Int. J. of Critical Computer-Based Systems

# OS-Level Hang Detection in Complex Software Systems

**Antonio Bovenzi**
**Marcello Cinque**
**Domenico Cotroneo**
**Roberto Natella**

Dipartimento di Informatica e Sistemistica
Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
{antonio.bovenzi,macinque,cotroneo,roberto.natella}@unina.it

**Gabriella Carrozza***

SESM SCARL
Via Circumvallazione Esterna di Napoli, 80014
Giugliano in Campania, Naples, Italy
gcarrozza@sesm.it

*Corresponding author

**Abstract:**
Many critical services are nowadays provided by large and complex software systems. However the increasing complexity introduces several sources of non-determinism, which may lead to *hang failures*: the system appears to be running, but part of its services are perceived as unresponsive. On-line monitoring is the only way to detect and to promptly react to such failures. However, when dealing with Off-The-Shelf based systems, on-line detection can be tricky since instrumentation and log data collection may not be feasible in practice.

In this paper, a detection framework to cope with software hangs is proposed. The framework enables the non-intrusive monitoring of complex systems, based on multiple sources of data gathered at the Operating System (OS) level. Collected data are then combined to reveal hang failures. The framework is evaluated through a fault injection campaign on two complex systems from the Air Traffic Management (ATM) domain. Results show that the combination of several monitors at the OS level is effective to detect hang failures in terms of coverage and false positives and with a negligible impact on performance.

**Keywords:** Failure Detection, Hang Failures, On-line Monitoring, Critical Software Systems, Operating Systems

Copyright © 2009 Inderscience Enterprises Ltd.

## 1 Introduction

Software faults represent today a major dependability threat for complex software systems. Testing and static code analysis are widely adopted techniques to remove such defects or "bugs" in a system under development. However, as shown by field data studies (Sahoo et al., 2010; Chillarege et al., 1995; Sullivan and Chillarege, 1991), a large slice of software faults are activated during the operational phase when transient conditions occur (e.g., overload, timing errors, and race conditions). Static analysis and testing techniques fail when dealing with this kind of faults, 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 non-determinism in the application behavior.

For these reasons, faults have to be treated during the use phase of the system, by detecting the occurrence of failures due to their activation. To this aim the execution state of the system has to be continuously monitored in order to reveal if one or more components are not running correctly.

However, there is a class of failures, namely *hang failures*, that pose serious issues to the failure detection. These failures cause the system to be partially or totally unresponsive, although it appears to be running; they can be due to infinite loops and indefinite wait conditions.

Existing detection techniques simply poll the health status of system components (i.e., heartbeat mechanisms), analyze system log files to uncover error messages and their correlation with failures or control the levels of CPU utilization. It is clear that the nature of hang failures prevent traditional techniques to be effective. For instance, a process may still be able to communicate even if the service is not delivered properly; this might be the case of a multi-threaded process in which the thread that answers to queries is not the one in which the hang actually occurred. At the same time, a stuck process may not be able to log events.

These problems are exacerbated when dealing with complex mission and safety critical software systems. Today these systems are being developed as the composition of several Off-The-Shelf (OTS) software modules and complex multi-threaded components. The unavailability of the source code complicates the detection task, since no extra-code can be added to observe the execution state. In addition, due to their particular criticality, these systems pose stringent requirements to failure detection:

- maximize the number of detected failures, in order to avoid catastrophic consequences;
- minimize the number of false positives, in order to avoid unnecessary (and costly) recovery actions
- minimize the latency of the detection, in order to timely trigger the proper countermeasures;
- minimize the overhead of the detection framework limiting the impact on the performance of the system.

To these ends, this paper proposes a lightweight and non-intrusive failure detection framework to reveal the occurrence of software hangs. It relies on several

simple *monitors* which exploit the Operating System (OS) support to trigger alarms when the behavior of the system differs from the nominal one. For instance, we infer indirectly the state of the system by monitoring different variables such as the waiting time on semaphores or the holding time into critical sections. Nominal behavior has been modeled experimentally by means of a training phase. To combine alarms from *detectors* we use the Bayes' rule and a *detection event* is triggered if the likelihood that *hang failure* occurs exceeds a given threshold. Our experimental results show that this framework increases the overall capacity of detecting hang failures (it exhibits a 100% *coverage* of observed failures) while keeping low the number of *false positives* (less than 6% in the worst case), the *latency* (about 0.1 seconds in average) and with a negligible impact on performance (less than 10% in the worst case). Moreover, it can be used even when OTS modules are used, because there is no need to modify the source code of the application.

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 framework on two complex systems from the ATM domain, which are based on OTS and legacy components; we performed fault injection experiments to accelerate the process of data collection.

This paper extends our previous results on OS-level hang detection presented in Carrozza et al. (2008). In particular, (i) we propose a more sound combination scheme to trigger *detection events*, (ii) we introduce additional monitors to collect events related to network sockets status, and (iii) we perform a more extensive experimental evaluation. More in detail, in order to generalize the previous results, we analyze one more case study: the *SWIMBOX*. The proposed case study is a complex and OTS based system which has been implemented in the framework of the SWIM SUIT FP6 European project.

The rest of the paper is organized as follows. Section 2 presents the related work on hang detection, while the proposed detection approach is described in Section 3. Implementation details are provided in Section 4, and the results of the experiments are presented in Section 5. Finally, Section 6 ends the paper with conclusions and directions for future work.

## 2 Related work

The problem of hang failures can be mitigated by removing software faults in advance. **Debugging techniques** use static and dynamic source code analysis to identify hang root causes. In Shen et al. (2005), the disk I/O subsystem is modeled analytically; the model is then compared to execution traces, to identify workload conditions under which performance is suspiciously low, and to fix anomalies (e.g., by improving disk I/O scheduling heuristics). In Wang et al. (2008), runtime traces are exploited to search for potential hang points within source code, to avoid unnecessary end-user waits. In Engler et al. (2000), developers' knowledge about the system is exploited to formulate coding assertions, and to check the source code for violations. Assertions enforced on the Linux kernel concern memory management errors, temporal ordering of operations, and deadlocks. Debugging techniques are useful to avoid the occurrence of hangs only when the root cause can be easily pinpointed into the source code. Unfortunately, they are not suitable to identify

failures occurred during the use phase of the system because of the activation of complex and transient conditions. On line monitoring and failure detection are thus the only way to uncover these residual faults.

One approach to hang failure detection is represented by **query-based techniques**. They rely on probing the monitored component health status (either locally or remotely) to discover a failure (Chen et al., 2002). The query can be performed by periodically sending "heartbeat" request and waiting for "alive" reply to that message, or a timeout can be enforced to detect anomalous slow responses. In Herder et al. (2006), a query based technique is adopted to detect stalled OS processes in the Minix 3 OS, by using heartbeat requests. This approach requires that the monitored process is a "server" process, i.e., the process performs some work when it receives a request from Inter-Process Communication (IPC) channels. Moreover it assumes that, at given time, the process can only serve a request or respond to the heartbeat. This approach has been extended in Cotroneo et al. (2010) by adapting the timeout at run-time on the basis of past heartbeat replies. Unfortunately, this approach has some drawbacks. On one hand, when dealing with multi-threaded systems, the hang might be localized on a different thread than the one that replies to the heartbeat. Hence, the component get the heartbeats correctly, while other components are stuck. On the other hand, the approach is not suitable for OTS based and legacy systems, because it requires *(i)* to specify heartbeat requests in a format that can be managed by the system, and *(ii)* to modify the application in order to send replies.

Traditional failure detection approaches include **log based techniques**. They perform the on-line analysis of log messages produced by the system to infer the occurrence of a failure. In particular, they are often adopted to diagnose failures due to hardware faults, by using statistical analysis and heuristic rules (Iyer et al., 1990; Lin and Siewiorek, 1990). Data mining and language processing techniques have also been adopted to automatically analyze log files (Bose and Srinivasan, 2005). These techniques assume the occurrence of some events in the log file to detect a failure; unfortunately we cannot rely on the availability of log messages when dealing with hang failures since the system may be unable to execute and thus to produce log messages (e.g., a stuck component cannot return an error code or cannot throw an exception).

**Hardware monitoring** techniques are also used in hang failure detection. These techniques require special extra hardware such as watchdog timers to detect software hangs. Timers are periodically reset in failure free conditions; otherwise, an alarm is triggered (a Non-Maskerable Interrupt) to signal that the timer has expired David et al. (2007). However, these approaches are not able to detect infinite loops where the application is not stuck thus not preventing the monitored events to occur (e.g., resetting the timer). Moreover hardware support may be not available.

Our approach belongs to the class of **anomaly based detection techniques**. These techniques rely on *(i)* the continuous monitoring of the status of system variables (e.g., CPU consumption) and *(ii)* on the comparison of these data with traces of normal and anomalous executions. Anomaly based detection has been adopted in several contexts, such as intrusion detection (Forrest et al., 1996; Lee and Stolfo, 1998) and hardware failure detection (Zheng et al., 2007; Pelleg et al., 2008), by exploiting data collected at the network layer (e.g., about TCP connections) and at the hardware layer (e.g., CPU, I/O, and memory usage).

Monitoring OS level variables is also exploited in Podgurski et al. (2003). Authors propose to use system behavior information (e.g., system call traces, I/O requests, call stacks, context switches) and a multi-class classifier to build a diagnosis tool. However this approach has a not negligible overhead (all system call parameters are recorded) and is not suited for failures which cannot be reliably reproduced as *hang failures*.

The work appeared in Wang et al. (2007) is the closest to ours; it proposed a detection approach at the OS level using CPU hardware counters. On the one hand, applications hangs are detected by estimating an upper bound to the number of instructions executed in each code block of the application. On the other hand, system hangs are detected counting the number of instruction executed between two consecutive context switches (if the system is stuck it does not schedule any other process, and the counter value increases indefinitely). The proposed approach is effective against livelocks and infinite loops, but it does not allow to detect indefinite wait conditions. The approach also requires the analysis of the application code (to identify the code blocks), thus it may be not suitable for OTS based and legacy systems.

## 3 The proposed detection approach

### *3.1 System and Failure Assumptions*

The detection framework is designed to address complex and distributed software systems relying on OTS components. We assume that the system can be decomposed as a set of *Detectable Units* (*DUs* in the following). A *DU* represents the *atomic* software entity that can be monitored to detect failures. In this work detection is performed at process level, i.e., we consider OS processes, either single-threaded or multi-threaded, as *DUs*. OS processes are often adopted for architecting complex and distributed systems, by allocating a set of functionalities to each process (e.g., in the client-server paradigm, a server process listens for processing requests from clients); some examples of complex systems based on OS processes are represented by the case studies in this work (Section 5.1). *DUs* can be located in the same node or in different nodes, as shown in Figure 1.

This work focuses on hang failures, i.e., a *DU* does not provide its services anymore or services are delivered unacceptably late. When a process terminates unexpectedly (e.g., due to run-time exceptions), we assume a crash of the *DU*. Detecting such a failure is fairly simple, since OS promptly deallocates the structures associated to the processes that have crashed. This does not happen with hang failures, since they do not result in process termination: the *DU* rather survives behaving as halted. Hang failures can be further distinguished in *active* and *passive* hangs:

- *Active Hang*. It occurs when a process is still running but its activity may be no longer perceived by other processes because one of its threads, if any, consumes CPU cycles improperly;

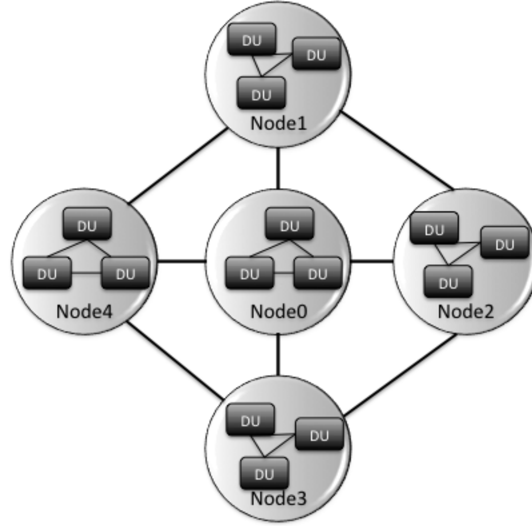



**Figure 1** System model.

- *Passive Hang*. It occurs when a process (or one of its threads) is indefinitely blocked, e.g., it waits for shared resources that will never be released (i.e., it encounters a deadlock).

Hangs might be either silent or non-silent. In the former case the hang compromises the communication capabilities of the process, e.g., it cannot reply to heartbeats. In the latter case, the process is still able to communicate, e.g., it responds to heartbeats or it generates log entries, even if the service is not delivered properly. In complex systems it is hard to tell whether a process (thread) is currently subject to a passive hang, because it can be deliberately blocked waiting for some work to be performed (e.g., this happens when pools of threads are used in multi-threaded server processes). Difficulties are also encountered with active hangs, because a process (thread) can deliver late heartbeat response, due to stressing workload and working conditions.

Along with crash and hang failures, systems may suffer value failures as well Avizienis et al. (2004). These do not cause the system to get halted nor delayed but the delivered service comes with erratic outputs. Awareness on the application logic and domain would be required to detect similar failures, as well as user involvement into the detection process. For this reason, we do not take these failures into account in our detection framework that is rather committed to be transparent to final users.

### 3.2 Detection Framework

We propose to leverage the OS support to perform system monitoring and to infer the health of *DUs* by observing their behavior and interactions with the external environment.

As stated in section 1, the proposed detection framework aims to achieve:

- high coverage, i.e., the ability to notify a failure, when the system is actually affected by a hang;
- low false positive rate, i.e., the ability of avoiding false alarms when the *DU* is actually working properly;
- low latency, in order to trigger alarms in due time.
- low overhead, in order to minimize the impact on the mission of the system as a whole.

To pursue these objectives, we propose to detect failures by leveraging several sources of information, through *monitors* placed at the OS level. Monitors concern resources used by the application and are realized by inserting software probes into the OS that are in charge of catching events.

Each monitor is in charge of observing a single resource and it is linked to an alarm generator ($\alpha_i$) which triggers the alarm in case of anomalies in the monitored resource. Monitors and alarm generators compose the overall detection system, named *detector*, depicted in Figure 2.

The final detection of a failure is performed by combining multiple alarms. As suggested by intuition, combining the alarms coming from multiple sources allows to detect a higher number of failures, if compared to detectors based on a single source. For instance, a passive hang does not lead to system call errors, but it can suspiciously increase the holding time into a critical section. This assumption is experimentally validated in section 5.

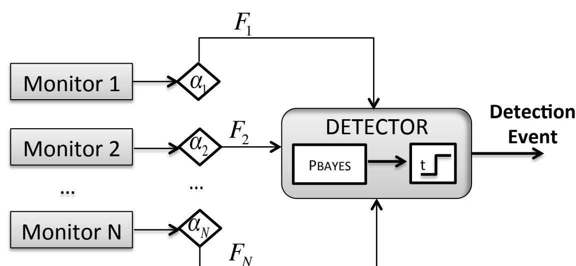



**Figure 2** Detection architecture.

Let $N$ be the number of monitors in charge of observing the resource usage for each $DU$. An alarm generator $\alpha_i$ collects the output of the $i$th monitor. An alert is produced by this monitor if the value of the observed variable ($v_i$) is out of a range $r_i = [r_i^-, r_i^+]$. The variable $v_i$ represents an event occurred for an OS resource (e.g., a thread waited for $10ms$ before entering a critical section), and the

range $r_i$ models the expected behavior of the *DU* with respect to the monitored resource. Moreover, we also take into account the bursty behavior of some events on OS resources, i.e., the events suddenly occur for a short time period and then disappear. To model this behavior and to detect anomalies in the burst length, the alarm generator also checks that $v_i$ is out of the range $r_i$ for $L_i$ consecutive times in a period $T_i$. Therefore the output of each $\alpha_i$ is a binary variable defined as:

$$F_i = \begin{cases} 1 \text{ if } v_i \notin r_i \text{ for } L_i \text{ times in the period } T_i \\ 0 \text{ otherwise} \end{cases} \quad (1)$$

To combine the outputs of all the monitors, we use the Bayes rule as the global detection logic (see equation 2). It allows to correlate existing beliefs (*a priori* probabilities) in the light of new evidence (*a posteriori*), i.e., to combine new data with existing knowledge about the occurrence of a given event.

$$P(F|\underline{\mathbf{a}}) = \frac{P(\underline{\mathbf{a}}|F)P(F)}{P(\underline{\mathbf{a}}|F)P(F) + P(\underline{\mathbf{a}}|\neg F)(1 - P(F))} \quad (2)$$

Applied to alarms and failures, equation 2 can be read this way:

- $F$ represents the event "faulty *DU*";
- $\underline{\mathbf{a}}$ is a vector containing the output of the alarm generators $\alpha$, i.e., $(F_1, F_2, ..., F_N)$.

The final *detection event* is triggered when $P(F|\underline{a})$ is greater than a given threshold value. The following probability distributions are estimated during the training phase:

- $P(\underline{\mathbf{a}}|F)$, represents the probability of detection. It is estimated as the number of occurrences of the $\underline{\mathbf{a}}$ vector under faulty executions, over the total number of vectors collected.
- $P(\underline{\mathbf{a}}|\neg F)$, represents the probability of false alarms. It is the number of occurrences of $\underline{\mathbf{a}}$ during fault-free executions.

Finally,

- $P(F)$ is the *a priori* probability of having a faulty *DU*. It can be estimated as $T$/MTTF (i.e., on the average, the *DU* becomes faulty once every MTTF/$T$, where T is the detection period and MTTF stands for Mean Time To Failure), if field data exist. Otherwise, it can be assumed by the literature where typical failure rates of complex software systems are provided. In our experiments, we assumed that $P(F) = 10^{-6}$ (Chillarege et al., 1995).

The parameters $r_i$, $L_i$ and $T_i$ are tuned during a preliminary training phase. The detection framework assumes that the parameters obtained during the training phase also apply during the operational phase of the system. Therefore, the parameters have to be gathered after observing the system execution for a time period *long enough*, in order to obtain a representative estimates that will apply also in the operational phase. This is a reasonable assumption with respect to the

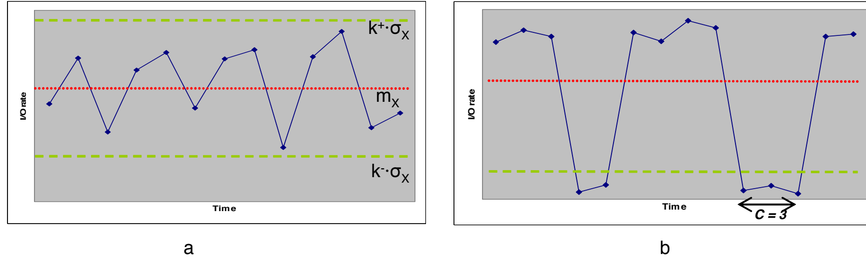



**Figure 3** Tuning of parameters.

critical systems we are addressing, since a significant amount of time is devoted to system validation that could be exploited to derive representative parameter estimates.

The training of the parameters should account for the variations in the monitored variables that occur during fault-free runs. The following heuristic approach has been adopted: the distribution of the $v_i$ (i.e., the frequency of values of $v_i$) is analyzed first, then a range $r_i$ that includes the most of the distribution is selected. For instance, the range can be selected by considering first order statistics (see Figure 3a), such as the mean ($m_{v_i}$) and the standard deviation ($\sigma_{v_i}$):

$$r_i = [m_{v_i} - k\sigma_{v_i}, m_{v_i} + k\sigma_{v_i}]. \quad (3)$$

An alternative approach, which has been adopted in our experiments, is to select the minimum ($\min_{v_i}$) and the maximum ($\max_{v_i}$) value in the distribution, namely:

$$r_i = [\min_{v_i}, \max_{v_i}]. \quad (4)$$

After selecting the range $r_i$, the parameter $L_i$ is set by taking into account the size of the bursts (see Figure 3b). These thresholds have to be set in order to keep low the number of false positives. For this reason, it is desirable to avoid false positives when training the monitor, i.e., during normal executions of the workload. Finally, the parameter $T_i$ is chosen empirically, i.e., by trying several candidate values and selecting the best one with respect to faulty and fault-free runs during training (e.g., minimizing false positives or latency, or maximizing coverage).

### *3.3 Monitors*

Bearing in mind the complexity of the target systems, in terms of concurrency and nodes distribution over a network, we consider the following variables for the detection process:

1. System call error codes;
2. OS signals;
3. Task scheduling timeouts;
4. Waiting time for critical sections;

5. Holding time in critical sections;
6. Process and thread exit codes;
7. Network sockets timeouts;
8. I/O throughput.

Hence, we implemented a set of monitors in charge of observing the above variables for each monitored *DU*; outputs are provided in the form of log files, formatted according to tight rules, and processed by the alarm generators. Although monitors have been implemented for the Linux OS, we believe that they can be adapted for other environments, since the monitored variables are not strictly dependent on the working environment.

### *3.3.1 System calls monitor*

In UNIX environments, system calls are associated to numerical error codes which are returned if exceptional events occur. Hence, the presence of error codes can be symptomatic of an anomalous system behavior.

All the occurrences of the cartesian product between system calls ID and error codes are considered. However, only a subset of these couples is meaningful. 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*

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 waiting process, or to notify 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. In UNIX environments, for example, signals are able to explicitly signal the crash (e.g., SIGSEGV) of a process. Additionally, they can be used to signal application specific conditions (e.g., SIGUSR1 or SIGUSR2) or they could represent the symptom of a failure, e.g., due to the loss of network connection. Therefore we believe that monitoring signals could be relevant for hang detection.

When a signal occurs, the monitor logs the following data: (i) PID (TID) of the sender and the receiver of a signal, and (ii) the type of the signal.

### *3.3.3 Critical sections waiting times monitor*

A long wait 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 can be useful for the detection of passive hangs. It represents the time that a process waits before actually entering a critical section. The critical section is defined as a piece of code that must not be accessed by more

than one thread or process, and it is implemented in UNIX using synchronization primitives (in particular, UNIX semaphores and the PThread library).

When waiting times exceed a given timeout, the monitor 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. This greedy behavior can reasonably be considered a potential cause of passive hangs.

For this reason, when holding times exceed a given timeout, the following data are logged: (i) PID (TID) of the waiting process (thread), (ii) the holding time, (iii) the entering time in the critical section.

### *3.3.5 Task scheduling monitor*

Another source of information for detecting a hang failure is represented by the last time when a process or thread is scheduled; a hang can be occurred if too much time is elapsed since its last execution. In particular, this monitor is helpful for detecting 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, scheduling 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., scheduling delays. When the timeout is exceeded, the monitor logs the following data: (i) PID (TID) of the delayed process (thread), (ii) scheduling delay, (iii) last de-scheduling time.

### *3.3.6 Processes and threads exit codes*

For long running application scenarios, unexpected process (thread) 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. In turn, the exit of a process may cause an indefinite wait in other processes.

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.

### *3.3.7 Network sockets monitor*

The delay between two consecutive packets sent on a given TCP/IP socket (both from and to the monitored task) is measured, for each thread and individual socket. A timeout is enforced to detect process (thread) suspiciously silent when communication is not taking place.

The following data are logged when the timeout is exceeded: (i) the PID (TID) of the process (thread), (ii) the port number of the socket, and (iii) the IP address of the remote process that communicate with the monitored process.

#### *3.3.8 I/O throughput monitor*

A decrease in the number of I/O operations represents another potential symptom of hang failures. For instance, the hang of a process may prevent I/O operations usually performed by the process (e.g., writing to a log file). Therefore, we argue that monitoring the I/O operations rate may help in the detection of hangs. In particular, we monitor the aggregate throughput of I/O operations with respect to reads and writes on files and socket descriptors. The monitor periodically samples the rate of I/O operations with period $T$, then the sampled value is compared to the bounds for this monitor.

This monitor logs the following data when the bound on I/O rate is exceeded: (i) the value of the I/O sample which caused the triggering, (ii) the I/O operation (read/write), (iii) the exceeded bound (lower/upper). 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 table reports the triggering condition for each monitor, i.e., the condition which cause the monitor to log an alert. The entries are then analyzed by alarm generators to produce alarms if $L_i$ alerts are produced within the period $T_i$.

**Table 1** Monitors at the operating system level.

| *Monitor* | *Triggering condition* | *Domain* |
|---|---|---|
| UNIX system calls | An error code is returned | Syscalls × ErrCodes |
| UNIX signals | A signal is received by the process | Signals |
| Task scheduling | Timeout exceeded (since the task is preempted) | $[0, \infty]$ |
| Waiting time for critical sections | Timeout exceeded (since the task begins to wait) | $[0, \infty]$ |
| Holding time in critical sections | Timeout exceeded (since the task acquires a lock) | $[0, \infty]$ |
| Process and thread exit codes | Task allocation or termination | Lifecycle event |
| Timeout on a socket | Timeout exceeded (since a packet is sent over a socket) | $[0, \infty]$ |
| I/O throughput | Bound exceeded | $[0, \infty]$ |

## 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 the kernel functions providing the monitored measures. When a breakpoint is hit, an handler routine is launched and 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 interfere with program execution, except for a short delay.

The complete detection system has been implemented as a loadable kernel module. To this aim we exploited the *SystemTap* tool (`http://sourceware.org/systemtap/`). 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 monitor. 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 kernel system calls are often not invoked at all during operations on mutexes when there is no contention between several threads . 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.

## 5 Experimental results

### *5.1 Case studies*

In this section, we evaluate the proposed framework with respect to two complex applications from ATM domain.

#### *5.1.1 FDP Case Study*

The first case study is a complex distributed application for Flight Data Processing (FDP). 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 4; it is based on CARDAMOM , a CORBA middleware for developing mission and safety critical applications compliant with the OMG Fault-Tolerant CORBA specification. CARDAMOM is jointly developed by SELEX-SI and THALES, the two leading industries in the European ATM scenario; in this work we based on the open source community edition which is available at `http://cardamom.objectweb.org`. CARDAMOM 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`) as Object Request Broker. The architecture we refer in this paper is made up of several components:

- *Facade* : 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

**Figure 4** Architecture of the FDP case study.

published by using DDS. This component is replicated several times on different nodes, and FDP operations are balanced among servers with a round-robin policy.

- *Correlation component* : it collects flight tracks generated by radars, and associates them to FDPs, by means of Correlation Managers (CORLM in the Figure 4).

This case study includes a workload generator that sends random requests to the system, both for flight tracks and FDP updates.

### *5.1.2 SWIMBOX Case Study*

The SWIMBOX case study was developed in the framework of the European-wide initiatives aiming at pursuing global interoperability in the Air Traffic Management (ATM) domain. SWIM (System Wide Information Management) is the world recognized initiative (both in Europe and USA in the context of SESAR and FAA programmers respectively) aiming to enable several stakeholders, i.e., airports, airlines, military air defense, Area Control Centers (ACC) and Air Navigation Service Providers (ANSP), to share information on a really large scale. It is meant to be the software infrastructure able to provide the one-for-all information model for data exchange and interoperability, as well as common interfaces to access specific services, at domain level. To this aim, it is going to define a common dictionary in terms of data and services, as well to use Commercial Off-The-Shelf (COTS) hardware and software to support a SOA aiming to facilitate systems dynamic composition and to increase common situational awareness.

The proposed case study is actually pilot prototype for SWIM, the SWIMBOX, which has been implemented in the framework of the SWIM SUIT FP6 European project (http://www.swim-suit.aero/swimsuit/).

The overall system is a grid of SWIM nodes, physically deployed at stakeholders premises and referred as “legacy” nodes, which are the users of the SWIM common

infrastructure and which are allowed to access the SWIM bus through the SWIM-BOX. Only SWIM-BOX instances can directly exchange data and invoke services over the net, acting as mediators between legacy nodes and the SWIM bus. The high-level endpoint perspective is shown in Figure 5, in which the role of Adapters can be appreciated. These have been implemented to let legacy nodes unaware of the SWIM semantic till all of them will be aligned to SWIM in a very next future.

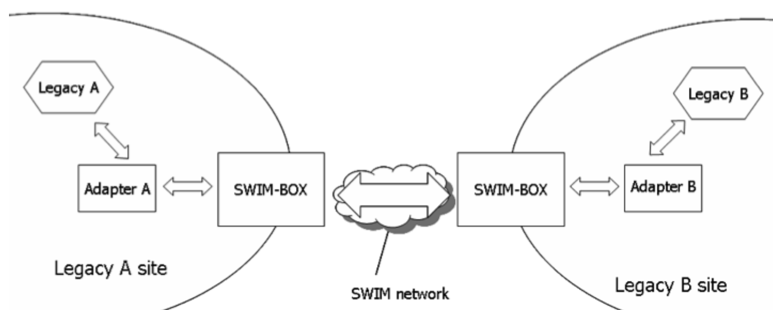



**Figure 5** End to end communication scenario between SWIM nodes.

The prototype architecture (see Figure 6) is organized in the following layers:

- *domain* level. It *(a)* defines a standard data representation embracing well defined models and collaborative approaches (i.e. FOIPS, ICOG2) and translates it in a flexible format (XML in the prototype), *(b)* exposes the external interfaces which define the domain specific operations on Flight, Surveillance and Aeronautical Data, e.g. create/update a flight plan, handover operation and, also, *(c)* define services to manage this domain specific components;
- *core* level. It implements synchronous/asynchronous communication pattern (i.e. request/reply, publish/subscribe), security services (i.e. encryption, authentication, access control), data storing (i.e. provides a transparently distributed and transactional storage mechanism allowing users to access shared data) and services registry.

It is worth noting that, in order to assure technology transparency, Publisher/Subscriber component actually provides an abstraction layer able to easily *masking* the underlying technology without impacting the uppermost domain level components. From a technological point of view, data distribution tasks can be accomplished by means of two different solutions: Data Distriubtion Service (DDS) and Java Messaging Service (JMS). The former is an OMG standard specification widely used in large scale networked applications. It is able to allow data transfer in the respect of QoS policies that can be customized according to the application needs. Commercial and open source implementations of the DDS standard are available. The SWIMBOX prototype is based on two different implementation of DDS: *(i)* the open source edition of OpenSplice DDS (OSPL) by Prismtech (http://www.opensplice.com) and *(ii)* the RTI DDS by Real-Time Innovations (`http://www.rti.com/`). Fault injection campaigns have been made for evaluating

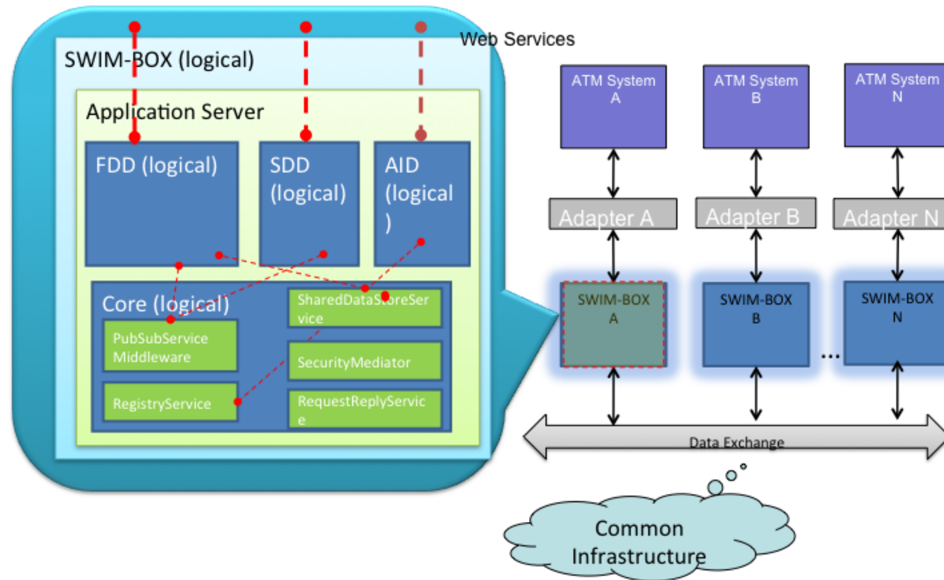



**Figure 6** SWIMBOX internals.

the effectiveness of the proposed approach. Due the the crucial role played by data distribution tasks into the most common SWIMBOX application scenarios, Publisher/Subscribe communication has been chosen as the injection target in order to understand how failures in the DDS components may propagate to the rest of the system. In fact, the communication layer may represent a dependability bottleneck for the whole system if faults at that layer are not properly coped with.

From the technical point of view, the application case study has been evaluated exploiting the FDD domain services of the SWIMBOX. The OpenSplice implementation has been used to accomplish DDS tasks. The application consists of two legacy entities, named the *Contributor* and the *Manager* respectively. Figure 7 describes an example of the interaction between the legacy systems. The *Contributor* acts as the subscriber, waiting for the information on Flight Object (i.e., a single entity including different information related to a flight) updates to be published. Also, it periodically reads all the available Flight Object summaries. Conversely, the *Manager* is in charge of *(i)* executing a given number of operations (e.g., Flight Data Object creation and update) at a fixed rate (20ops/sec), as well as of *(ii)* distributing data over the SWIM network exploiting the Pub/Sub middleware facilities. Once the operations have been completed the Contributor requires to unsubscribe from the FDD subsystem.

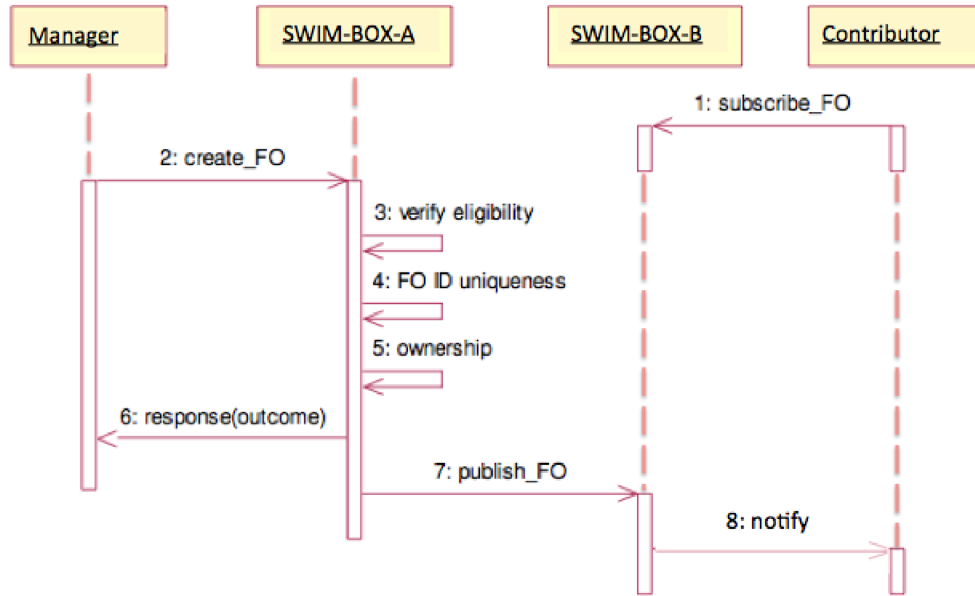



**Figure 7** Application interaction schema of the SWIMBOX case study.

### *5.2 Fault injection campaigns*

In order to evaluate the detection framework, 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 Duraes and Madeira (2006) to inject software faults. It defines the 17 most representative classes of software faults; for instance, fault classes frequently occurring in real systems are “missing function calls” (MFC) and “wrong value assigned to a variable” (WVAV). These fault classes are defined with respect to Orthogonal Defect Classification schema (Sullivan and Chillarege, 1991). The distributions of the injected software faults is provided in Table2.

Injected faults resulted in different failures, which are reported in Figure 8. “Wrong” means that content type failures occurred which are not considered in this work. “OK” means instead that the injected fault did not result in a failure. The analysis of the hang detection framework focused on experiments in which hang failures, either active ot passive, have been observed. Results reveal that software faults result in hang failures frequently; in particular, hang account for the majority of failures in the SWIMBOX case study. Passive hangs have been usually the effect of message loss or corruption, leading to an “indefinite wait” condition. Experiments were divided in two sets of equal size, namely *training set* and *test set*; the former has been adopted to tune the detectors whereas the latter to evaluate their effectiveness.

**Table 2** Source-code faults injected in the case study application.

| *ODC type* | *Fault Nature* | *Fault Type* | *Case study* | |
|---|---|---|---|---|
| | | | FDP | SBOX |
| **Assignment** | MISSING | MVIV - Missing Variable Initialization using a Value | 8 | 3 |
| | | MVAV - Missing Variable Assignment using a Value | 5 | 7 |
| | | MVAE - Missing Variable Assignment using a Value | 5 | 7 |
| | WRONG | MVAV - Wrong Value Assigned to Variable | 26 | 7 |
| | EXTRANEOUS | EVAV - Extraneous Variable Assignment using another Variable | 2 | - |
| **Checking** | MISSING | MIA - Missing IF construct Around statement | 2 | 1 |
| | WRONG | WLEC - Wrong logical expression used as branch condition | 3 | 2 |
| **Interface** | MISSING | MLPA - Missing small and Localized Part of the Algorithm | 2 | 1 |
| | WRONG | WPFV - Wrong variable used in Parameter of Function Call | 1 | 1 |
| **Algorithm** | MISSING | MFC - Missing Function Call | 13 | 5 |
| | | MIEB - Missing If construct plus statement plus Else Before statement | 1 | 1 |
| | | MIFS - Missing IF construct plus Statement | 1 | - |
| **Function** | MISSING | MFCT - Missing Functionality | 2 | - |
| | WRONG | WALL - Wrong Algorithm (Large modifications) | 1 | - |
| **Total** | | | 72 | 35 |

## *5.3 Results*

The goal of a detection system is to uncover as many failure as possible while at the same time keeping low the false positive rate. In order to evaluate our detection framework, we adopted the following quality metrics:

- *Coverage*: the conditional probability that, if there is a failure, it will be detected. It can be estimated from the ratio of the number of experiments in which the failure is detected to the number of experiments with a fault activated;
- *False positive rate*: the conditional probability that an alarm will be issued during fault free executions (i.e. application execution where no fault has been injected). It can be estimated from the ratio of false alarms (i.e. alarms triggered during correct execution) to the number of normal events collected.
- *Latency*: time interval between the fault activation (i.e. the time when the fault-injected code is executed) and detection (i.e. the time when an alarm is triggered);
- *Overhead*: the difference in the average execution time of application methods, by comparing executions with and without monitoring.

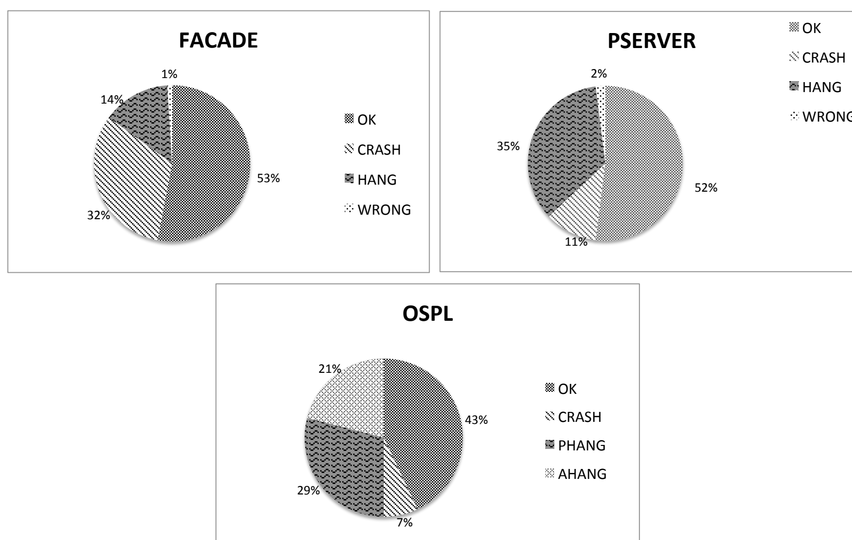



**Figure 8** Outcomes of the injection experiments.

We first evaluated the performance of individual monitors for both the case studies, with respect to the metrics mentioned above. For each monitor, a sensitivity analysis has been made, to tune the $T_i$ parameter. We considered timeouts within the range $[0.1s, 4s]$. The best performance and corresponding parameters for all monitors are shown in Tables 3 and 4.

Different monitors achieve different performance in terms of coverage, since they focus on failures impacting on different resources (e.g., a process may be indefinitely waiting for a mutex or for a message). Actually, monitors are unable to achieve full coverage keeping the False positive rate and Latency low (e.g., *Mutex Timeout* and *Sockets*). Monitors also provide different rate of false positives, which is remarkably high in some cases (e.g., *UNIX semaphores hold timeout* in the FDP case study). 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). To take in account this problem the combination rule (explained in section 3.2) has been adopted to prevent false alarms.

It is worth noting that, even if the detection framework can be applied to any application (it relies on several simple monitors at O.S. level), the performance of single monitors varies with the specific case study. For example, in the SWIM-BOX case study, the monitors on *Unix Semaphores* seem not to be helpful because the application does not call Unix semaphore primitives; instead, these monitors revealed some failures in the FDP case study. Therefore, we cannot claim that there is an individual monitor able to effectively detect hang failures in all scenarios. However, the inclusion of several monitors in the framework provides the potential for detecting hang failures in different scenarios; this goal can be achieved by tuning the combination rule, which accounts for the effectiveness of the individual monitors.

**Table 3** Coverage, false positive rate, and latency provided by the individual monitors in the FDP case study.

| *Monitor* | $T_i$ | *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% | - |
| *Scheduling threshold* | 4 s | 74.1% | 3.25% | 1912.22 |
| *Syscall error codes* | 1 s | 45.1% | 0.6% | 768.97 |
| *Signals* | 1 s | 45.1% | 0% | 816.57 |
| *Process/Thread exit* | 1 s | 45.1% | 0% | 830.64 |
| *Process/Thread Creation* | 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 |

To correlate all the different monitors alarms we adopted the Bayesian combination rule as explained in section 3.2. The conditional probabilities have been estimated by counting the frequency of the alarms in faulty and fault free experiments of the training set. Table 5 shows the performance achieved by the joint detector in the FDP and SWIMBOX case studies. The results seem to confirm the benefits of using a combined detector: it is able to achieve full coverage, while keeping low the false positive rate (it is comparable to the best rates in Tables 3 and 4) and the mean latency.

Finally, the overhead of continuous monitoring DUs at the OS level has been measured both for FDP and SWIM-BOX applications, by comparing the execution time *with* and *without* monitoring of representative methods provided by the case studies; moreover, we varied the request rate and the number of operations. Figures 9, 10 and 11 show the execution time observed with and without the detection framework. It should be noted that the overhead was lower that 10% in every case (in the SWIM-BOX case it is just over 2%), even during most intensive workload periods.

**Table 4** Coverage, false positive rate, and latency provided by the individual monitors in the SWIMBOX case study.

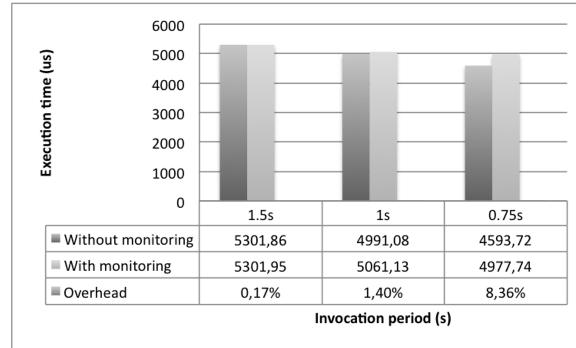
| *Monitor* | $T_i$ | *Coverage* | *False positive rate* | *Mean Latency (sec)* |
|---|---|---|---|---|
| *UNIX semaphores hold timeout* | - | - | - | - |
| *UNIX semaphores wait timeout* | - | - | - | - |
| *Pthread mutexes hold timeout* | 0.1 s | 100% | 9.7% | 0.1 |
| *Pthread mutexes wait timeout* | 0.1 s | 38% | 0% | 0.1 |
| *Scheduling threshold* | 2 s | 100% | 24.1% | 2 |
| *Syscall error codes* | 0.1 s | 12.5% | 8.2% | 15,41 |
| *Signals* | 0.1 s | 0% | 1,0% | 76.65 |
| *Process/Thread exit* | 0.1 s | 50% | 2.9% | 0.1 |
| *Process/Thread creation* | 0.1 s | 50% | 5.4% | 0.53 |
| *I/O throughput network input* | 0.1 s | 0% | 1.6% | 17.9 |
| *I/O throughput network output* | 0.1 s | 75% | 0.5% | 10.7 |
| *I/O throughput disk reads* | 0.1 s | 75% | 1.2% | 3.97 |
| *I/O throughput disk writes* | 0.1 s | 75% | 0.5% | 7.72 |
| *Sockets* | 2 s | 100% | 23.3% | 2 |

**Table 5** Coverage, false positive rate, and latency provided by the joint detector.

| | *FDP* | *SWIMBOX* |
|---|---|---|
| *Coverage* | 100% | 100% |
| *False positive rate* | 4.85% | 5.4% |
| *Mean Latency* | 100.26±135.76 ms | 100±33.33 ms |

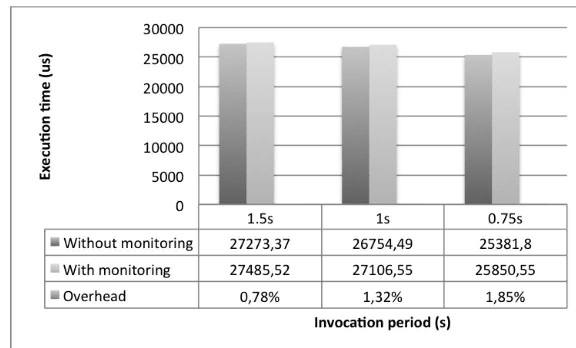
## 6 Conclusions

This paper proposed a framework for detecting hang failures in complex systems. The framework is based on monitors inserted at the OS level, in order to enable failure detection in the presence of OTS and legacy components. The monitors collect events related to OS resources (e.g., I/O devices, synchronization primitives), which are then analyzed by alarm generators using an anomaly detection technique.



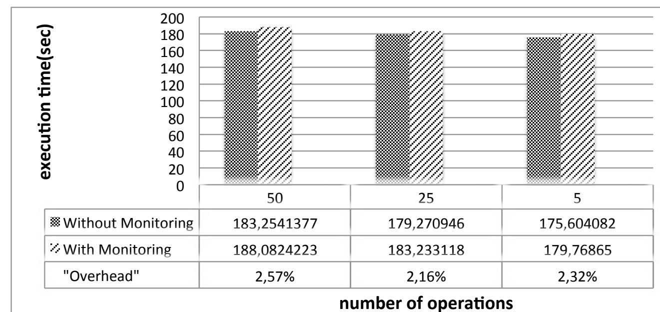

| | 1.5s | 1s | 0.75s |
|---|---|---|---|
| Without monitoring | 5301,86 | 4991,08 | 4593,72 |
| With monitoring | 5301,95 | 5061,13 | 4977,74 |
| Overhead | 0,17% | 1,40% | 8,36% |

**Figure 9** Overhead imposed to the execution of facade's *update_callback* method.




| | 1.5s | 1s | 0.75s |
|---|---|---|---|
| Without monitoring | 27273,37 | 26754,49 | 25381,8 |
| With monitoring | 27485,52 | 27106,55 | 25850,55 |
| Overhead | 0,78% | 1,32% | 1,85% |

**Figure 10** Overhead imposed to the execution of facade's *request_return* method.




| | 50 | 25 | 5 |
|---|---|---|---|
| Without Monitoring | 183,2541377 | 179,270946 | 175,604082 |
| With Monitoring | 188,0824223 | 183,233118 | 179,76865 |
| "Overhead" | 2,57% | 2,16% | 2,32% |

**Figure 11** Overhead imposed to the execution of SWIM-BOX's *main* method.

The proposed approach was evaluated by an experimental campaign on two real-world case studies. The non-intrusiveness of the approach allowed to deploy the detection framework even in the presence of OTS and legacy components. We noticed that the approach provides the best results when several monitors are combined. The combination of several monitors proved to be effective with respect to coverage by detecting all hang failures, thus confirming that monitoring at the

OS level is a good strategy for hang failure detection. Moreover, the approach is able to keep low the number of false positives and the computational overhead due to on-line monitoring (less than 6% and 10% in the worst case, respectively). Therefore, we believe that the proposed framework can effectively be deployed in real-world scenarios, in order to develop recovery strategies to be triggered when a failure is detected. The development of complex recovery strategies based on failure detection is thus a future research direction we aim to purse.

## References


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.

R.P.J.C. Bose and S.H. Srinivasan. Data mining approaches to software fault diagnosis. In *Research Issues in Data Engineering: Stream Data Mining and Applications, 2005. RIDE-SDMA 2005. 15th International Workshop on*, pages 45–52, 2005.

G. Carrozza, M. Cinque, D. Cotroneo, and R. Natella. Operating System Support to Detect Application Hangs. In *International Workshop on Verification and Evaluation of Computer and Communication Systems*, 2008.

W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002.

R. Chillarege, S. Biyani, 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.

D. Cotroneo, D. Di Leo, and R. Natella. Adaptive monitoring in microkernel OSs. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 66–72. IEEE, 2010.

Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Exploring recovery from operating system lockups. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association. ISBN 999-8888-77-6.

J. A. Duraes and H. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

Dawson Engler et al. Checking system rules using system-specific, programmer-written compiler extensions. In *Symp. on Operating Systems Design & Implementation*, 2000.

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, 1996.

J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. 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.

R.K. Iyer, L.T. Young, and P.V.K. Iyer. Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data. *IEEE Transactions on Computers*, pages 525–537, 1990.

W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

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.

Dan Pelleg et al. Vigilant: Out-of-Band Detection of Failures in Virtual Machines. *Operating Systems Review*, 42(1), 2008.

Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang. Automated support for classifying software failure reports. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X.

Swarup Kumar Sahoo, John Criswell, and Vikram Adve. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 485–494, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6. doi: http://doi.acm.org/10.1145/1806799.1806870.

Kai Shen, Ming Zhong, and Chuanpeng Li. I/O System Performance Debugging Using Model-driven Anomaly Characterization. In *Proc. USENIX Conf. on File and Storage Technologies*, 2005.

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.

L. 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, 2007.

X. Wang et al. Hang analysis: fighting responsiveness bugs. In *Proc. EuroSys Conf.* ACM, 2008.

Ziming Zheng, Yawei Li, and Zhiling Lan. Anomaly Localization in Large-Scale Clusters. *Proc. IEEE Intl. Conf. on Cluster Computing*, 2007.