A statistical anomaly-based algorithm for on-line fault detection in complex software critical systems

Antonio Bovenzi[1], Francesco Brancati[2], Stefano Russo[1] and Andrea Bondavalli[2]

[1] Dipartimento di Informatica e Sistemistica (DIS), Università degli Studi di Napoli
"Federico II", Napoli, Italy
{antonio.bovenzi, sterusso}@unina.it

[2] Dipartimento di Sistemi e Informatica (DSI), Università degli Studi di Firenze, Italy
{Francesco.brancati, bondavalli}@unifi.it

Abstract. The next generation of software systems in Large-scale Complex Critical Infrastructures (LCCIs) requires efficient runtime management and reconfiguration strategies, and the ability to take decisions on the basis of current and past behavior of the system. In this paper we propose an anomaly-based approach for the detection of online faults, which is able to (i) cope with highly variable and non-stationary environment and to (ii) work without any initial training phase. The novel algorithm is based on Statistical Predictor and Safety Margin (SPS), which was initially developed to estimate the uncertainty in time synchronization mechanisms.

The SPS anomaly detection algorithm has been experimented on a case study from the Air Traffic Management (ATM) domain. Results have been compared with an algorithm, which adopts static thresholds, in the same scenarios [5]. Experimental results show limitations of static thresholds in highly variable scenarios, and the ability of SPS to fulfill the expectations.

Keywords: Anomaly detection; SPS; on-line software fault diagnosis.

1 Introduction

Large scale Complex Critical Infrastructures (LCCI), such as transport infrastructures (e.g., the novel European Air Traffic Management federated system[1]) or power grids, play a key role into several fundamental human activities. It is easy to think about their economic and social impact: the consequences of an outage can be catastrophic in terms of efficiency, economical losses, consumer dissatisfaction, and even indirect harm to people.

LCCIs are the result of the integration of heterogeneous stand-alone subsystems and their scale is strongly increasing, due to deregulation and the development of "mixed market infrastructures" [10] and technological improvement. Such interconnection requires not only designing a way to interconnect heterogeneous systems, but also imposes that legacy systems have to operate beyond the original design parameters [9].

[1] http://www.sesarju.eu/

These systems rely on the efficacy of services, such as system management, replication, load balancing and group communication, which require suitable algorithms able to take runtime decisions on the basis of actual and past behavior of the system. All these characteristics exacerbate the complexity of the infrastructure, making crucial the designing of intelligent on-line monitoring and detection mechanisms to infer (i) if the whole system is performing well and, if not, (ii) how to face with possible failures.

Huge amount of data, coming from different probes spread over the system, need to be analyzed in order to reveal that something is not working properly. In other words, it must be possible to identify the case where anomalies occurred in the system. With the term anomaly we refer to changes in the variable characterizing the behavior of the system caused by specific and non-random factors [11], e.g., overload, the activation of faults, malicious attacks, etc.

It is an interesting open issue to detect relevant anomalies in systems that exhibit variable and non-stationary behavior, and may be affected by perturbations. Moreover, the detection problem is exacerbated when the anomaly detector has to support timely decisions based on online instead of offline analysis.

Detection systems usually assume worst-case thresholds to allow distinguishing between nominal behaviors and anomalies [5][6]. However these thresholds are typically tuned in a preliminary training phase and cannot fit all dynamically changing situations in which the system could evolve. For instance, these thresholds may depend on the application requirements, on system operational parameters and on the current environment, and usually there are not a-priori fixed for the entire system, and for all the system life-cycle; therefore, detectors can take advantage from the possibility to adapt the expected thresholds online, e.g., because of operational conditions modifications.

This work proposes an anomaly-based approach for software fault detection in complex critical system, exploiting statistical analysis on data gathered at the Operating System level. The proposed detector is able (i) to cope with variable and non-stationary behavior, (ii) to perform an online (instead of offline) analysis and (iii) to work without any initial training phase.

The detector receives data coming from a monitoring infrastructure, described in [5], and it exploits the Operating System tracking mechanisms to collect different kinds of information (e.g., syscall errors, signals, scheduling time of processes), which reveal to be useful for detection.

The statistical analysis is performed by means of a recent algorithm, i.e., Statistical Predictor and Safety Margin (SPS), which was initially designed to estimate the synchronization uncertainty of a software clock [7]. Our intuition was that such algorithm could be exploited to detect anomalies, due to software faults activation, in high variable context. Such intuition is confirmed by our experimental results, which encourage further research.

The detector performance have been evaluated by means of an experimental campaign (see section 5) on a case study coming from the ATM (Air Traffic Management), namely the SWIM-BOX®, which is a prototype developed at SESM[2] to allow the cooperation and the interoperability of future ATM systems. Results have

[2] SESM s.c.a.r.l., a Finmeccanica company. http://www.sesm.it/

been compared with an algorithm, which adopts worst-case thresholds (in the following we refer to this algorithm as Static Thresholds Algorithm), in order to explore possible improvements adopting this algorithm in the same situations [5]. The experimental campaign involved a complete and sound testing activity, which explores the performance of both the algorithms using a large set of possible configurations. In particular, we execute fault injection experiments to accelerate the failure related data collection, which allows labeling the relevant anomalies (namely, those due to fault activation).

To evaluate performance we relied on the metrics for failure prediction used by Malek in [1] and the Quality of Service (QoS) metrics for failure detectors provided by Chen, Tuoeg and Aguilera in [12], furthermore an analysis of which class of metrics best describes this class of algorithm has been provided.

The paper is organized as follows. Section 2 gives a brief description of the Detection Framework, section 3 introduces the main steps to adapt the SPS algorithm to this context, section 4 gives a survey of the most used metrics in literature, section 5 describes the experimental campaign and section 6 analyzes the obtained results. Finally conclusion and future work are in section 7.

2 The Detection Framework

The Detection Framework was proposed in [5]. The Authors propose an approach based on indirectly (and locally) inferring the health of the monitored component by observing its behavior and interactions with the external environment. The basic idea is to shift the observation perspective and to leverage OS support to detect application failures.

2.1 Assumptions

System model. The aim is to detect failures in complex Off-The-Shelf (OTS) based safety critical software systems. These are often distributed on several nodes, which communicate through a networking infrastructure. However, we focus on a single node of the system to perform failure detection and we do not care of system topology. In this context, failure detection is performed at process (thread) level.

Failure model. According to the failure classification proposed in [1] we focus on (i) crash failures and (ii) hang failures, i.e., failures which cause the delivered service to be halted and the external state of the service to be constant. In our context, a service is crashed when the process terminates unexpectedly (e.g., due to run-time exceptions). Thus we refer to systems whose failures are to an acceptable extent halting failures only, namely fail-halt (or fail-stop) system. For a more detailed description of this class of failures we refer to [5].

2.2 The detection approach

The detection framework is based on the combination of several OS level monitors. As suggested by intuition, combining multiple alarms coming from several sources allows revealing a higher number of failures, as well as to gain a better accuracy, if compared to detector without combination.

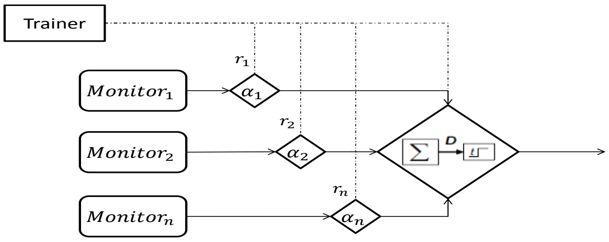

Figure 1. The detection framework architecture.

As depicted in Figure 1, multiple monitors M_i keep track of OS data related to a given process (thread). Each monitor is followed by an alarm generator α_i. If the monitored value n does not belong to the specified range $\boldsymbol{r}_i = [r_i^-, r_i^+]$, an alarm is triggered. Actual thresholds have to be preliminary tuned for each monitored variable, during a so-called training phase (see the Trainer block in Figure 1). Training is performed by means of an initial profiling phase analyzing both normal and faulty runs (namely, runs when a failure occurs). As previously stated, alarms, triggered by alarm generators, are combined in order to improve detection quality.

The detector D performs the overall detection by means of a simple heuristic defined as the weighted sum of single alarms, where weights are defined after the training phase. A failure is finally detected if the output of D exceeds a given threshold t tuned during the training phase too.

2.3 Limitations of the static thresholds

As described in previous section static threshold algorithms perform well enough if the environmental conditions in which the system operates are similar to the training phase. Performance goes worse if the operational conditions of system differ from those of the training phase since the evaluated thresholds may no longer be able to model the nominal behavior. This last situations is no far from real scenarios if we consider Large scale Complex Critical Infrastructures (LCCI) as possible application field, in which detection algorithms have to deal with highly variable scenarios, in which the whole system evolves in time and space dimension, alternating periods of heavy workloads, which involved high number of nodes and heterogeneous type of service requests, with periods of low computational activities. In this type of environment, training phases need to be performed periodically in order to keep tuned the algorithm. Since this kind of dynamical environment changes are often very hard to predict an algorithm that computes adaptive, instead of static, thresholds could overcome such limitations because, as shown in Section 6, it does not depend from an initial training phase.

3 Using SPS algorithm to estimate adaptive thresholds

In this section we first give an overview of SPS algorithm, by discussing its assumptions and by showing how it can be used to provide adaptive thresholds to the detector, then we illustrate the fundamental differences between static and adaptive thresholds.

3.1 SPS-based detection algorithm

The SPS algorithm was initially designed to compute uncertainty interval at a time t within a given coverage, namely the probability that the next value of the time series will be inside the uncertainty interval. This algorithm can be adapted to compute adaptive bounds for anomaly detection activities with minor changes. As shown in [7], the uncertainty computed by SPS algorithm consists in a combination of left and right bounds. These bounds are computed starting from three quantities: (i) the *last value of the series*, (ii) the output of a *predictor* function and (iii) the output of a *safety margin* function. The output of the SPS at $t \geq t_0$ is constituted by the two values:

$$U_r(t)=\max(0, \widetilde{\Theta}(t_0))+P_r(t)+SM_r(t_0) \quad U_l(t)=\min(0, \widetilde{\Theta}(t_0))+P_l(t)+SM_l(t_0)$$

Where $\widetilde{\Theta}(t_0)$ is the last value of the time series (i.e., the estimated offset for the time synchronization environment). In computing uncertainty in time synchronization the requirements state that the uncertainty interval must contain the global time, so we distinguish left from right uncertainty by considering the offset only if it is negative in the former, positive in the latter.

We adapted the computation of the bounds assuming symmetrical values for the predictor and the safety margin functions ($P_r(t) = -P_l(t)$ and $SM_r(t) = -SM_l(t)$) and computing the adaptive bounds as:

$$T_u(t) = x(t_0) + P(t)+SM(t_0) \qquad T_l(t)=x(t_0)-P(t)-SM(t_0)$$

Where $T_u(t)$ and $T_l(t)$ are the upper and the lower bounds at time t, and $x(t_0)$ is the last value of the series.

The *predictor* function provides an estimation of the behavior of the time series. The *safety margin* function aims at compensating possible errors in the prediction and/or in the measurement of the actual value of the time series. The *safety margin* is computed at t_0 and it is updated only when new measurements arrive. We refer to [7] for technical details about the predictor and the safety margin functions.

The set-up parameters used by SPS are: four probabilities $p_{ds}, p_{dv}, p_{os}, p_{ov}$, that can be combined in one single parameter c (coverage of the algorithm) and the two different values for memory depth $M_1 = M_2 = m$. The performance achieved by SPS depends on these parameters [7].

Computational cost of the SPS algorithm depends on the computation of a population-weighted variance. Since variance is computed using sums of the elements, the computational cost of the algorithm is linear with the number of samples. If we use accumulators to store the value of the sums in memory, the computational cost of SPS becomes constant. This last solution is obviously preferred when we want to use the algorithm at runtime considering a large set of samples.

3.2 SPS assumptions

The measurements provided by monitors are received at regular interval of time. Let be k the number of different monitored variables.

Def. 3.1 (timeseries). A time series $T_i = x_{i1}, \dots, x_{ij}$ is an ordered set of j real-valued variables.

Def. 3.2 (anomaly). With respect to a monitored time series T_i, an **anomaly** is a change in the characteristics of T_i, caused by specific and non-random factors.

The continuous stream of data points $x \in R^k$ constitutes the collection of measurements. These measurements correspond to certain physical events in the event space S, which we assume can be divided into two subspaces corresponding to normal events (S_N) and anomalous events (S_A).

In order to apply SPS algorithm to time series T_i, we make the following assumptions.

Random walk model. We assume that the monitored process behavior can be modeled as a random walk, with or without drift. Namely, the variability of the process is the result of the cumulative effect of small but unavoidable constant and casual factors.

Interleaved behavior. We assume that the environment alternates *stable periods*, during which the monitored process has some stability properties (i.e., it is under control), with *transient periods* (smaller compared with the stable), during which a variation of environmental or system condition occurs (due to workload, new configuration) involving a change in the characteristics of the monitored process. This assumption is supported by the results of many recent works [13].

Transient period changing. We assume that, during the transient period, changes in the monitored processes behavior consist in continuous increments or decrements with respect to previous values. These changes are due to some specific factors, which are not casual, such as: a modification of system structural parameters (e.g., the number of active nodes), overloading conditions (e.g., due to a burst of requests), the activation of a residual fault leading to system failures (e.g., crashes, hangs).

It is worth noting that, the proposed detection approach does not make any assumptions about the stationarity of the monitored variables. Namely, if the statistical properties of the monitored process (e.g., mean and variance) change over time, the detection of anomalies is still possible. Relaxing the stationary hypothesis makes the detector more suitable for real variable contexts with respect to the case in which these properties are statically derived by means of preliminary profiling phase.

3.3 The Detector equipped with SPS

The detector can be easily equipped with SPS modifying the alarm generator component, (i.e., α_i in Figure 1). SPS continuously processes data received from the associated monitor, thus it will be in charge to provide adaptive thresholds to each α_i. In this way the training phase, which in the previous version of the detector was necessary to compute static thresholds and to find weights for each monitor, is totally avoided.

3.4 Comparison between adaptive and static thresholds algorithm

Figure 2 shows adaptive thresholds computed by SPS compared with Static thresholds for the same monitored variable (total number of timeouts expired for scheduling of processes).

SPS thresholds signals the failure (at about 160 *s*), while, as we can observe in the left part of Figure 2, the monitored value is very often above the upper static threshold, producing a lot of False Positive.

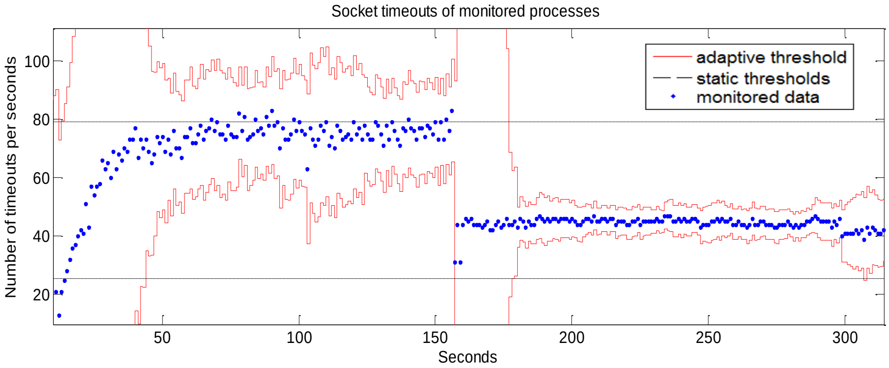

Figure 2. Static thresholds and Adaptive thresholds on monitored data.

4 Metrics for performance evaluation

In order to provide a fair and sound comparison between SPS and Static Thresholds Algorithm we analyze criteria that should be used to characterize on-line detectors performance for our target applications. We first summarize some of the most used metrics in literature then we must specify, which metrics are the most representative in our scenario and why.

Roughly speaking the goal of an on-line failure detector is i) to reveal all the occurring failures, ii) to reveal them timely and ii) not to trigger false alarms.

To ease the description of metrics and to better understand their meaning we introduce some definitions:

- *True Positive* (TP)*:* if a failure occurs and the detector triggers an alarm;
- *False Positive* (FP): if no failure occurs and an alarm is given;
- *True Negative* (TN): if no real failure occurs and no alarm is raised;
- *False Negative* (FN): if the algorithm fails to detect an occurring failure.

Clearly many TPs and TNs are good, while the vice versa for FPs and FNs.

Metrics coming from diagnosis literature are usually used to compare the performance of detectors [9]. For instance *coverage* measures the detector ability to reveal a failure, given that a failure really occurs; *accuracy* is related to mistakes that a failure detector can make. Coverage can be measured as the number of detected failures divided by the overall number of failures, while for accuracy there are different metrics.

Basseville et al. [3] consider the *mean delay for detection* (MDD) and the *mean time between false alarms* (MTBFA) as the two key criteria for on-line detection algorithms. Analysis are based on finding algorithms that minimize the mean delay for a given mean time between false alarms and on other indexes derived from these criteria.

In [2] metrics borrowed from information retrieval research are used, namely *precision* and *recall*. In their context recall measures the ratio of failures that are correctly predicted, i.e., *TP/(TP+FN)*, while precision measures the portion of the predicted events, which are real failure, i.e., *TP/(TP+FP)*. Thus perfect recall (recall=1) means that all failures are detected and perfect precision (precision=1) means that there are no false positives. A convenient way of taking into account precision and recall at the same time is by using *F-measure*, which is the harmonic mean of the two quantities. Since in diagnosis the ratio of failures correctly detected (recall) is also called coverage, in the following we refer to it as coverage.

However using solely precision and coverage is not a good choice because they do not account for true negatives, and since failures are rare events we need to evaluate the detector mistake rate when no failure occurs. Hence, in combination with precision and coverage, one can use *False Positive Rate* (FPR), which is defined as the ratio of incorrectly detected failures to the number of all non-failures, thus *FP/(FP+TN)*. Fixing Precision and Coverage, the smaller the false positive rate, the better. Another metric is *Accuracy* [2], which is defined as: the ratio of all correct decisions to the total number of decisions that have been taken, i.e., (*TP*+*TN*)/(*TP*+*TN*+*FP*+*FN*).

Chen, Toueg and Aguilera [12] propose three primary metrics to evaluate detectors quality, in particular their accuracy. The first one is *Detection Time* (DT), which, informally, accounts for the promptness of the detector. The second one is the *Mistake Recurrence Time* (T_{MR}), which accounts for time elapsed between two consecutive erroneous transitions from Normal to Failure. Finally they define *Mistake Duration* (T_M), which is related to the time that detector takes to correct the mistake. Other metrics can be simply derived from the previous one. For instance, *Average Mistake Rate* (λ_M), represents the number of erroneous decisions in the time unit; *Good period duration* (T_G) measures the length of period during which the detector does not trigger a false alarm; *Query accuracy probability* (P_A) is the probability that the failure detector's output is correct at a random time.

Bearing in mind classes of our target applications we believe that, when dealing with long running and safety critical systems, mistake duration (and thus T_G) is less appropriate than Coverage, accuracy and λ_M, since just an alarm may be sufficient to trigger the needed actions (e.g., put the system in a safe state). Coverage is essential because if the detector does not reveal a failure, then more severe (and potentially catastrophic) consequences may happen. Accuracy and λ_M are useful to take into account false positives because each failure detector mistake may result in costly actions (such as shut down, reboot, etc.).

The query accuracy probability is not sufficient to fully describe the accuracy of a failure detector, in fact, as discussed in [12], for applications in which every mistake causes a costly interrupt the mistake rate is an important accuracy metric too.

We point out the differences between Accuracy, defined in [2], and P_A defined in [12]. Since we consider a fail stop model, *TP*<<*TN*, so if *FP*<<*TN*, then Accuracy≈1. For these reasons we consider P_A as more representative than Accuracy to compare SPS algorithm with Static Thresholds Algorithm.

Finally we introduced an additional parameter for performance evaluation, the time-to-detection d. This parameter represents the maximum delay to detect a failure. Thus, considering T_f the time when a failure occurs, if an alarm is raised after $T_f + d$ we do not account it as a TP. To be sure that a true positive is effectively related to the failure we set $d \leq m$ (where m is the memory depth of SPS algorithm).

Table 1. Metrics for performance evaluation of failure detectors.

Metric	*Formula*	*Metric*	*Formula*
Coverage (C)	$TP/(TP+FN)$	**Accuracy-Coverage TradeOff**	$\mathrm{C} \cdot \mathrm{A}$
Precision (P)	$TP/(TP+FP)$	**MTBFA** (Mean Time Between Mistakes)	$E(T_{MR})$
F-Measure	$(2 \cdot P \cdot C)/(P+C)$	**MDD** (Mean Delay for Detection)	$E(T_D)$
FPR (False Positive Rate)	$FP/(FP+TN)$	**λ M** (Average Mistake Rate)	$1/E(T_{MR})$
Accuracy (A)	$\frac{TP+TN}{TP+FP+TN+FN}$	**P_A** (Query accuracy probability)	$\frac{E(T_{MR}-T_M)}{E(T_{MR})}$

5 Experimental campaign

We performed an experimental campaign with the aim of comparing performance of two Detection frameworks using both Static Thresholds and SPS algorithm as alarm generators.

The testing activity was performed analyzing a large amount of data monitored in a real and complex case application, namely the SWIM-BOX®. The application is made of several OTS, e.g., OS, the application Server (JBoss) and the data distribution middleware (OpenSplice). We executed tests under different workloads and faultloads, using fault injection technique described in [4], in order to accelerate the failure related data collection. The monitored data of each test was loaded into a well-structured data-repository following an OLAP approach [13]. The two algorithms are then applied and evaluated in a post-processing phase.

5.1 Case study description

The SWIM-BOX® is actually a pilot prototype, which has been implemented in the framework of the SWIM SUIT FP6 European project[3], to support global

[3] http://www.swim-suit.aero/swimsuit/

interoperability for the novel Air Traffic Management (ATM) systems. It is a complex OTS-based application, which offers several facilities to SWIM-BOX users: synchronous/asynchronous communication pattern (i.e. request/reply, publish/subscribe), security services (e.g., authentication, authorization, encryption) and distributed and transactional data storage.

The case study scenario consists of two legacy entities, named the Contributor and the Manager, which collaborate by means of the SWIM-BOX to manage Flight Data Plan.

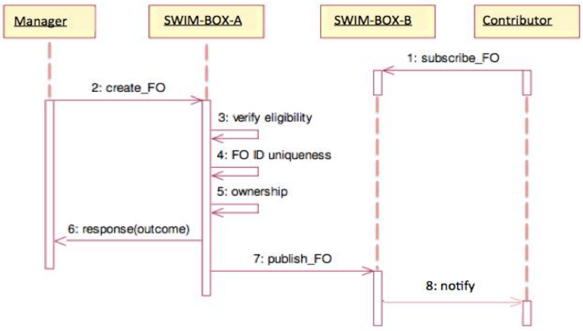

Figure 3. Simplified interaction scenario.

Figure 3 describes an example of the interaction between the legacy systems. The Contributor acts as the subscriber, waiting for Flight Object updates (i.e., an entities including several data related to a flight) 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 creations and updates) at a variable rate (20 to 300 operations/min in the experiments), as well as of (ii) distributing data over the network. Once all the operations have been completed the Contributor requires unsubscribing. Two legacy entities Manager and Contributor, represent the system under test.

5.2 The experimental activity

Detection Framework analyzes several variables by means of the monitoring infrastructure described in [5]. Monitors are based on OS instrumentation facilities, which allow implanting probes into the kernel and register corresponding probe handlers. Probes are breakpoints inserted dynamically into the kernel module avoiding recompiling and rebooting. When a breakpoint is hit, a handler routine is launched to register an event with the needed information (e.g., input parameters or return values of called functions).

The events collected during this experimental campaign are shown in Table 2.

Monitors are associated to probes and provide measurements by aggregating the number of events recorded by a probe in a given time period. Other monitors, measure the disk/network throughput, by summing bytes read or write, respectively on the disk and on the network interfaces. The total number of types of monitor is 18.

This infrastructure is completely configurable such that one can choose (i) processes and network interfaces to observe, (ii) timeouts associated to a particular probe (when needed) and (iii) the sample period of monitors (i.e., the interval of time between two consecutive measurements).

Table 2. Probe description.

Probe	Trigger condition for events registration
System call error code	An error code is returned
Time scheduling of process	Timeout exceeded since the process is preempted
Signal	A signal is received
Process/Thread creation/termination	Creation or Termination of a Process (Thread)
I/O on Disk	Timeout exceeded since last disk read/write
I/O on Socket	Timeout exceeded since last socket read/write
Holding time for Mutex/Semaphore	Timeout exceeded for mutex/semaphore possession
Waiting time for Mutex/Semaphore acquisition	Timeout exceeded for mutex/semaphore acquisition
Disk Throughput	A byte is read/write
Network Throughput	A byte is send/received

We evaluate the overhead of the monitoring infrastructure by measuring the execution time of our application when the monitors are turned off and when they are enabled. Overhead results almost negligible (about 3%).

Table 3. Experimental activity dimensions

Dimensions	Description
Target System	Characteristics of the Target System,(CPU, RAM, disk speed, …)
Events	Monitored events
Run	Information on the executed run (start time, end time, …)
Scenario	In this campaign we considered only the scenario described in 5.1: Two Entity, one Manager and one Contributor.
Workload	Adopted Workloads differs from message rate, message burst rate and message per burst.
Faultload	Several faults injected (one per injection) by means of code mutation technique [4].

Monitored data were stored in an online data repository following an OLAP approach [13]. Dimensions and description of the OLAP repository are shown in Table 3.

We considered a subset of the mathematical combinations of all dimensions, and performed 17 faulty runs and 19 nominal runs. After the execution of the runs both algorithms were applied in a post-processing phase, varying several configurations. Combinations of runs and considered configurations give 6300 different sets of data (see sec.5.3).

5.3 The post processing phase

After the execution of all the runs both Algorithms are applied in a post-processing phase, varying several configurations. The evaluation of metrics is carried out by applying algorithms to dataset, which have not been used for parameters tuning. For these reasons the dataset is divided into a training set and a validation set, which consist respectively in 6 and 30 runs.

Post processing SPS. SPS algorithm was applied varying the following parameter: coverage c: $\{0.9, 0.99, 0.9999\}$, memory depth m (in terms of number of data considered in the statistics): $\{10, 20\}$, time for detection d: $\{m, \left\lfloor\frac{m}{2}\right\rfloor, \left\lfloor\frac{m}{3}\right\rfloor\}$. To be sure that a true positive alarm is effectively related to the failure we set $d \leq m$. Combining these tree dimension we obtains 18 different configurations for SPS.

Post processing Static Thresholds. Static Thresholds algorithm was applied by varying (i) method for the evaluation of thresholds $\boldsymbol{r_i}$ for each monitor M_i and the time for detection d: $\{3, 6, 10, 20\}$. In particular we estimate the thresholds on the training set using 3 different methods: $\boldsymbol{r_i} = \{[min, max], [\mu - \sigma, \mu + \sigma], [\mu - 2\sigma, \mu + 2\sigma]\}$. The total number of configurations is 12.

For Both SPS and Static Threshold we consider 5 values for parameter t in the global detector, so we have a total of 150 different configurations. All 90 SPS configurations was post processed for the validation set (30 runs), while the 60 Static Algorithm configurations were post processed first for validation set by carrying out the training phase of algorithm, and then to the validation set without the training phase, for a total of 6300 sets of analyzed data.

6 Results Analysis

We pointed out 3 different scenarios to be compared: SPS Algorithm; Static Thresholds Algorithm with a training phase; Static Thresholds Algorithm without a training phase.

Since it could be very difficult and confusing but also not useful to compare the two algorithms for all possible configurations and all metrics we decided to compare only the best configuration for the two algorithms. To select the best configuration among all the metrics described in section 4, we introduced a synthetic measure $Synthesis = \quad mean(|aPA| + |Coverage| + |Accuracy| + (1 - (|aTM|)) + (1 - |aMR|))$ that takes into account most relevant metrics that are not correlated together. Synthesis allows weighting the metrics according to the type of system at hand. For our purposes we considers metrics with the same weight.

Configurations with best Synthesis measure were $c = 0.99, m = 20, d = 20,$ t=0.4 for SPS, $d = 3, t = 0.5$ for Static Thresholds with the training phase and $d = 3,\ t = 0.3$ for Static thresholds without a training phase. It's noteworthy that the absence of the training phase for Static algorithm involves setting all weight to 1 in the global detector.

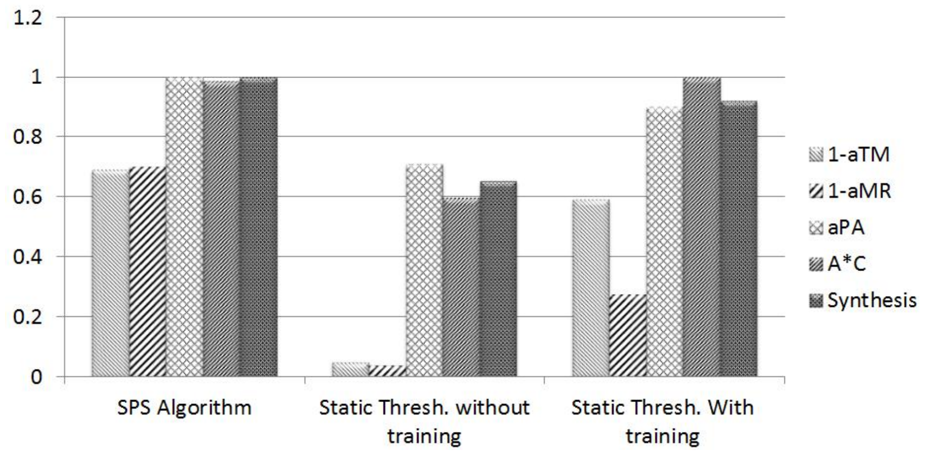

Figure 4 Compared experimental results

The following metrics: average Mistake Duration (aTM), average Mistake Rates (aMR), average Query Accuracy (aPA), Accuracy-Coverage tradeoff (A*C), and the Synthesis metric has been normalized in Figure 4 to ease the comparison, while the absolute values are showed in Table 4.

Table 4. Experimental results

	aTM (sec)	aMR	aPA	A*C	Synthesis
SPS Algorithm	**3.611111**	**0.009804**	**0.95098**	**0.96004**	**0.914551**
Static T. without training	**11.63951**	**0.032724**	**0.674494**	**0.582356**	**0.59768**
Static T. with training	**4.75**	**0.023715**	**0.85584**	**0.974436**	**0.842296**

The weakness of static thresholds algorithm described in section 2.3 has been proved by the poor results achieved without the training phase. Moreover, SPS-based detection algorithm shows better results, proving that adaptive thresholds give an effective gain in this class of detectors.

7 Conclusion and future work

This work proposes an anomaly-based approach for software fault detection in complex critical systems, exploiting statistical analysis on data gathered at the Operating System level. The proposed approach relies on the SPS algorithm, which was initially designed to compute uncertainty in clock synchronization [7]. Performance of this algorithm has been evaluated by means of an experimental campaign and compared with a static threshold algorithm proposed in [5]. The comparison has been made, first exploring the existing metrics in literature [2],[12] and then applying the most representative to both algorithms. Experimental results confirm the limitations of static threshold algorithms in highly variable scenarios, where the operational conditions of system differ from those of the training phase. Moreover results show that, compared to static threshold based algorithms, the SPS algorithm is able (i) to cope with variable and non-stationary behavior, (ii) to work without any initial training phase, and (iii) to perform better even when a training phase is applied, in terms of coverage, query accuracy probability, mistake rate and mistake duration.

As future work we will implement this algorithm to test and validate it when executed online and compare it with other techniques used in failure detection field, such as machine learning algorithm, and ARIMA models [3].

Acknowledgment. This work has been partially supported by the Italian Ministry for Education, University, and Research (MIUR) in the framework of the Project of National Research Interest (PRIN) “DOTS-LCCI: Dependable Off-The-Shelf based middleware systems for Large-scale Complex Critical Infrastructures” (dots-lcci.prin.dis.unina.it).2008. It is also in the context the ”Iniziativa Software” Project, an Italian Research project which involves Finmeccanica company and several Italian universities (www.iniziativasoftware.it)

References

1. A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. Dependable Secure Computing, 2004.
2. F. Salfner, M. Lenk, M. Malek. A survey of online failure prediction methods. ACM Computing Surveys (CSUR), 2010.
3. M .Basseville, I.V. Nikiforov. Detection of abrupt changes: theory and application. Prentice-Hall, Inc. 1993.
4. R. Natella, D. Cotroneo, Emulation of transient software faults for dependability assessment: A case study, in Proceedings of the Eighth European Dependable Computing Conference (EDCC), 2010.
5. G. Carrozza, M. Cinque, D. Cotroneo, R. Natella. Operating System Suppor t to Detect Application Hangs. International Workshop on Verification and Evaluation of Computer and Communication Systems, VECoS, 2008.
6. I. Irrera, J. Duraes, M. Vieira, H. Madeira. Towards Identifying the Best Variables for Failure Prediction Using Injection of Realistic Software Faults. Pacific Rim International Symposium on Dependable Computing, IEEE, 2010.
7. A. Brancati, A. Bondavalli, A. Ceccarelli. Safe estimation of time uncertainty of local clocks. International Symposium on Precision Clock Synchronization for Measurement, Control and Communication, 2009.
8. F. Salfner. Event-based failure prediction: an extended hidden Markov model approach, Dissertation.de, Berlin, 2008
9. A. Daidone, "Critical infrastructures: a conceptual framework for diagnosis, some applications and their quantitative analysis", PhD thesis, Università degli studi di Firenze, December 2009.
10. C. Johnson, M. Malek, "Progress achieved in the research area of Critical Information Infrastructure Protection by the IST-FP6 Projects CRUTIAL, IRRIIS and GRID", Technical report, EU Report, March 2007.
11. D.C. Montgomery. Controllo statistic della qualità. McGraw-Hill italia, 1^{a}ed. 2000.
12. Chen, Wei and Toueg, Sam and Aguilera, Marcos Kawazoe. On the Quality of Service of Failure Detectors. Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8), 2000.
13. A. Casimiro, P. Lollini, M. Dixit, A. Bondavalli, P. Verissimo. A framework for dependable QoS adaptation in probabilistic environments. Proceedings of the 2008 ACM symposium on Applied computing, 2008.
14. H. Madeira, J. Costa, M. Vieira, The OLAP and data warehousing approaches for analysis and sharing of results from dependability evaluation experiments, Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on, pp. 86- 91, 22-25 June 2003