Operating System Support to Detect Application Hangs

G. Carrozza[1], M.Cinque[1], D.Cotroneo[1], R. Natella[2]
(1) Dipartimento di Informatica e Sistemistica - Università degli Studi di Napoli Federico II
Via Claudio 21, 80125 - Naples, Italy
(2) Laboratorio CINI ITEM, Complesso Univ. M.S.Angelo, via Cinthia, Naples, Italy
ga.carrozza, macinque, cotroneo@unina.it, rnatella@consorzio-cini.it

Abstract

On-line failure detection is an essential means to control and assess the dependability of complex and critical software systems. In such context, effective detection strategies are required, in order to minimize the possibility of catastrophic consequences. This objective is however difficult to achieve in complex systems, especially due to the several sources of non-determinism (e.g., multi-threading and distributed interaction) which may lead to software hangs, i.e., the system is active but no longer capable of delivering its services. The paper proposes a detection approach to uncover application hangs. It exploits multiple indirect data gathered at the operating system level to monitor the system and to trigger alarms if the observed behavior deviates from the expected one. By means of fault injection experiments conducted on a research prototype, it is shown how the combination of several operating system monitors actually leads to an high quality of detection, at an acceptable overhead.

Keywords: Failure detection, critical systems

1. INTRODUCTION

The increasing complexity of mission and safety critical software systems asks for effective mechanisms to detect the occurrence of failures. Failure detection plays a key role to trigger prompt automatic system reconfiguration, recovery, and correction actions. In addition, it is prerequisite to implement on-line dependability evaluation, which is of particular use to system administrators/deployers to improve current and future versions of the system. Today these systems are being developed as the composition of several Off the Shelf (OTS) software modules and complex multi-threaded components. Hence, detecting failures efficiently becomes difficult, especially if we consider the particular criticality of these systems: no failures should remain undetected in order to avoid catastrophic consequences (i.e., ideally, the detector should exhibit a 100% *coverage* of observed failures). In addition, the number of false positives should be minimized, in order to avoid unecessary (and costly) recovery actions (i.e., the detector should be highly *accurate*). Finally, the overhead introduced by the detection process should be kept low, in order to not compromise the mission of the system.
Many detection techniques have been developed for failures due to hardware faults (e.g., single event upsets caused by particles striking the integrated circuit). Examples of these techniques are Algorithm Based Fault Tolerance [1], assertions [2], and Control Flow Checking [3]. As for failures due to software faults, e.g., programming errors, static code analysis and testing strategies are commonly used to find and to correct errors in the code. However, software bugs account for a limited part of software faults, which have been rather recognized to be mostly transient in nature. Studies on field data analysis show that the majority of software faults are due to overload, timing and exception errors and race conditions [4, 5]. Static analysis and testing techniques fail when dealing with this kind of faults, namely Heisenbugs [6], since their condition of activation cannot be reproduced systematically. This is especially true in the case of complex concurrent applications, where multi-threading and shared resources represent a source of nondeterminism in the application behavior.
For these reasons, failure detection has to take place during the operational phase of the system. This is usually achieved through direct detection techniques at the application level, such as polling the health status of system components (i.e., heartbeat mechanisms), or using system log files, where available, to analyze error messages and their possible correlation.
However, the problem is exacerbated in the case of OTS software modules, where the source code is often

unavailable. On the one hand, the implementation of simple heartbeat schemes generally requires extra code. On the other hand, log-based detection typically requires a significant computational load to be performed on-line. Moreover, both techniques may result inadequate at detecting software hangs, i.e., halt failures due to either infinite loops (active hangs) or indefinite wait conditions (passive hangs). First, when dealing with multi-threaded components, the hang might be localized on a different thread than the one that answers to the heartbeat. Hence, the component correctly answers to polls, while portions of it result permanently hanged. Second, it is hard to tell whether it does not respond to queries or it does not log any data because it is really hanged or it is just “very slow” (e.g., it might be overloaded or correctly blocked on a waiting condition). In any case, hang failures cannot remain undetected in mission and safety-critical systems, where reliability and predictability are the most important requirements.

To overcome these limitations, we investigated indirect detection techniques, which have been recently proposed in the literature. These techniques are based on the observation of low level system parameters related to the component of interest (e.g., invoked system calls, hardware performance counters), in order to build a model of the correct or expected behavior of the system. Similarly to anomaly detection, an alarm is triggered if the operational behavior deviates from the expected one. The deviation is usually assessed by using data mining or artificial intelligence techniques. Although these techniques are promising, they generally require a non-negligible overhead due to complex data analysis and constant system observation. In addition, as better detailed in section 2, in most of the cases it has not been shown whether they are able to detect software hangs. To the best of our knowledge, only one of the most relevant work in this field accounts for active application hangs [7], however no means is provided to detect passive hang conditions.

This work proposes an *indirect* failure detection framework for pinpointing active and passive halt failures (see section 3). This is achieved by leveraging Operating System (OS) support, conversely from above mentioned direct mechanisms which work at the application-level. In order to be of practical use in mission- and safety-critical systems, the detection framework has been designed with two main objectives in mind: i) to achieve high quality of detection, in terms of coverage and accuracy, and ii) to keep the overhead low. Pursuing objective i), we propose to combine multiple sources of indirect data gathered at the OS level. This actually increases the overall capacity of detecting failures, as confirmed by our experimental results. For instance, we monitor the waiting time on semaphores and the holding time into critical sections to detect possible passive hangs. As for objective ii), we design a lightweight scheme to manipulate gathered data and to trigger detection events. The triggering scheme is parametric (e.g., thresholds on waiting times on semaphores are set for each thread in a process); values for its parameters are experimentally “trained” on the actual system under observation, hence they characterize its normal behavior. A detection event is then triggered if the current behavior deviates from the expected one.

As explained in section 4, the proposed framework has been implemented for the Linux OS by means of dynamic probes placed in the kernel code. To show the effectiveness of the approach, we applied the implemented scheme to an operational OTS-based system, i.e., an open source middleware platform for the development of safety critical systems, namely CARDAMOM[1]. As it will be discussed in section 5, the experimental results confirm that combining multiple monitors the overall detection quality can be improved. In particular, in our testbed the implemented detection strategy is able to uncover hang failures with 100% coverage, while minimizing false positives (it achieves almost 100% accuracy), at an acceptable overhead (we measured a 2% overhead related to the time to completion of remote method invocations).

2. BACKGROUND AND RELATED RESEARCH

Failure detection strategies can be implemented as local or remote detection modules, depending on whether they are or not deployed on the same node of the monitored component. Local and remote detection can be both implemented with query-based or log-based techniques. Query-based techniques rely on interrogating the monitored component health status, to discover potential anomalies. The query can be performed periodically (heartbeat techniques, such as [8, 9]), or implicitly each time another distributed component tries to invoke the component of interest [10]. On the other hand, log-based techniques consist in analyzing the log files produced by the component, if available [11, 12]. In fact, these may contain many useful data to comprehend the system dependability behavior, as shown by many studies on the field [13, 14]. Moreover, logs are the only direct source of information available in the case of OTS items.

Both query-based and log-based techniques can be considered as direct detection techniques: they try to infer the component health status by directly querying it or by analyzing data items directly logged by the component itself. A different approach would instead be to infer the health of the observed component by monitoring its behavior from an external viewpoint, along with its interactions with the environment. These approaches can be labeled as indirect failure detection techniques. As an example, the work in [7] exploits hardware performance counters and OS signals

[1] http://cardamom.objectweb.org

to monitor the system behavior and to signal possible anomalous conditions. For instance, as better detailed in section 3.2, an active hang is flagged if the instruction count executed by the process goes outside a given bound. However, this technique is not useful in the case of passive hangs, where instructions are not executed.
Indirect techniques have been especially adopted for intrusion detection systems. In these cases, anomaly detection approaches are used to compare normal and anomalous runs of the system. In [15] the execution of a program is characterized by tracing patterns of invoked system calls. System call traces produced in the operational phase are then compared with nominal traces to reveal intrusion conditions. Other solutions, such as [16, 17], are based on data mining approaches, such as document classification. They extract recurrent execution patterns (using system calls or network connections) to model the application under nominal conditions, and to classify run-time behaviors as normal or anomalous. Notwithstanding the high degree of accuracy that these solutions are able to achieve, they require high computational load (due to the huge volume of analyzed data), which is inadequate to perform on-line detection in critical systems. Moreover, they have been applied to intrusions, which are deliberate software faults, rather than on transient software faults. Hence, it is not clear whether they are able to detect software hang failures due to transient faults.

3. THE PROPOSED DETECTION APPROACH

3.1. Assumptions

System model. We aim to detect failures in complex OTS based mission- and safety critical software systems. These are often distributed on several nodes which communicate through a networking infrastructure. However, we focus on a single node of the system to perform failure detection and we do not care of system topology. Nodes are assumed to be organized in software layers and to be made up of several *Detectable Units* (DUs in the following), representing *atomic* software entities for which we want to detect failures. In the context of this work, we assume every single software process to be the DU, i.e., we perform failure detection at process (thread) level.

Failure model. Failure detection has to be performed with respect to a failure model. According to the failure classification proposed in [18] we focus on halt failures, i.e., failures which cause the delivered service to be halted and the external state of the service to be constant. We further distinguish halt failures into crash failures and hang failures. In the context of our system model, a DUs is crashed when the process terminates unexpectedly (e.g., due to run-time exceptions). Even if crashes can be considered as the most severe failures, their detection is fairly simple to be performed locally, since the process structure associated with the DUs is deallocated when the process crashes. This is not the case of hang failures, where the DUs behaves as halted, even if the process is not terminated. In particular we encompass two types of hang failures, namely active and passive hang. An active hang occurs when the process is still running (i.e., one of its threads, if any, consumes CPU cycles), but its activity is no longer perceived by other processes and by the user. Conversely, a passive hang occurs when the process (or one of its threads) is indefinitely blocked, e.g., it waits for shared resources that will never be released. In complex systems it is hard to tell whether a thread is currently subject to a passive hang, or it is deliberately blocked waiting for some work to be performed (e.g., this happens when pools of threads are used in multi-threaded server processes).
Hangs might be either silent or non-silent. In the former case the hang compromises DUs communication capabilities, hence the process does not respond to queries, such as heartbeats. In the latter case the process is still able to communicate, hence it responds to heartbeats or it writes to log files, even if the service is not delivered properly. For instance, this might be the case of a multi-threaded process where the hang might be localized on a thread different from the one that answers to queries.
The failure classification proposed in [18] also considers erratic failures, i.e., failures for which the service is delivered (hence it is not halted) but the provided outputs are erratic. In this work, we do not explicitly take into account such failures, since the detection of wrong values is application dependent: to identify value failures, the user has to be involved in the detection process.

3.2. Detection Framework

Our approach is based on indirectly (and locally) inferring the health of the monitored component by observing its behavior and interactions with the external environment. In particular, we propose to shift the observation perspective and to leverage OS support to detect application failures.
As stated in section 1, we designed the detection framework with two main objectives in mind: to achieve high detection quality in terms of coverage and accuracy, and to minimize the overhead due to the detection itself. By coverage, we mean the detector capacity to uncover failures. It can be measured as the number of detected failures divided by the overall number of failures which actually occurred. By accuracy we mean instead the detector ability of

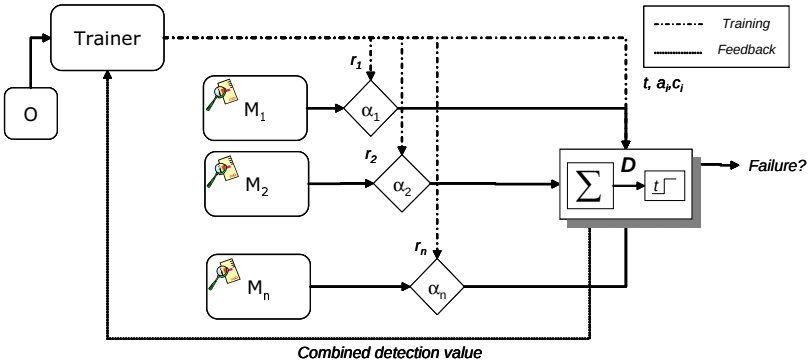


FIGURE 1: Scheme of the detection framework

avoiding false positives, i.e., to avoid wrong detection events when the DU is actually working properly.
Following the first objective, we propose a detection framework based on the combination of several OS level monitors. As suggested by intuition, combining multiple alarms coming from several sources allows to detect a higher number of failures, as well as to gain a better accuracy, if compared to single detectors. Most of indirect detection approaches in the literature, as the ones cited in section 2, are based on a single monitor, e.g., they monitor all the system calls made by the process under observation. The work in [7] proposed a software architecture (namely, *Reliability MicroKernel*) for development of detection policies using hardware- or software-based monitors (e.g., CPU hardware counters for instruction counting, OS context switches, system calls invocations): remarkable events in the system are monitored in order to detect hangs in the applications or the OS. In [7] applications hangs are detected by estimating an upper bound to instructions executed by each thread in each code section (e.g., a critical section or a loop body). The proposed approach is however unable to identify those hangs which prevent the application from execution, causing a passive wait (e.g., a deadlock).
In this work, we propose further detection policies, based on similar monitors, to also detect passive hangs. Moreover, we use an anomaly detection technique to analyze alarms from several monitors, in order to distinguish between false positives and actual failures. The basic idea is to regard failures as anomalies with respect to normal behavior of the system. As depicted in FIGURE 1, multiple monitors, $M_i \ \forall i = 1...n$, keep track of OS data related to a given process (thread). Each monitor is followed by a local alarm generator (α_i) which is responsible to trigger an alarm if the monitored value deviates from the normal behavior. To model the normal behavior, we associate a range of admitted values to each alarm, specifically $r_i = [r_i^-, r_i^+]$. If the monitored value n does not belong to the specified range within a given temporal window T, the alarm is triggered. The output of each α_i block is thus a binary variable defined as:

$$F_i = \begin{cases} -1 & \text{if } n \notin r_i \text{ in T} \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

More precisely, let $t(k)$ be the time of the $k-th$ measurement. T(k) is the temporal window containing the measurement, as depicted in FIGURE 2. The main reason for using a windowing technique is to minimize false positives in the detection process: if the number of times that $n \notin r_i$ within T(k) falls outside r_i, an alarm is signaled. A similar approach is a common practice in the context of intrusion detection studies, where n-grams are used to extract regular patterns from traces (the number of alarms is used instead of a time interval [15, 17]).

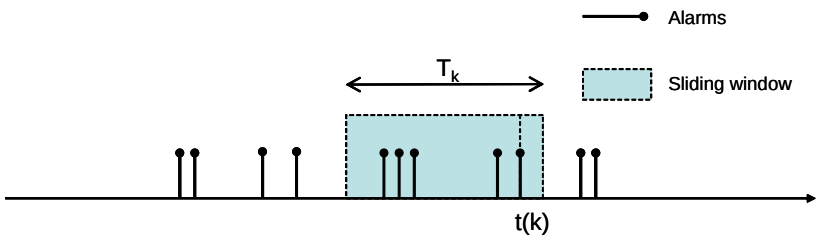


FIGURE 2: Windowing technique to discriminate anomalies

Actual values for each r_i have to be preliminary tuned during a so-called training phase (see the Trainer block in the FIGURE 1). The training is performed by analyzing both normal and faulty runs of the system. It is worth to note that,

even in normal executions of the system, anomalous values could appear due to the complexity of the applications we are addressing, e.g., a certain number of threads in a thread pool might be blocked, waiting for being activated; the long waiting time is thus not pathologic in this case. These anomalies are ruled out by the training phase, during which appropriate ranges are established for each monitor.
As previously stated, single alarms generated from monitors are combined in order to improve detection quality. The overall detection, which is performed by the global detector D in FIGURE 1, is achieved by means of a simple heuristic, defined as the weighted sum of single monitors contributions. In particular:

$$d = \sum_{i=1...n} p_i \cdot F_i \tag{2}$$

where weights p_i are defined as:

$$p_i = \begin{cases} a_i & \text{if } F_i = 1 \\ c_i & \text{if } F_i = -1 \end{cases} \tag{3}$$

This allows to take into account monitors detection capabilities: a_i and c_i are respectively the accuracy and the coverage measured for each monitor. Hence, in the weighted sum, positive contributions (each of them equals to a_i) are due to monitors which trigger alarms. On the contrary, negative contributions (each of them equals to c_i) are due to monitors claiming the absence of anomalies. The reason for this simple combination rule is twofold. First, we want to maximize the quality of detection. Hence, we decided to penalize inaccurate monitors since they are more likely to make mistakes than accurate ones. This is the reason for $p_i = a_i$, when a failure is signaled by the monitor M_i. In addition, we let monitors with high coverage to give higher (negative) contributions to the sum in equation 2. These monitors have to be trusted more than the other ones when claiming that a failure has not occurred (in the contrary case they would have signaled the anomaly with high probability). Second, we want to minimize the overhead. The proposed weighted sum is very simple, hence computationally efficient if compared to data mining or statistical classification techniques.
A failure is finally detected if d in equation 2 exceeds a given threshold t. The threshold is set during the training phase in order to be never exceeded during normal runs. Of course, its value can be tuned during system lifetime: this justifies the feedback in FIGURE 1. The oracle, O, is responsible for supervising the training and for evaluating a_i and c_i for each M_i and α_i. This evaluation can be done by means of fault injection campaigns (see section 5).

3.3. Monitors

Monitors design has been driven by the nature of the applications we are addressing. Since they are complex and safety critical applications, which usually make use of concurrency mechanism, we believe information about semaphores and other shared resources to be useful for detection. Furthermore, by studying several safety critical applications, we noticed that their workloads significantly stimulate the I/O subsystem, both to perform disk operations and network communications, hence we also monitored I/O interfaces in order to measure system throughput. To summarize, we decided to monitor the following measures:

1. System Calls;
2. OS Signals;
3. Task Schedulation Timeout;
4. Waiting times on semaphores;
5. Holding times for critical sections;
6. Processes and threads exit codes;
7. I/O throughput.

The detection system has been realized to work in UNIX environment. Even if the monitored measures are general (they can be observed in the most of OSs) we prefer to describe monitors by making explicit reference to the UNIX system in order to better justify design choices.
Each monitor is able to write log entries on log files. For each of them we defined a set of information which are written in a given format, along with the alarm timestamp.

3.3.1. System Calls Monitor

In UNIX environments, system calls are associated to numerical error codes which are returned if the exceptional events occurred during functions invocation. For this reason, we monitor the occurrence of error codes in order to detect failures. In fact, the presence of error codes is symptomatic of anomalous system behaviors.

This monitor takes into account all the occurrences of the cartesian product between system calls ID and error codes. Of course, only a subset of these couples represent acceptable values. Each time an error code is returned, the monitor records (i) the PID(TID) (Process (Thread) IDentifier) of the calling process (thread), (ii) the system call id (iii) the error code.

3.3.2. UNIX signals monitor

Since signals are generally used to notify exceptional events, their presence can be assumed to be far from system normal behavior, as for error codes. In UNIX environments, signals are commonly used to notify the occurrence of a given event, both from processes and the kernel. In the former case, they have coordination purposes, e.g., a signal could be sent to wake a process up from waiting, or to signal exceptional conditions. In the latter case, instead, signals are used either to inform a process about hardware and/or software exceptions, e.g., an invalid memory access, or the loss of a socket connection, or to signal normal events, e.g., to signal that I/O data became available. UNIX signals are able to explicitly signal process crashes (e.g., SIGSEGV) or process stop (e.g., SIGSTOP). However, they could also be the symptom of failures due to the loss of network connection. Additionally, application specific signals (e.g., SIGUSR1 or SIGUSR2) can be used to signal application specific conditions.
The monitor takes into account the whole set of UNIX signals. When a signal is triggered, it logs the following data: (i)PID (TID) for signal sender and receiver, (ii) the type of the signal.

3.3.3. Semaphores waiting times monitor

A long waiting for a given mutex to be released can be reasonably considered a symptom of indefinite waiting. In other words, the mutex is likely to be never released, hence the waiting process (thread) is intended to remain blocked. Measuring the waiting time is quite useful for the detection of hang conditions which are due to deadlock situations. It represents the time that a process waits before actually entering a critical section.
This monitor takes into account all the values of waiting times. When they exceed a given timeout, it records the following data: (i), PID (TID) of the waiting process (thread), (ii) the waiting time, (iii) the time from the beginning of waiting interval.

3.3.4. Critical sections holding times monitor

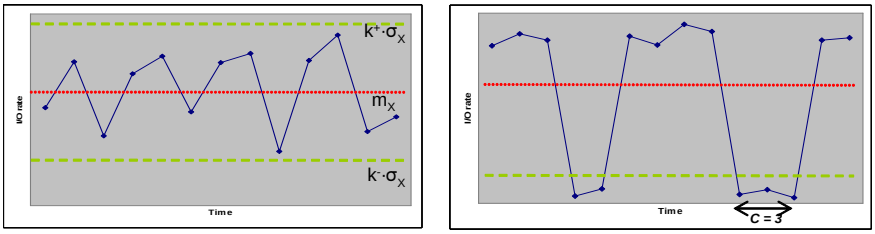
A process holding a critical section for a long time is likely to preclude shared resources usage to all the processes which are waiting for them. Thus, such a greedy behavior can reasonably be considered a potential cause of indefinite waitings and deadlocks. For this reason, we monitor processes holding time in critical sections, and we let the monitor trigger an alarm when holding times exceed a given timeout. In this case, the monitor logs the following data: (i), PID (TID) of the waiting process (thread), (ii) the holding time, (iii) the entering time in the critical section.

3.3.5. Task schedulation monitor

By monitoring schedulation timeouts, we are able to achieve information about process hangs. In particular, this measure is quite helpful for the detection of hang conditions which are not due to deadlock, e.g., a process may be waiting for messages coming from a sender process which has failed. For this reason, schedulation timeouts represent a complementary measure with respect to the previous two.
Similarly to the previous monitors, this monitor takes into account time values, i.e., schedulation delays. When the timeout is exceeded, the monitor logs the following data: (i) PID (TID) of the delayed process (thread), (ii) schedulation delay, (iii) last deschedulation time.

3.3.6. Processes and threads exit codes

Since we are addressing long running application scenarios, premature processes (threads) exits can be considered exceptional conditions deviating from system normal behavior. In fact, these event may be the symptom of crash failures or overloading conditions which forced the OS to kill the process (thread) unexpectedly.
This monitor takes into account all the processes (threads) deallocations event and it records data each time a process (thread) is deallocated. In particular, the following data are logged: (i) the PID (TID) of the exiting process (thread), (ii) the return code.



a b

FIGURE 3: I/O operations rate

3.3.7. I/O throughput monitor

This monitor takes into account I/O operations and it triggers alarms according to an algorithm we implemented by our own.
By means of preliminary fault injection campaigns, we observed that the number of I/O operations typically decreases during and after the activation of a software fault. In other words, faulty executions of the workload exhibited throughput losses in terms of the number of I/O operations per second. Based on this observation, we argue that monitoring the I/O operations rate would be very helpful into the detection of hangs, especially when they are non-silent.
The application scenarios we are referring to are characterized from having I/O bounded workload. For this reason, we monitor I/O operations both in terms of reads (writes) on files and socket descriptors.
I/O operations rate can vary both during the execution of a given workload and between different workloads runs (see FIGURE 3a). This makes quite difficult the definition of lower and upper bounds for I/O rates in charge of properly discriminating actual failures (i.e., losses) from normal fluctuations which can occur, in the form of operations slowdowns and bursts, even in the case of steady workloads (see FIGURE 3b). For this reason, the definition of an accurate model in charge of properly discriminating normal, and frequent, fluctuations from failures becomes quite complicated, thus increasing the risk of false positives in the detection process. We propose a parametric model which makes use of statistical average and standard deviation of the I/O traffic in order to detect anomalous fluctuations. It is based on the assumption that these statistics do not significantly change for a long execution period, i.e., the workload can be modeled as a stationary random process. This is a reasonable assumption with respect to the long running application workloads we are addressing. However, we also suggest a general solution suitable for workloads those statistics vary over time.
The monitor periodically samples the rate of I/O operations, then the computed sample value is compared to the statistical mean. If a given bound is exceeded (see equation 4), i.e., if the samples value significantly differs from the average, a given counter, namely C is incremented. On the contrary, if the bound is not exceeded, C is reset to zero.

$$y = \begin{cases} |X(t) - m_X| > k^+\sigma_X & \text{if } X(t) > m_X \\ |X(t) - m_X| > k^-\sigma_X & \text{if } X(t) < m_X \end{cases} \quad (4)$$

In the previous equation, $X(t)$ represents the I/O sample at time t, whereas m_X and σ_X represent, respectively, the precomputed mean and standard deviation values. Normal fluctuations around the mean have to be taken into account by properly tuning k^+ and k^-. These thresholds have to be set in order to minimize false positives. For this reason, it is desirable to have not false positives when training the monitor, i.e., during normal executions of the workload (see FIGURE 3a). If this is not achievable (see FIGURE 3b), the thresholds have to be set to filter out outliers, i.e., I/O samples which significantly differs from the statistical mean.
The C counter plays a key role in alarm triggering. During normal executions, a threshold value $\overline{C}$ has to be defined which, if exceeded, causes a warning to be triggered. This threshold is introduced to filter short-time deviations from the mean value, which normally occur during workload executions. In order to encompass longer deviations (e.g., long I/O bursts), a more complex model should be defined (e.g., by using Hidden Markov Models to characterize several system states; the same approach can be used on these states with different bounds and $\overline{C}$ thresholds).
This monitor logs the following data: (i) the value of the I/O sample which caused the alarm, (ii) the I/O operation (read/write), (iii) the exceeded bound (lower/upper), (iv) the C value. I/O operations are monitored with respect to processes, hence the PID is also recorded, without distinguishing between single threads.
Monitors are schematically summarized in TABLE 1.
The proposed detection architecture is not dependent on the particular working environment, i.e., it can be applied in several systems. Additionally, monitored measures (e.g., signals, semaphores and system calls) are available for

TABLE 1: Monitors parameters

Monitor	Domain	Output	Triggering Condition	α_i Parameters
UNIX System Calls	I={SysCall x ErrCode}	PID (TID) caller, system call ID, error code, timestamp	An error code is returned	None
UNIX Signals	N {0….32}	PID (TID) sender, PID (TID) receiver, shared, timestamp	At least one signal is triggered	None
Task schedulation timeout	R^+ {0, …+inf}	PID (TID), last schedulation time, schedulation delay, timestamp	Timeout exceeded	Timeout value
Waiting times on semaphores	R^+ {0,…,+inf}	PID (TID) waiting process (thread), mutex ID, waiting time, initial time of waiting interval, timestamp	Timeout exceeded	Timeout value
Holding times in critical sections	R^+ {0,…,+inf}	PID (TID), holding time, initial time of holding interval, timestamp	Timeout exceeded	Timeout value
Process/Thread lifecycles	{deallocation event}	PID (TID) deallocated process (thread), PID father process, return code, timestamp	Process/thread exits	None
I/O throughput	R^+ {0,…,+inf}	PID, I/O sample value, I/O operation type, exceeded bound, C value, timestamp.	Upper/lower bounds exceeded	Bounds

a large class of commonly used OSs. In the following we refer to the Linux OS in order to explain implementation issues. This UNIX-like OS provides many built-in kernel facilities we exploited to implement the monitors.

4. IMPLEMENTATION ISSUES

Monitors have been implemented by means of dynamic probing. To this aim, we used the *KProbes* framework to place breakpoints (i.e., special CPU instructions which “break” the execution of kernel code by means of interrupts) into the kernel code. Breakpoints have been placed in kernel functions which provide the monitored measures. When a breakpoint is hit, a handler routine is launched. It is executed just before the kernel code in order to quickly collect data (e.g., input parameters or return values of called function). This does not introduce any interferences on correct program execution, except for a very short delay.
The complete detection system has been implemented as a loadable kernel module. To this aim we exploited the *SystemTap* tool[2]. It allows to program breakpoint handlers by means of a high-level scripting language. *SystemTap* scripts are then translated into C code, encompassing also the *KProbes* framework. Synchronization issues between threads have been tricky to be monitored. Indeed, we were not able to have a complete view of all the lock/unlock operations on shared resources only by tracing kernel code. This is because when there is no contention between several threads, kernel system calls are often not invoked at all during operations on mutexes. For this reason we implemented a shared library to wrap PThread API provided by the standard *glibc* library which, in fact, overloads the PThread functions we want to monitor.
We did not experience significant delays in program executions when using *KProbes* and *SystemTap*, i.e., they do not cause significant overhead. In particular we compared the overall execution time of the application and the time to completion of a single remote method in the absence and in the presence of our detector. The overall execution time has not been significantly influenced by the detector, since it is mostly dependent on other random factors, such as network delay. As for the time to completion of a single remote method, we measured an average 2.35% overhead when the detector is used.

5. EXPERIMENTAL RESULTS

5.1. Testbed and workload

As stated in section 1, we use a real world safety critical application as case study. It is a CORBA based application, exploiting Fault Tolerance and Load Balancing facilities provided by the CARDAMOM middleware. The application implements a system in charge of updating flight data plans. Clients forward update requests to a *facade* process which forwards them to several processing servers (which belong to a load balancing group). They perform the actual update; the *facade* collects server responses and informs the clients that the request has been served. Communications between processes is performed by means of a Data Distribution System (DDS). In order to fulfill

[2] `http://sourceware.org/systemtap/`

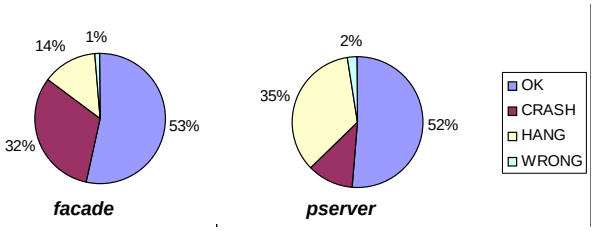


FIGURE 4: Outcomes of the injection experiments

dependability requirements, the *facade* process is replicated by using CARDAMOM mechanisms. The application is deployed on three nodes of a cluster of computers installed at our research laboratory. Each node is equipped with a Intel Xeon processor and with 4Gb of RAM memory.

5.2. Fault injection campaigns

In order to evaluate detector performances we conducted fault injection experiments, i.e., we corrupted application source code in order to emulate software faults. We refer to the injection framework described in [19] to inject software faults. It defines the 17 most representative classes of software faults, e.g., missing function calls (MFC), wrong value assigned to a variable (WVAV), which are also mapped to ODC software faults classification provided in [4].
We injected faults into both the facade and one processing server (*pserver* in the following), accounting for a total number of 88 injections (72 faults into the *facade*, the others in the *pserver*). Injected faults resulted in different failures according to the percentages reported in FIGURE 4. “Wrong” means that erratic failures occurred which are not considered in this work. However, they account for the minority of the observed failures. “OK” means instead that the injected fault did not result in a failure.

5.3. Preliminary Results

The proposed detection approach has been evaluated in terms of accuracy and coverage metrics (see section 3.2). The accuracy is measured as $1 - N_e/N$, where N is the total number of normal execution traces, i.e., traces where no failures occurred, whereas N_e is the number of traces erroneously labeled as anomalous by the single monitor. The coverage is instead measured as N_c/N_f, where N_f is the number of traces where failures actually occurred and N_c is the number of traces correctly labeled as anomalous.
As for tunable parameters (see TABLE 1), they have been empirically trained. Both timeout values for semaphores waiting times and tasks schedulation have been set to 1 second, as well as for the I/O sampling rate and the sliding window length. Of course, automatic training should be performed to set parameters values optimally. However, results show that the overall approach is robust with respect to sub optimal choices. In any case, we reserve to devote future work to perform optimal training.
TABLE 2 shows accuracy and coverage values for each monitor and with respect to injection target processes (namely *facade* and *pserver*). The metrics have been evaluated by performing a first set of experiments. The results are relative to the capacity of detecting both passive and active hang failures. We can notice significant differences in single monitors performances, also depending on the monitored process. The differences between *facade* and *pserver* are due to several factors. The two processes are inherently different: they make a different use OS resources and each one has its own failure modes, even if both are injected with the same type of faults. Hence, it is important to have multiple monitors, since their performance can be influenced by the monitored process. For instance, the holding times on critical sections shows a significantly different coverage with respect to the monitored process (0% for the *facade*, and 89.66% for the *pserver*). Nevertheless, there are monitors which show good results independently from the monitored process. In our case, the I/O throughput monitor exhibits high accuracy and coverage values for both the processes. It can be noticed that the process and threads exit codes and OS signal monitors exhibit a very low coverage with respect to hang failures for both the processes. However, they demonstrated their usefulness at detecting crash failures, with 100% coverage. The same set of experiments used for TABLE 2 has been used to set the t threshold (see equation 2) for both the processes. It is interesting to note that the threshold values are significantly affected by the monitored process, i.e., it is $t = -468$ for the facade and $t = -195$ for the pserver. Once again, this is consistent with the different run-time behavior exhibited by the two processes.
TABLE 3 shows the accuracy and coverage obtained by combining single monitors by means of our global detector D. Values for a_i and c_i in the weighted sum (see equation 2) have been set according to TABLE 2. The measurements

TABLE 2: Coverage and accuracy of hang failures detection evaluated for each monitor.

Monitor	c_i *facade*	c_i *pserver*	a_i *facade*	a_i *pserver*
I/O throughput	100%	100%	90.74%	66.67%
Task Schedulation Timeout	83.33%	75.86%	61.11%	57.41%
Waiting times on semaphores	25%	96.55%	77.77%	97.0%
Holding times for critical sections	0%	89.66%	81.48%	90.74%
OS Signals	8.33%	0%	100%	100%
System calls	8.33%	17.24%	97.0%	100%
Processes and threads exit codes	0%	0%	100%	94.44%

have been performed by running the detector on a different set of injection experiments, where the percentage of observed outcomes were very close to those in FIGURE 4. The overall coverage and accuracy values (respectively c_{global} and a_{global}) reported in TABLE 3 confirm our intuition, that is, combining multiple monitors improves the overall detection quality. Specifically, the global detection is able to preserve the good coverage result of the I/O throughput monitor, while minimizing the number of false positives, hence improving the overall accuracy. For instance, with reference to the pserver process, using only an I/O throughput-based detector would have led to the same high coverage result. Nevertheless, the accuracy value would have been lower (i.e., $a_i = 66.67\%$) than in the case of the global detector (a_{global}=100%).
This result is achieved despite the empirical training of tunable parameters, which is approximative.

TABLE 3: Coverage and accuracy of hang failures detection evaluated for the global detector D.

Monitor	c_{global} (*facade*)	c_{global} (*pserver*)	a_{global} (*facade*)	a_{global} (*pserver*)
Global Detector	100%	100%	96.15%	100%

6. CONCLUSIONS

This paper proposed an indirect detection framework for uncovering hang failures in complex, OTS-based systems. The framework is based on using multiple OS-level monitors to capture the application behavior, and to trigger detection events based on a lightweight heuristic.
The main results of this work are summarized in the following. First, by means of experimental campaigns, we show how the combination of several monitors is effective to improve detection coverage and accuracy, at a reasonable overhead. Second, a certain set of monitors have been proposed in the paper, which we believe to be helpful at detecting active and passive hangs, other than crashes. Nevertheless, the framework is general enough to be extended with more sophisticated monitors, if required for the systems under study. Third, we found that monitoring I/O throughput is a valid means to detect hang failures, as demonstrated by individual coverage and accuracy values in TABLE 2.
Future activities will be devoted to a thorough evaluation of the framework on even more complex critical systems, in order to encourage a possible uptake by industries.

ACKNOWLEDGMENTS

This work has been partially supported by the Consorzio Interuniversitario Nazionale per l'Informatica (CINI) and by the Italian Ministry for Education, University, and Research (MIUR) within the frameworks of the "Centro di ricerca sui sistemi Open Source per la applicazioni ed i Servizi MIssion Critical" (COSMIC) Project (`www.cosmiclab.it`) and of the "Iniziativa Software" Project, an Italian Research project which involves Finmeccanica company and several Italian universities (`www.iniziativasoftware.it`).

REFERENCES

[1] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computer*, 33(6):518–528, 1984.
[2] H. Madeira and J.G. Silva. Experimental evaluation of the fail-silent behavior in computers without error masking. *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 350–359, 15-17 Jun 1994.
[3] S.S. Yau and Fu-Chung Chen. An approach to concurrent control flow checking. *Software Engineering, IEEE Transactions on*, SE-6(2):126–137, March 1980.

[4] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 2–9, 1991.
[5] S. Biyani R. Chillarege and J. Rosenthal. Measurement of failure rate in widely distributed software. In *FTCS '95:Proc. of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 424. IEEE Computer Society, 1995.
[6] P. Jalote Y. Huang and C. Kintala. Two techniques for transient software error recovery. In *Workshop on Hardware and software architectures for fault tolerance: experiences and perspectives*, pages 159–170. Springer-Verlag, 1994.
[7] Long Wang, Z. Kalbarczyk, Weining Gu, and R.K. Iyer. Reliability microkernel: Providing application-aware reliability in the os. *Reliability, IEEE Transactions on*, 56(4):597–614, Dec. 2007.
[8] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. 6th European Dependable Computing Conference (EDCC-6)*, pages 3–12. IEEE Computer Society, October 2006.
[9] Open, Secure, Scalable, Reliable, Unix for Power5 Servers, 2004.
[10] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.
[11] R. K. Iyer, Luke T. Young, and P. V. Krishna Iyer. Automatic recognition of intermittent failures: An experimental study of field data. *IEEE Transactions on Computer*, 39(4):525–537, 1990.
[12] T. T. Y. Lin and D. P. Siewiorek. Error log analysis: statistical modeling and heuristic trend analysis. *Reliability, IEEE Transactions on*, 39(4):419–432, 1990.
[13] A. Thakur and R. K. Iyer. Analyze-NOW - An Environment for Collection and Analysis of Failures in a Network of Workstations. *IEEE Trans. on Reliability*, 45(4), 1996.
[14] M. Kaâniche C. Simache and A. Saidane. Event Log based Dependability Analysis of Windows NT and 2K Systems. *Proc. of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, December 2002.
[15] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for unix processes. *Security and Privacy, 1996. Proc., 1996 IEEE Symposium on*, pages 120–128, 6-8 May 1996.
[16] R. P. J. Chandra Bose and S. H. Srinivasan. Data mining approaches to software fault diagnosis. In *RIDE'05: Proc.of the 15th Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications*, pages 45–52. IEEE Computer Society, 2005.
[17] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.
[18] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. on Dependable and Secure Computing*, 1(1):11–33, 2004.
[19] João A. Durães and Henrique S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.