimental Risk Assessment and Comparison Using Software Fault Injection. *37th IEEE/IFIP Conference on Dependable Systems and Networks*, (pp. 512-521).

OMG. (2001). *Fault Tolerant CORBA Standard, v2.5.*

Oppenheimer, D. L., & Patterson, D. A. (2002). Studying and Using Failure Data from Large-Scale Internet Services. (pp. 255–258). 10th ACM SIGOPS European Workshop.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., et al. (2003). Automated Support for Classifying Software Failure Reports. *25th Conference on Software Engineering*, (pp. 465-475).

Preparata, F. P., Metze, G., & Chien, R. T. (1967). On the Connection Assignment Problem of Diagnosable Systems. *IEEE Transactions on Electronic Computers , 16* (6), 848–854.

Sebastiani, F. (2002). Machine Learning in Automated Text Categorization. *ACM Computing Surveys , 34*, 1-47.

Serafini, M., Bondavalli, A., & Suri, N. (2007). On-Line Diagnosis and Recovery: On the Choice and Impact of Tuning Parameters. *IEEE Transactions on Dependable and Secure Computing , 4* (4), 295-312.

Sullivan, M., & Chillarege, R. (1991). Software Defects and Their Impact on System Availability - A Study of Field Failures in Operating Systems. *21st Symposium on Fault-Tolerant Computing*, (pp. 2-9).

Tucek, J., Lu, S., Huang, C., Xanthos, S., & Zhou, Y. (2007). Triage: Diagnosing Production Run Failures at the User's Site. *21st ACM SIGOPS Symposium on Operating Systems Principles*, (pp. 131–144).

Vaidya, N. H., & Pradham, D. K. (1994). Safe System Level Diagnosis. *IEEE Transactions on Computers , 43* (3), 367–370.

Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory.* Springer-Verlag.

Xu, J., Kalbarczyc, Z., & Iyer, R. K. (1999). Networked Windows NT System Field Data Analysis. *Pacific Rim Symposium on Dependable Computing*, (pp. 178-185).

Yuan, C., Lao, N., Wen, J. R., Li, J., Zhang, Z., Wang, Y. M., et al. (2006). Automated Known Problem Diagnosis with Event Traces. *EuroSys ACM Conference*, (pp. 375–388).

Zheng, A. X., Lloyd, J., & Brewer, E. (2004). Failure Diagnosis Using Decision Trees. *1st IEEE Conference on Autonomic Computing*, (pp. 36-43).