Adaptive Monitoring in Microkernel OSs

Domenico Cotroneo, Domenico Di Leo, Roberto Natella
Dipartimento di Informatica e Sistemistica, Università degli Studi di Napoli Federico II
Via Claudio 21, 80125, Naples, Italy
{cotroneo, domenico.dileo, roberto.natella}@unina.it

Abstract

The microkernel *architecture has been investigated by both industries and the academia for the development of dependable Operating Systems (OSs). This work copes with a relevant issue for this architecture, namely* unresponsive *components because of deadlocks and infinite loops. In particular, a monitor sends heartbeat messages to a component that should reply within a timeout. The timeout choice is tricky, since it should be dynamically adapted to the load conditions of the system. Therefore, our approach is based on an adaptive heartbeat mechanism, in which the timeout is estimated from past response times. We implement and compare three estimation algorithms for the choice of the timeout in the context of the Minix 3 OS. From the analysis we derive useful guidelines for choosing the best algorithm with respect to system requirements.*

Keywords: Runtime monitoring, Fault detection, Microkernel, Minix 3 OS

1 Introduction

Operating Systems (OSs) are perhaps the most complex and critical part of the software stack in modern systems: they are made up of millions of lines of code (LoCs), provide fundamental services to user programs, and are tightly coupled with hardware resources. However, complexity raises concerns about dependability, since it is challenging to detect and fix software faults (*bugs*) in such a complex system within time-to-market constraints [1], as demonstrated by several studies on OSs failures occurring after release [2, 3]. Therefore, significant research efforts have been spent on dependable architectures for OSs, in order to mitigate the issue of software faults in the OS. *(*Microkernel) OSs have a modular based architecture that can potentially used for developing more dependable systems. The architecture is composed by several unprivileged components implementing the most of services (e.g., device drivers), and a tiny privileged component (*microkernel*) of a limited number of lines of code implementing basic services (e.g., scheduling) [4]. This architecture is adopted in some commercial OSs (MacOS X Darwin kernels and QNX [5,6]

Since components are unprivileged and execute in private address spaces, this architecture prevents a fault from affecting other components in the system, thus isolating the fault within the buggy component. Moreover, the microkernel architecture can be exploited for recovering from faults, by automatically restarting a crashed component. However, we believe there is still room for improving the microkernel architecture with respect to more complex failure behaviors than component crashes. In particular, this architecture can be exploited to deal with *unresponsive* components, i.e., the component is not crashed but it does not reply to any service request. This behavior is a common issue caused by software faults, which is related to deadlocks, infinite loops, and software aging [2, 3, 7].

In this paper, we propose a strategy for detecting unresponsive components in microkernel OSs, which is based on a *heartbeat* mechanism, i.e., a "ping" message is periodically sent to a component, and it is deemed unresponsive if a reply is not received within a timeout. To the best of our knowledge, in existing OSs such as QNX [8] and in Minix 3 [9] the timeout is chosen *a priori*. However, the choice of the timeout is the trickiest part of this mechanism. On one hand, if the timeout is too short, a component that is performing a time-consuming computation can be erroneously considered unresponsive. On the other hand, if the timeout is too long, an unresponsive component is detected after a long delay, which can negatively affect performance and dependability. On top of that, the expected response time from a correct component can vary because of several random factors, such as the user workload, I/O, and memory management. For these reasons, the proposed strategy adaptively selects the timeout, based on the history of response times of the component. We implemented and evaluated three algorithms for timeout choice in the Minix 3 microkernel OS, which are experimentally analyzed using simulations and real system execution. From experimental results, we derive some useful guidelines for choosing the best algorithm with respect to system requirements.

The paper is structured as follows. In section 2, we discuss related work in the field of OS dependability. In section 3, we provide an overview of the Minix 3 microkernel OS. In section 4, we discuss open issues in more detail and describe the proposed approach. In section 5, we evaluate timeout estimation algorithms. Section 6 closes the paper.

2 Related Work

We analyze past studies on OS dependability from two major categories: OSs *enhanced* with fault tolerance mechanisms and OSs *designed* to be reliable.

2.1 OSs enhanced with fault tolerance mechanisms

This class of OSs is not designed to be reliable, but their native architecture has been extended with additional components that increase their dependability. To the best of our knowledge, these OSs do not provide a native mechanism that detects unresponsive components. For example, Nooks [10] provides isolation for device drivers in order to prevent kernel memory corruption. Other works focus on fast recovery techniques. For instance, in [11] the "recovery box" technique is proposed to increase availability by storing system data in a stable area of memory, which is exploited to avoid a full reboot. In [12], hang detection in the Linux kernel is improved by introducing a monitor implemented as a kernel module. The detector tries to detect whether the system is stuck or not, by monitoring the interface between the drivers and the kernel. However, this approach was not aimed at hang detection in individual OS components. In [13], the support of special hardware (e.g., performance counters in the CPU) are exploited to detect hangs in the OS and in user programs. In [14], a machine learning approach is proposed to identify anomalies in virtual machines, by monitoring information collected by the hypervisor such as CPU usage and I/O operations. However, this approach comes with the overhead and complexity of virtualization, and it needs to be preliminarily tuned for a specific system workload.

2.2 OSs designed to be reliable

For these OSs, high reliability requirements drive their design. For example, in Windows 2000 kernel memory protection was added into the design of the OS. In the previous releases of Windows OS, errant applications could access the kernel space during the installation procedure. To protect the kernel space from misbehaving applications they designed Windows 2000 with a kernel address space protection [15].

Minix 3 is a microkernel OS whose design is completely devoted to achieve high reliability. Its architecture has been recently revisited and loosely resembles Minix 2. The first improvement was to execute all the servers and device drivers in user space on the top of a microkernel [4, 16]. Minix 3 has the capability to recover faulty device drivers without user intervention [9]. This feature is achieved by means of a dedicated component (namely Reincarnation Server, RS, or Driver Manager) that monitors drivers and recovers them when needed. If a monitored process is stuck or unexpectedly exits, RS restarts it. In [17], a resource reservation framework is introduced in order to add "temporal protection" to Minix 3. A new server (Constant Bandwidth Server, CBS) guarantees that a process executes within a given interval, and no one can monopolize the CPU. CBS implements a CPU reservation algorithm that replaces the native Minix scheduler. The reliability-driven design also involved the Minix Inter Process Communication system (IPC) [18]. The native IPC is enhanced with new primitives and rules that restrict the communication among trusted processes (e.g., file system server) and untrusted ones (e.g., drivers). In [19], the device driver isolation mechanism has been assessed using a practical approach based on extensive SoftWare-Implemented Fault-Injection (SWIFI) testing. SWIFI testing was also helpful to fix rare bugs that occurred even in the microkernel, and the results showed the high dependability level achieved by Minix 3. In [20] a filter driver, settled between the file system and disk driver, controls the integrity and the sequence of exchanged messages. The filter driver aims at protecting the file system from data corruption. Moreover, in Minix 3 RS constantly monitors device drivers by sending heartbeat messages. However, the RS heartbeat mechanism is still elementary and we show in section IV how it can be improved.

3 The Minix 3 OS

Minix is a microkernel POSIX-compliant OS. Its tiny microkernel encompasses low-level operations (namely IPC, scheduling, interrupt handling, clock management, MMU); other services are provided by user-space processes called *servers* (e.g., File System server, Process Manager). The basic component of Minix 3 is the *process*, in which servers are executed. As depicted in Figure 1, drivers also run in user-space processes, and their communication is restricted to microkernel and server layers.

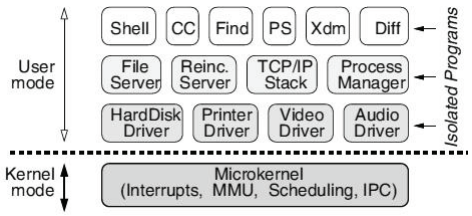


Figure 1. The design of Minix 3.

Communication among components is based on the message passing technique, thus the architecture works according to the client-server paradigm: when a message is received, the server process executes a computation and returns a result. Minix 3 also provides self-healing mechanisms [9]: it is possible to recover faulty components using the Reincarnation Server. This server monitors drivers and other servers at run-time. They are restarted when a crash, panic, or unexpected exit occurs in a process. RS is also able to detect unresponsive process by sending *heartbeat* messages to it. A process replies to the heartbeat when it completes its current computation. If a process does not reply within a *fixed* timeout, RS assumes that it is stuck and

recovers the component. However, the timeout choice is a non-trivial aspect in complex scenarios (section 4.1). The Minix 3 OS is composed by simple components and it lacks some complex features of modern OSs (e.g., swapping, multithreading), therefore its heartbeat mechanism was not designed to adapt the timeout at runtime. We develop our adaptive monitoring strategy on top of the RS (section 4.2).

4 Timeout Adaptation

4.1 Problem statement

In this section, we describe the proposed approach and provide details about its implementation in the Minix 3 OS. Our approach copes with the main issue of heartbeat mechanisms, namely the choice of the timeout. This choice affects detection for the following reasons: (i) the timeout should be equal or greater than the computation time needed by the monitored component under fault-free conditions, in order to not erroneously deem faulty a working process, and (ii) an exceedingly high timeout leads to a long detection delay. However, this computation time may be unknown, and can vary during execution due to several random factors, such as:

- Memory management (e.g., page faults for data or instructions can occur during execution; memory allocation operations take a variable amount of time);
- I/O (e.g., filesystem fragmentation may slow down disk operations);
- Algorithmic complexity (e.g., if a program takes $O(\log(n))$ time, the response time depends on the "input size" n);
- System overload (e.g., the number of processes currently in execution).

For existing heartbeat mechanisms in OSs (section 2), the timeout is fixed and it is chosen before running the component to be monitored. However, this approach does not always work properly, because it is difficult to foresee the most effective timeout *a priori* due to the mentioned issues.

4.2 Our approach

To overcome this limitation, our approach (Figure 2) is based on a *dynamic* timeout, that is, the timeout is continuously updated during execution (Figure 3). As previously mentioned, the timeout value should adapt to the expected computation time for a process. To this aim, we store the most recent response times of past heartbeats in a FIFO circular buffer of size *N* (i.e., a new value overwrites the oldest value in the buffer). Response times are processed to obtain the next timeout, using an *estimation algorithm*. This step is based on the assumption that *past response times can be used to estimate the near future*. The assumption follows from the observation that the above-mentioned factors affecting response times are unlikely to change within a short time period. For instance:

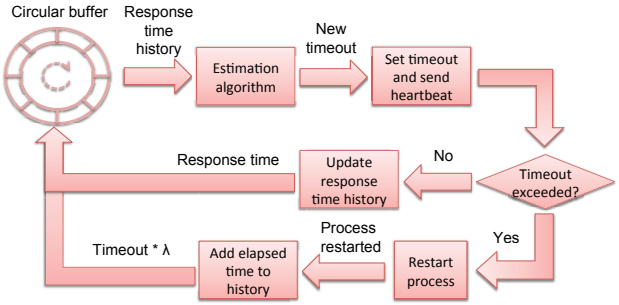


Figure 2. Overview of the proposed approach.

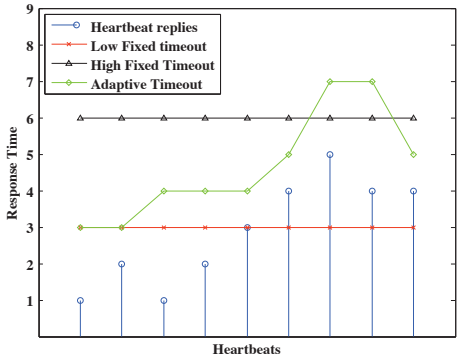


Figure 3. An example of timeout estimation algorithm. Vertical bars represent heartbeat response times of a monitored component.

- A page fault is followed by more page faults when the memory references access to new data or instructions;
- If a memory allocation or I/O operation is slowed down by memory exhaustion or fragmentation, following operations will likely be slow;
- An intensive user workload will likely last until the user gets his work done.

Therefore, on the basis of past response times, the approach is able to adapt the timeout to the current condition of the monitored process. Nevertheless, when an abrupt change of response times occurs, the approach is able to adapt to the new conditions. In particular, two opposite cases may occur, that is, the response time is significantly lower or higher than past values. In the former case, the estimated timeout will be higher than the processing time, leading to a higher detection delay; however, as more response times are collected, the estimated timeout will automatically adapt to the lower response time. In the latter case, a process can be erroneously deemed failed and restarted; in order to avoid further restarts due to wrong detection, a new value greater than the current timeout is added to the history, namely $\text{timeout} \cdot \lambda$ where $\lambda > 1$ (see Figure 2).

In order to estimate the next timeout, several algorithms could be conceived. We consider three algorithms derived

from past work; although they were developed in contexts different than ours, they are worth considering since they share some interesting similarities with our problem.

The *EWMA* (Exponential Weighted Moving Average) algorithm [21] was adopted by the TCP protocol to estimate the Round-Trip Time (RTT) over a connection. In this context, the RTT is exploited to set a timeout for TCP segments (i.e., segments are retransmitted if an acknowledge is not received within the timeout). The estimated RTT of a segment to be sent (*EstRTT*) is derived from the average RTT observed from previous segments (*SampleRTT*):

$$\begin{aligned}\mathrm{EstRTT}_{\mathrm{new}} = (1-\alpha)\cdot \mathrm{EstRTT}_{\mathrm{old}} \\ + \alpha \cdot \mathrm{SampleRTT}.\end{aligned} \quad (1)$$

In (1), the weight given to past RTTs decreases exponentially, in order to give more importance to most recent response times. Moreover, since the RTT may vary because of random fluctuations, an additional term is considered, which is a weighted average of deviations around the estimated RTT (*DevRTT*):

$$\begin{aligned}\mathrm{DevRTT}_{\mathrm{new}} = (1-\beta)\cdot \mathrm{DevRTT}_{\mathrm{old}} \\ + \beta \cdot \left|\mathrm{SampleRTT} - \mathrm{EstRTT}_{\mathrm{old}}\right|.\end{aligned} \quad (2)$$

The final estimated timeout is given by:

$$\mathrm{Timeout} = \mathrm{EstRTT}_{\mathrm{new}} + 4 \cdot \mathrm{DevRTT}_{\mathrm{new}}. \quad (3)$$

The *Max* algorithm is a heuristic for estimating the number of instructions executed by a process before blocking (e.g., the process invokes a system call or it is preempted by the scheduler) [13]. This algorithm estimates the timeout by picking the highest response time in the history of response times. The timeout is then multiplied with a constant factor γ to cope with random fluctuations:

$$\mathrm{Timeout} = \gamma \cdot \max\left(\mathrm{history}[0 \ldots N-1]\right). \quad (4)$$

The last algorithm, which we refer to as *WeightedSum*, was proposed in [22] for monitoring distributed objects in CORBA-based systems. In that context, heartbeat messages are sent to detect a crashed object. The heartbeat timeout is obtained by adding the weighted sum of previous response times ($N-1$ and 0 are respectively the most and the least recent values in the history) to the latest response time:

$$\mathrm{Timeout} = \mathrm{history}[N-1] + \frac{\sum_{j=0}^{N-1} \mathrm{weight}[j] \cdot \mathrm{history}[j]}{N}. \quad (5)$$

In [22], weights were found empirically and validated by means of simulations. We include this algorithm to study how this empirical choice performs in our context. In particular, the most recent value ($N-1$) is given the greatest weight, which is inversely proportional to the size N of the history (the greater the history, the lower the weight), and the remaining values have decreasing weights:

$$\mathrm{weight}[j] = \begin{cases} \frac{N+4}{N \cdot 4} & j = N-1 \\ \left(1 - \sum_{k=j+1}^{N-1} \mathrm{weight}[k]\right) & \\ \qquad \cdot \mathrm{weight}[N-1] & 0 < j < N-1 \\ \left(1 - \sum_{k=1}^{N-1} \mathrm{weight}[k]\right) & j = 0 \end{cases} \quad (6)$$

The proposed approach and the algorithms were implemented into the RS of Minix 3.1.2 with minimal intrusiveness. In particular, the implementation required to add 154 LoCs and to modify 5 LoCs, spanning over 6 source files.

5 Experimental analysis

5.1 Overview

The goal of our experimental analysis is to find out which estimation algorithm is the most suitable for dynamically adapting the timeout when response times vary. In order to do so, we consider a realistic scenario in which a server process is monitored by the RS, and response times can vary because of algorithmic complexity (section 4.1). We conducted several experiments on this scenario using different workloads and settings of algorithm parameters, in order not to bias the results. A workload is a sequence of requests for the monitored server process, in which each request takes a specific amount of time to be processed. For a given workload, we vary the parameters of algorithms within a range, and average the results over all settings. Since the number of experiments becomes large when several workloads and parameters are considered, we evaluated the algorithms using simulations to reduce the duration of experiments. This was accomplished in three steps, that is, (i) the scenario was actually implemented in Minix 3, (ii) the system was executed under several kinds of request, to profile the computation time needed for each request, and (iii) the measured response times were fed to a simulator in order to evaluate the algorithms. Additionally, the accuracy of simulation was validated by comparison of real and simulated executions. Algorithms are then compared with respect to two metrics relevant for the issue of timeout choice.

5.2 Test scenario

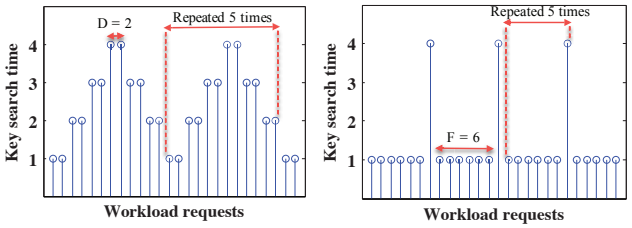
The test scenario encompasses the following processes:

- **Key Searcher Server (KSS).** It is the only process monitored by RS. We reproduced the situation in which response times vary because of algorithmic complexity. In particular, this process manages a Red-Black Tree (RBT), i.e., a self-balancing binary tree in which search, insertion, and delete operations are $O(log(n))$, where n is the total number of elements in the tree. An RBT is a realistic scenario, since it is adopted in real complex applications such as OSs (e.g., for memory and process management) and DBMSs (e.g., for storing table indexes). We implemented KSS as a Minix server that listens for search requests; the request input is the key to search, and the request output is a value associated to the key. In order to emulate a complex computation, we consider the scenario in which $n = 10^6$ and a search is repeated $6 \cdot 10^6$ times for each request.
- **Key Requester (KR).** This process sends requests to KSS for the value associated with a key. This process is responsible for generating our workload as described in the next section.

In the test scenario, the KSS receives messages from either RS or KR; the former sends heartbeat messages and the latter sends requests for RBT data. KSS replies to heartbeat messages with a "ping reply", and to KR with the requested data.

5.3 Workload

Since we want to compare the effectiveness of our algorithms under varying response times, we considered two workloads in which response times vary on purpose. Workloads are designed to frequently vary response times, in order to analyze the algorithms under worst cases. Workload W_1 (Figure 4a) is a sequence of requests that progressively increments and decrements the search time. In particular, we have a set of 4 keys $k_1, \ldots, k_4$, where the processing time for the i-th key is $t(k_i)$. We chose the four keys among all keys in the RBT such that $k_1 = \text{argmin}_{k_i \in \text{RBT}}(t(k_i))$, $k_4 = \text{argmax}_{k_i \in \text{RBT}}(t(k_i))$, and $t(k_4) - t(k_3) = t(k_3) - t(k_2) = t(k_2) - t(k_1)$. Keys are requested in the following order: $k_1, k_2, k_3, k_4, k_3, k_2$. Each key is requested D times, and this sequence is repeated 5 times. Workload W_2 (Figure 4b) represents a sequence of requests in which the processing time peaks for a short period. Specifically, we consider keys k_1 and k_4 from W_1. Key k_1 is requested F times, and key k_4 once; this sequence is repeated 5 times. Moreover, in order to avoid biasing the results, we executed several tests with different values of D and F ($D \in \{1, \ldots, 9\}$ and $F \in \{1, \ldots, 9\}$ respectively).



(a) Workload W_1 (D: step duration). (b) Workload W_2 (F: pulse distance).

Figure 4. Workload request sequences.

5.4 Performance measures

As criteria for comparing the algorithms, we take into account the need for an algorithm that is both fast and accurate. Therefore, we adopted two popular metrics [23], *False Positives* (FP) and *Latency* (L). FP is the number of times that the process is mistakenly restarted during a test (e.g., a process under heavy workload fails to reply before the timeout expires). Latency is the mean difference between timeout values and response times. It represents the expected detection delay in the case that the process will never reply to a request. The greater is the latency, the greater is the time to detect a faulty component. Latency is expressed in ticks, which is the elementary time unit in Minix (60 ticks = 1 second). It should be noted that we execute tests in a fault-free scenario, since an unresponsive process is eventually detected [23]. Therefore, both FP and L should be as close as possible to zero.

5.5 Simulations

In our experiments, we considered several settings of algorithm parameters, in order to take into account the sensitivity of the algorithms with respect to their parameters. We considered a set of reasonable settings for each parameter, and simulated each algorithm using every combination of settings. Table 1 shows the values we considered for each parameter.

Table 1. Algorithm parameters.

Algorithm	Parameter	Range	Step
EWMA	α	$[0.1, 0.9]$	0.1
	β	$[0.1, 0.9]$	0.1
Max	γ	$[0.1, 0.9]$	0.1
	N	$[1, 9]$	2
WeightedSum	N	$[1, 9]$	2

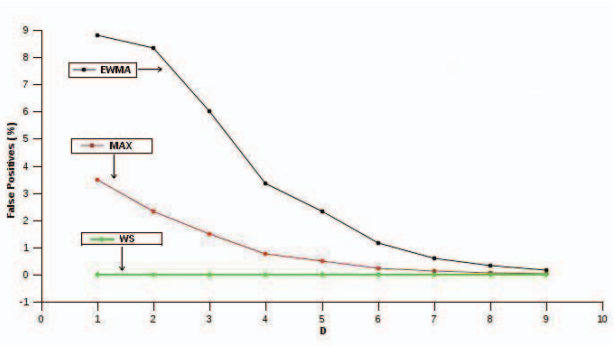
The simulator is a program that takes in input a sequence of response times (discussed in section 5.3). Given an algorithm, the simulator evaluates the timeout value estimated for the current heartbeat. It takes into account the FIFO policy adopted for storing past heartbeats, and the insertion of the elapsed time in the presence of a process restart, as shown in Figure 2. The simulator provides in output the average performance measures (section 5.4) for a given algorithm configuration. Finally, results from different algorithm configurations are averaged to obtain a summarizing result for a given workload.

We validated the results of our simulations against real executions for a subset of cases (we considered the cases $D = 1$ for W_1 and $F = 1$ for W_2). The number of false positives calculated by the simulator and obtained from the real system were always the same. The accuracy of the latency measure, in the worst case, was 97.46%. The slight inaccuracy of latency from the simulator is due to factors such as process scheduling and the time that RS needs to execute the estimation algorithms.

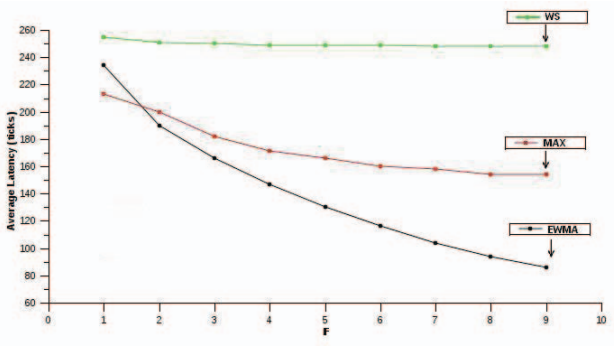
5.6 Results

From the simulation results, we obtained the average number of false positives and the average latency for each workload and for each algorithm. These measures are shown in Figure 5 for several values of D and F (section 5.3). In particular, Figure 5a and 5c show the percentage of false positives, i.e., the ratio between the number of requests not processed within the timeout, and the total number of requests in the workload. Figure 5b and 5d provide the mean latency in ticks.

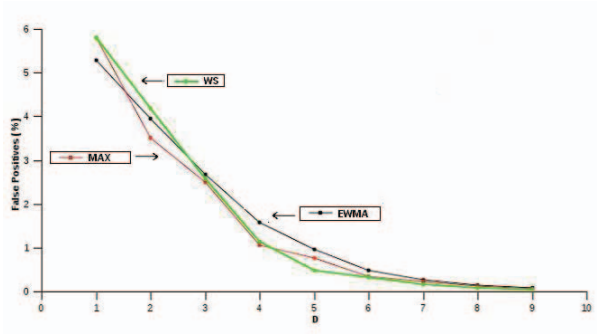
Since the results are obtained from worst-case workloads, they should not be interpreted as an absolute measure of quality of the algorithms. Instead, we analyze the results



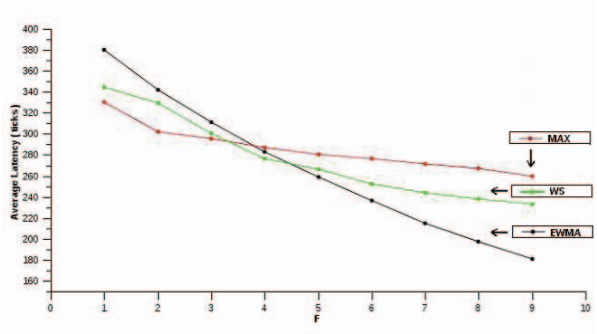
(a) False positives for W_1.



(b) Latency for W_1.



(c) False positives for W_2.



(d) Latency for W_2.

Figure 5. Simulation results.

to compare algorithms between each other in worst-case conditions. It can be observed that both the false positives and latency measures are monotonically decreasing with respect to D and F, for all algorithms. This result is justified by the fact that the algorithms perform better when fluctuations in the workload are rare (i.e., high values of D and F), since an algorithm may not be able to adapt to a change in response times thus causing false positives or high latency.

The most significant differences between algorithms were observed for workload W_1 (Figure 5a and Figure 5b). It appears that EWMA is the worst algorithm with respect to false positives, especially for low values of D. This result occurred because EWMA gives a lower weight to oldest values in the history than the other algorithms (in EWMA weights decrease exponentially). The weight of oldest values is relevant for workload W_1, since the sequence is long (in Figure 4a, the sequence is 12 samples long for $D = 2$) and the algorithm has to take into account the previous repetition of the sequence to properly set the timeout. Therefore, the sequence length prevents EWMA from adapting to increments in workload W_1. The Max algorithm behaves better than EWMA with respect to false positives, since the γ factor in eq. (4) prevents incremental variations in the workload from causing false positives. The WeightedSum algorithm gives the lowest percentage of false positives. This is due to the high timeout values produced by the algorithm (see Figure 5b). Before an increment in the response time occurs, which may cause a false positive, the algorithm adds to the latest response time an increment always greater than the variation in the workload (eq. (5)). However, high timeout values also cause a mean latency higher than the other algorithms.

In workload W_2, differences between algorithms are not as noticeable as W_1, even for low values of F. In particular, for low F the EWMA algorithm is not worse than the other algorithms (Figure 5c). This is due to the shorter sequence length than W_1 (for instance, the sequence is 3 samples long for $F = 2$), therefore weights for oldest values are not as important as in the previous experiment. When F increases ($3 \leq F \leq 5$), EWMA provides a slight higher percentage of false positives, since the sequence becames longer. Finally, when $F \geq 6$, although EWMA is still worse than the other algorithms, the absolute difference between algorithms is negligible, since peaks in the workload are rare. Instead, in W_2 Max and WeightedSum are very close with respect to both false positives and latency.

Our results can be summarized as follows:

- The estimation algorithm has a major impact on the proposed approach with respect to W_1 (i.e., incremental workload variations). Among the evaluated algorithms, there is not a best one with respect to both metrics, due to the trade-off between false positives and latency.
  - The WeightedSum algorithm should be preferred when false positives is the most important metric, at the cost of a higher latency.

  - It appears that EWMA should be preferred when latency is the most important metric. Nevertheless, the results does not take into account the cost of false positives, that is, the time required for killing and restarting a process, which negatively affects availability. Moreover, a high number of false positives may lead to wasted work and to loss of data (in the case of stateful processes), therefore we do not believe that EWMA is adequate for a general-purpose OS such as Minix 3.
  - The Max algorithm ranks in the middle, and it represents a trade-off between FP and L.
- In W_2 (i.e., sporadic variations) the estimation algorithms do not differ as significantly as in W_1. The algorithms provide similar false positives and latency, and the EWMA algorithm seems to be the most sensitive to F (the ranking of algorithms for $F \leq 3$ is opposed to the ranking for $F \geq 4$). In this regard, the WeightedSum and Max algorithms provide more stable measures.

6 Conclusions and future work

In this paper we proposed an approach to detect unresponsive components in microkernel OSs. Our approach is based on an *adaptive* heartbeat mechanism, in which the timeout is estimated from past response times. We implemented and compared three estimation algorithms in the context of the Minix 3 microkernel OS, with respect to false positives and latency of detection. From our analysis, we conclude that the WeightedSum or Max algorithm should be preferred, respectively, when little false positives are needed or when a trade-off between false positives and latency is needed. Instead, we do not believe the EWMA algorithm is adequate for a general purpose OS such as Minix 3, since it provides highly variable measures, and the potentially high number of false positives offsets the low latency it can provide.

Future work encompasses the possibility to evaluate the algorithm in different scenarios (e.g., by considering more random factors such as I/O and memory management), to provide an approach for tuning algorithm parameters, to evaluate the scalability of the approach (e.g., by increasing the number of monitored servers), and to analyze more estimation algorithms.

Acknowledgment

This work has been partially supported by the project "CRITICAL Software Technology for an Evolutionary Partnership" (CRITICAL-STEP, http://www.critical-step.eu), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 230672, within the context of the Seventh Framework Programme (FP7).

References

[1] E. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software*, vol. 15, no. 5, 1998.

[2] M. Sullivan and R. Chillarege, "Software Defects and their Impact on System Availability—A Study of Field Failures in Operating Systems," in *Symp. on Fault-Tolerant Computing*, 1991.

[3] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An Empirical Study of Operating System Errors," in *ACM Symp. on Operating Systems Principles*, 2001.

[4] J. Herder, H. Bos, B. Gras, P. Homburg, and A. Tanenbaum, "Construction of a Highly Dependable Operating System," in *European Dependable Computing Conference*, 2006.

[5] "Mac OS X System Architecture," Apple Developer Connection: http://developer.apple.com/macosx/architecture/.

[6] "The QNX Neutrino microkernel," QNX Developer Support: http://www.qnx.com/developers/docs/6.3.0SP3/neutrino/sys_arch/about.html.

[7] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software Rejuvenation: Analysis, Module and Applications," in *Symp. on Fault-Tolerant Computing*, 1995.

[8] "QNX Software Development Platform - High Availability Framework," QNX documentation: http://www.qnx.com/download/group.html?programid=18790.

[9] J. N. Herder et al., "Failure Resilience for Device Drivers," in *Conf. on Dependable Systems and Networks*, 2007.

[10] M. Swift, B. Bershad, and H. Levy, "Improving the Reliability of Commodity Operating Systems," *ACM Transactions on Computer Systems*, vol. 23, no. 1, pp. 77–110, 2005.

[11] M. Baker and M. Sullivan, "The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment," in *Summer USENIX Conf.*, 1992.

[12] D. Cotroneo, R. Natella, and S. Russo, "Assessment and Improvement of Hang Detection in the Linux Operating System," in *Symp. on Reliable Distributed Systems*, 2009.

[13] L. Wang, Z. Kalbarczyk, W. Gu, and R. Iyer, "Reliability MicroKernel: Providing Application-Aware Reliability in the OS," *IEEE Trans. on Reliability*, vol. 56, no. 4, 2007.

[14] D. Pelleg et al., "Vigilant: Out-of-Band Detection of Failures in Virtual Machines," *Operating Systems Review*, vol. 42, no. 1, 2008.

[15] B. Murphy and L. Bjorn, "Windows 2000 dependability," in *Conf. Dependable Systems and Networks*, 2000.

[16] J. N. Herder et al., "MINIX 3: A Highly Reliable, Self-Repairing Operating System," vol. 40, no. 3. ACM SIGOPS, 2006.

[17] A. Mancina et al., "Enhancing a Dependable Multiserver Operating System with Temporal Protection via Resource Reservations," in *Conf. on Real-Time and Network Systems*, 2008.

[18] J. N. Herder et al., "Countering IPC Threats in Multiserver Operating Systems," in *Pacific Rim Symp. on Dependable Computing*, 2008.

[19] ——, "Fault Isolation for Device Drivers," in *Conf. on Dependable Systems and Networks*, 2009.

[20] ——, "Dealing with Driver Failures in the Storage Stack," in *Latin-American Symp. on Dependable Computing*, 2009.

[21] V. Jacobson, "Congestion Avoidance and Control," in *ACM SIGCOMM*, 1988.

[22] S. Lee and H. Youn, "Dynamic Window-based Adaptive Fault Monitoring for Ubiquitous Computing Systems," in *Pacific Rim Symposium on Dependable Computing*, 2005.

[23] W. Chen, S. Toueg, and M. K. Aguilera, "On the Quality of Service of Failure Detectors," *IEEE Trans. on Computers*, vol. 51, no. 5, 2002.