Context-Awareness to improve Anomaly Detection in Dynamic Service Oriented Architectures

Tommaso Zoppi, Andrea Ceccarelli, Andrea Bondavalli
University of Florence, Viale Morgagni 65, Firenze, Italy
{tommaso.zoppi, andrea.ceccarelli, bondavalli}@unifi.it

Abstract. Revealing anomalies to support error detection in software-intensive systems is a promising approach when traditional detection mechanisms are considered inadequate or not applicable. The core of anomaly detection lies in the definition of the expected behavior of the observed system. Unfortunately, the behavior of complex and dynamic systems is particularly difficult to understand. To improve the accuracy of anomaly detection in such systems, in this paper we present a context-aware anomaly detection framework which acquires information on the running services to calibrate the anomaly detection. To cope with system dynamicity, our framework avoids instrumenting probes into the application layer of the observed system monitoring multiple underlying layers instead. Experimental evaluation shows that the detection accuracy is increased considerably through context-awareness and multiple layers monitoring. Results are compared to state-of-the-art anomaly detectors exercised in demanding more static contexts.

Keywords: Anomaly Detection · Monitoring · Service Oriented Architecture · SOA · Context Aware · Multi-Layer

1 Introduction

Complex software-intensive systems include several different components, software layers and services. Often, these systems are characterized by a dynamic behavior related to changes in their services, connections or components themselves. In particular, *Service-Oriented Architectures* (SOAs) may aggregate proprietary as well as *Off-The-Shelf* (OTS) services, hiding their implementation details. It is a matter of fact that SOA dynamicity and information hiding obstacle monitoring solutions that directly observe the SOA services [19]. This collides with the increasing interest in using these systems for (safety) critical applications, and raises a call for adequate solutions to monitoring and error detection [1], [21].

Anomaly detection aims to find patterns in monitored data that do not conform to the expected behavior [1]. Such patterns are changes in the trends of indicators such as memory usage or network data exchange characterizing the behavior of the system caused by specific and non-random factors. As an example, anomalies can be due to a system overload, adversarial intrusion attempts, malware activity or manifestation of

errors. Anomaly detection was proved [7] to be effective, highlighting anomalies and timely triggering reaction strategies to finally improve system safety or security.

Investigating dynamic contexts makes the definition of normal (and consequently anomalous) behavior a complex challenge: currently, there are no clear state-of-the-art answers on applying anomaly detection in highly dynamic contexts. Focusing on SOAs, anomaly detection usually requires a reconfiguration step to define the nominal behavior when services are updated, added or removed from the SOA [1]. It follows that anomaly detectors may be reconfigured frequently, reducing their effectiveness and with a negative impact on the SOA execution.

In this paper we present an anomaly detection framework that aims to tackle the challenges above. We tune the monitoring system to observe the underlying layers (e.g., operating system, middleware and network) instead of directly instrumenting the services with monitoring probes. This allows detecting anomalies due to errors or failures that manifest in the services without directly observing them [22]. Therefore, this *multi-layer* approach turned out very suitable to cope with dynamicity of complex systems, at the cost of a calibration time to reconfigure the parameters of the anomaly detector when changes of the components of the complex system are detected. This approach was previously proved effective on systems with reduced dynamicity respect to complex systems [14], while experimental results showed that a more accurate definition of the context was needed in highly dynamic systems [6] to improve detection accuracy. In this study we consider knowledge of basic information on the context - referred as *context-awareness* - that can be easily retrieved from integration modules of SOAs. This knowledge helps defining more precisely the expected behavior of the dynamic target system, resulting in more accurate definition of anomalies and, consequently, a more effective anomaly detection process. In fact, our multi-layer monitoring structure makes available a wide set of indicators, and the most relevant ones for anomaly detection purposes are identified depending on the current context. Consequently they are observed, with corresponding monitoring probes, building time series that are analyzed for anomaly detection purposes.

Summarizing, our main findings are: i) describing how context-awareness on the SOA services can be used to improve detection; ii) defining a methodology and the associated framework for anomaly detection in dynamic contexts using context-awareness; iii) structuring a multi-layer anomaly detection module observing *operating system*, *middleware* (*Java Virtual Machine*, JVM) and *network* layers, iv) assessing the whole solution on a case study, showing the obtained detection accuracy, which is presented using well-known metrics and v) compare our detection system with state-of-the-art [2], [3], [14] solutions exercised in less dynamic contexts.

The paper is structured as follows. Section 2 motivates the use of context-awareness, which is at the basis of our work. Section 3 describes the resulting anomaly detection framework and the devised methodology. Section 4 presents the experimental evaluation. State of the art on related approaches and comparison are explored in Section 5. Section 6 concludes the paper.

2 Learning from the past

This work stems from studies by the same authors [14], [6] who devised multi-layer anomaly detection [22] strategies to perform error detection using the *Statistical Predictor and Safety Margin* (SPS, [9]) algorithm. SPS is able to detect anomalies with-

out requiring offline training; this was proved to be very performing in less dynamic contexts [14], where the authors applied SPS to detect the activation of software faults in an *Air Traffic Management* (ATM) system. Observing only OS indicators, SPS allowed implementing an anomaly detector which performed error detection with high precision. Therefore we adapted this promising approach to work in a more dynamic context [6], where we instantiated the multi-layer anomaly detection strategy on the prototype of the Secure! [8] SOA. The results achieved showed that analysing such a dynamic system without adequate knowledge on its behavior reduces the efficiency of the whole solution. Despite the observed data stream was rapidly processed, we obtained a detection time - the time interval between the manifestation of the error and its detection - of 40 seconds with a high number of false positives and negatives.

We explain these outcomes as follows. SPS detects changes in a stream of observations identifying variations with respect to a *predicted* trend: when an observation does not comply with the predicted trend, an alert is raised. If the system has high dynamicity due to frequent changes or updates of the system components, or due to variations of user behavior or workload, such trend may be difficult to identify and thus predict. Consequently, our ability in identifying anomalies is affected because boundaries between normal and anomalous behavior cannot be defined properly.

2.1 Considering context-awareness

We previously highlighted the need of acquiring more information on the target system, still maintaining the main benefits of the abovementioned approach. Consequently, we investigate which information on SOA services we can obtain in absence of details on the services internals and without requiring user context (i.e., user profile, user location). In SOAs, the different services share common information through an *Enterprise Service Bus* (*ESB*, [15]) that is in charge of i) integrating and standardizing common functionalities, and ii) collecting data about the services. This means that static (e.g., services description available in *Service Level Agreements* - SLAs) or runtime (e.g., the time instant a service is requested or replies, or the expected resources usage) information can be retrieved using knowledge given by ESB. Consequently, having access to the ESB provides knowledge on the set of generic services running at any time *t*. We refer to this information as *context-awareness* of the considered SOA; note that we do not require information on the user context, contrary to what is typically done in the state-of-the-art on context-awareness [16], [17].

We can exploit this information to define more precisely the boundaries between normal and anomalous behavior of the SOA. For example, consider a user that invokes a *store file* service at time *t*. We can combine context-awareness with information on the usual behavior of the service, which here regards data transfer. Therefore, if the *store file* service is invoked at time *t*, we expect the exchange of data during almost the entire execution of the service. If we observe no data exchange, we can reveal that something anomalous is happening.

2.2 Enhancing Detection Capabilities

Collect services information. Let us start from the example of the *store file* service. Our objective is to characterize the normal behavior the service, building a *fingerprint*

of its usage. More in details, we need a description of the expected behavior of the service, meaning that we need to describe the *usual trend of the observed indicators* (examples of indicators are in Table 2 and Table 3) while the service is invoked. In such a way, we can understand if the current observation complies or not with the expectations. This information can be retrieved in a SOA by observing the ESB and producing a new service fingerprint when the addition, update or removal of a service is detected. In several cases it is also possible to obtain a static characterization of the services looking at their SLA, where each service is defined from its owner or developer for the final user. We remark that we do not consider any assumption about the services except their connection with the ESB: consequently, we can obtain services information from any kind of service running in the SOA platform.

Integrate information in the anomaly detector. Summarizing, information about the services can be obtained i) statically, looking at SLAs, ii) at runtime, invoking services for testing purposes or iii) combining both approaches. In this paper we explore the second approach, discussing this choice in Section 3.2. This information needs to be aggregated and maintained (e.g., in a database) together with the calculated statistical indexes (e.g., mean, median), whenever applicable, to support the anomaly detection solutions.

3 Description of the Anomaly Detection Framework

3.1 Architectural overview

In Figure 1 we depict a high level view of the framework. Starting from the upper left part of the figure, the framework can be described as follows. The user executes a *workload*, which is a sequence of invocations of SOA services hosted on the *Target Machine*. In this machine *probes* are running, observing the indicators coming from 3 different system layers: i) *OS*, ii) *middleware* and iii) *network*. These probes collect data, providing a *snapshot* of the target system composed by the observation of indi-

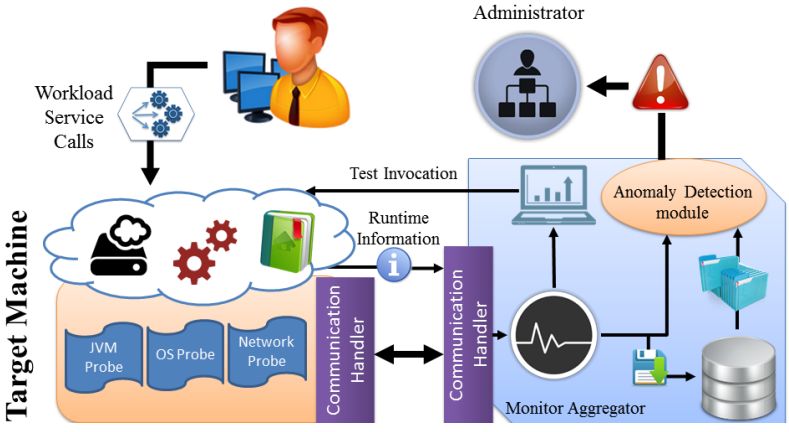

Fig 1. High-level view of the resulting multi-layer monitoring and anomaly detection framework

cators retrieved at a defined time instant. The probes forward the snapshot to the *communication handler*, which encapsulates and sends the snapshot to the *communication handler* of the *Detector Machine*. Data is analyzed on a separate machine, the Detector Machine (which includes a *Complex Event Processor* - CEP [18]). This allows i) not being intrusive on the Target Machine, and ii) connecting more Target Machines to the same Detector Machine (obviously the number of Target Machines is limited by the computational resources of the Detector Machine). The communication handler of the Detector Machine collects and sends these data to the *monitor aggregator*, which merges them with *runtime information* (e.g., list of service calls) obtained from the ESB. This allows storing context-awareness information in the database. Looking at *runtime information*, the monitor aggregator can detect changes in the SOA and notify the administrator that up-to-date services information is needed to appropriately tune the anomaly detector. The administrator is in charge of running tests (*test invocation*) to gather novel information on such services.

The snapshots collected when SOA is opened to users are sent to the *anomaly detection module*, which can query the database for services information and analyzes each observed snapshot to detect anomalies. If an anomaly is detected, the *system administrator,* which takes countermeasures and applies reaction strategies (which are outside from the scope of this work and will not be elaborated further), is notified.

3.2 Methodology to exercise the framework

The framework is instantiated specifying i) the *workload* we expect will be exercised on the target system, ii) the way (static/runtime) the administrator prefers to obtain services information described in Section 2.2, iii) the monitored layers on the Target Machine and the number of probes per layer, and iv) the *number of preliminary runs* necessary to devise the detection strategy elaborated in Section 3.3. The methodology is composed of two phases: *Training the Anomaly Detector* and *Runtime Execution*.

Training the Anomaly Detector. This phase is organized in 3 steps. In the first step, *services information* characterizing the fingerprint of the investigated services can be obtained statically (e.g., from SLA) or at runtime (through the *test invocation* in Figure 1). In our implementation, we chose this second option because it allows retrieving accurate information on the trend of the individual indicators; static information as SLA usually defines only general service characteristics and requirements.

In the second step, once services information is collected, *preliminary runs* using the expected workload are executed, and the retrieved data – a time series for each monitored indicator - are stored in the database. These data are complemented with data collected conducting error injection campaigns, where errors are injected in one of the SOA services, to witness the behavior of the Target Machine in such situations. The service in which errors are injected may be a custom service devoted exclusively to testing, allowing to modify its source code. This strategy can result particularly useful when performing injections into the services that compose the target system is not feasible (e.g., when services source code is not available as in OTS services).

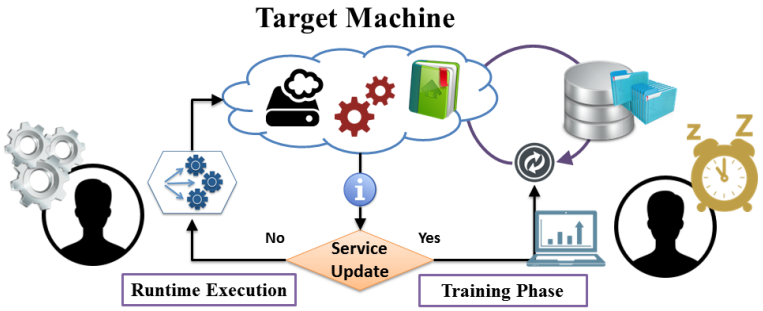

Fig. 2. Methodology: SOA hosted on target machine is available to users until a service update is detected from the *runtime information*. In that case, the training phase starts collecting services information and executing preliminary runs; the user needs to wait until it completes. Then the SOA is again available to users.

In the third step, services information and preliminary runs data are used by the anomaly detection module to tune its parameters, automatically choosing the configuration that maximizes detection efficiency for the current SOA (see Section 3.3).

We remark that we figured out two ways of obtaining the data in the first two steps: i) execute online tests before the user start working, or ii) copy the platform on another virtual machine and execute the tests on the spare machine in a controlled experimental environment. The first solution will force the user to wait until tests complete (see Figure 2), and consequently may reduce the availability of the SOA to the users. The second option requires additional resources to maintain and execute a copy of the Target Machine. In the rest of the paper we considered the first option: we collect context information through online tests before the SOA is opened to users. The induced delays on service delivery are measured in Section 4.1.

In some cases, to avoid downtime, it may be considered to postpone the execution of tests to low peak load periods such as at night. Obviously, delaying the execution of the tests (instead of running them immediately after services changes) implies that the anomaly detection module works with previous services information until the next training phase. This services information is now out of date: it is easy to note that this will negatively impact the accuracy of the anomaly detection module.

Runtime Execution. Once the anomaly detector is trained, the system is opened to users. *Monitor aggregator* merges each snapshot observed by the *probing system* with *runtime information*, and it sends them to the anomaly detection module. This module provides a numeric anomaly score (see Section 3.3). If the score reaches a specified threshold *alpha,* an anomaly alert is risen and the administrator is notified. If during this phase a service update is detected, a new *training phase* is scheduled and it will be executed depending on the policies defined by the administrator (see Figure 2).

3.3 Insights on the Anomaly Detection Module

Periodically (e.g., once per second), the *monitor aggregator* provides a *snapshot* of the observed system, composed of the quantities retrieved from the indicators. For each indicator, two quantities are sent: i) *value*: the current observation read by the probes, and ii) *diff*: the difference among the current and previous *value*.

This allows building a set of *anomaly checkers* as follows. An anomaly checker is assigned to the *value* or to the *diff* quantity of an indicator, i.e., two anomaly checkers can be created for each indicators. More precisely, each anomaly checker observes a specific time series made with the observations of the *value* or the *diff* quantity of a given indicator. Each anomaly checker decides if the quantity of the indicator is anomalous or normal following rules as described in the section below. The anomaly score for an observed snapshot is built combining the individual outcomes of the selected anomaly checkers; an anomaly is raised only if the *alpha* threshold is met.

Anomaly Checkers. For each indicator, we build three types of *anomaly checkers*:

- *Historical*: for a given indicator, this module compares the *value* or *diff* quantity with the expectations defined in services information. If this quantity is outside of the interval defined by *average* ± *standard deviation* in service information for that indicator, an anomaly is raised.
- *SPS*: for a given data series (*value*, *diff*) of an indicator, this module applies an instance of the SPS algorithm described in [9], [14].
- *Remote call*: this checker observes the response time and the HTTP response code for each service invocation. If the response code is not correct (e.g., HTTP Success *2xx*) or if the response time is not in the range of the acceptability interval defined by services information, an alert is raised.

For example, let us consider a set of 50 indicators. We obtain 201 possible anomaly checkers: 1 *remote call* checker and 200 anomaly checkers from the 50 indicators, organized in 4 anomaly checkers for each indicator (*historical* on *value/diff* data series, *SPS* on *value/diff* data series). The checkers to be used are selected during the training phase, analysing their scores for *specified metrics* (see below). As a result, the most performing checkers are selected i) choosing the *n* checkers with the highest score, or ii) considering checkers with a score greater than a threshold δ.

Specified Metrics. The anomaly checkers are evaluated during the training phase using measures based on indexes representing the correct detections - *true positives* (TP), *true negatives* (TN) - and the wrong ones i.e., missed detections (*false negatives*, FN) or false detections (*false positives*, FP). More complex measures based on the abovementioned ones are *precision*, *recall* and *F-Score(β)* [12]. Especially in the *F-Score(β)*, varying the parameter β it becomes possible to weight the *precision* w.r.t the *recall* (note that *F-Score(1)* is referred as *F-Measure*). Considering that we are targeting safety-critical systems, we prefer to reduce the amount of missed detections (FN), even at the cost of a higher rate of FP. For this reason, we selected as reference metric the *F-Score(2)*, which considers the recall more relevant than the precision: the F-Score(2) for each anomaly checker is computed, and checkers are selected accordingly (choosing the n best, or those whose F-Score(2) > δ).

4 Experimental Evaluation

We describe the experimental evaluation of the framework. To the purposes of the evaluation, we run an *automatic controller* that checks input data and manages the communications among the different modules of the Target Machine and Detector

Machine. This facilitates the automatic execution of the experimental campaigns without requiring user intervention. All data are available at [20].

Table 1. Execution time of tests and workload.

Workload		Single Test (s)	
Name	**Type**	**avg**	**std**
getCredentials	Serv. Test	8.88	0.60
createFolder	Serv. Test	10.71	0.69
addFiles	Serv. Test	10.04	2.01
addEventCalendar	Serv. Test	11.38	1.87
All Tests	Test All	92.98	7.37
All Services	Workload	86.04	4.87

4.1 Set-up of the Target and the Detector Machine

We conducted an experimental campaign using as target system one of the four virtual machines that host the Secure! crisis management system [8], which is built on the Liferay [13] portal, and uses Liferay services such as authentication mechanisms, file storage, calendar management. We identified 11 different services that can be invoked by the Secure! users. To simulate a set of possible user actions, we created the *All Services* workload calling a sequence of services, with a time interval of 1 second and overall lasting approximately 85 seconds (see Table 1).

Target and Detector Machines are virtual machines that run on a rack server with 3 Intel Xeon E5-2620@ 2.00 GHz processors. The Target Machine runs the Secure! prototype and it is instrumented with the probing system which reads 1 snapshot per second. Following our methodology in Section 3.2, after defining the *expected workload* we execute tests to collect *services information*. In Table 1 we compute the time required to obtain services information: we report the time needed to test a single service and all the 11 services (*All Tests*). The execution of these tests forces the users to wait until the SOA is available again. When the SOA has to be deployed for its first time, this only implies that deploy is delayed to wait for the tests completion. Once the SOA is deployed and available to users, it is expected that only few services will be updated each time, requiring only specific tests and consequently only short periods of unavailability. Consequently, except for the time needed for the initial test of all the services, the framework scales well also with a wider pool of services running on the SOA.

Regarding the most relevant anomaly checkers, we set $n = 20$, meaning that the 20 best anomaly checkers are selected following the *F-Score(2)* metric. Finally, we set *alpha* = 50%, meaning that an alert is raised if at least half of the anomaly checkers detect an anomaly for the considered snapshot. We want to point out that we considered a basic setup for the monitored indicators, the best checkers and the *alpha* parameter. A more detailed sensitivity analysis exploring all the possible settings will be performed as future work targeting the identification of the most performing setup of these parameters for the scenario under investigation.

4.2 Experiments description

We inject the following errors: i) a memory consumption error (filling a Java *LinkedList*), and ii) a wrong network usage (fetching *HTML* text data from an external web page). We executed 60 preliminary runs in which we inject the memory consumption error and other 60 in which we inject the network error in our services. The validation experiments are organized as follows: in 40 runs we inject the memory error, while in the other 40 runs the network error is injected, considering different

Liferay services involved by the workload as injection points. Regarding the probing system, we observe 55 indicators [6], [22] from three different layers: 23 from the *CentOS* operating system, 25 from the middleware (the *JVM* [24]) and 7 from the *Network*. As explained in Section 3.3, we select the 20 most performing anomaly checkers (and consequently, the most relevant indicators) out of a set of 221 options.

4.3 Discussion of the results

We show the results of the anomaly detection framework. We first comment on the indicators and the anomaly checkers: in Table 2 and 3 we can observe the most performing anomaly checkers for each of the two error injections. Intuitively, the memory error injection can be detected observing indicators related to *Cpu* and *Java* memory; indeed, this can be verified considering the first three checkers selected in the training phase (Table 2). Similarly, concerning the network error, we expect to observe anomalies in the network layer (see *Tcp_Listen* in Table 3) or in the OS structures that process the incoming data flow (e.g., *Buffers* in Table 3).

In line *iv)* of Table 4 we show the results for the anomaly detection module: it behaves far better than the single anomaly checkers, because it uses a set of them. Moreover, despite the scores of the checkers are on average better for the experiments with network error, the detection capabilities of the framework are worse compared to the experiments with memory consumption injection. It follows that combining "better" anomaly checkers does not always lead to better scores for our anomaly detector. This efficiency strongly depends on the synergy between checkers: if a checker is not able to detect an error while another one is (e.g., they are related to indicators coming from different areas of the monitored system), this can *fix* the missed detection giving the framework the ability to answer correctly. In this study we considered each checker as a separate detector, and consequently the best checkers are chosen depending only on their score, without taking care of their characteristics. A possible improvement could be achieved considering the best *n* checkers for each monitored layer: in such a way, we are sure to consider checkers that observe different parts of the system, raising the likelihood of detecting anomalies.

In the experiments considered as validation set we obtained anomaly alerts in 95.8% of the runs when the memory error is injected: the missed detections are the remaining 4.2%. Regarding the 40 validation experiments with the network error

Table 2. 10 most performing anomaly checkers for the experiments with memory error injected

Indicator		Data type (Check)	FScore(2)
Name	Layer		
SysCpuLoad	OS	Diff (Hist)	0.37
SysCpuLoad	OS	Value (Hist)	0.35
ActVirtMPag	JVM	Value (SPS)	0.33
I/O Wait Proc	OS	Value (SPS)	0.31
Active Files	OS	Value (SPS)	0.30
Tcp_Syn	NET	Value (SPS)	0.28
Tcp_Listen	NET	Diff (SPS)	0.27
ProcCpuLoad	OS	Value (Hist)	0.26
ProcCpuLoad	OS	Diff (Hist)	0.25
Cached Mem	JVM	Value (SPS)	0.25

Table 3. 10 most performing anomaly checkers for the experiments with network error injected

Indicator		Data type (Check)	FScore(2)
Name	Layer		
Buffers	OS	Value (SPS)	0.45
PageIn	OS	Value (SPS)	0.42
Tcp_Listen	NET	Value (SPS)	0.40
PageIn	OS	Diff (SPS)	0.34
Cached Mem	JVM	Value (Hist)	0.33
Active Files	OS	Value (SPS)	0.31
User Procs.	OS	Value (SPS)	0.30
Tcp_Syn	NET	Value (Hist)	0.29
ActVirt Pages	JVM	Diff (Hist)	0.29
PageOut	OS	Value (SPS)	0.28

Table 4. Anomaly detection module performance

Detector Setup				Anom. Checks	Memory Experiment			Network Experiment		
#	Layers	Data	C-Aw		Precision	Recall	FScore(2)	Precision	Recall	FScore(2)
i	OS, JVM	*value*	NO	48	16.1%	59.5%	37.6%	35.1%	44.3%	42.1%
ii	OS, JVM, Net	*value*	NO	55	19.1%	65.6%	44.1%	43.8%	55.0%	52.3%
iii	OS, JVM, Net	*value, diff*	NO	110	22.7%	78.3%	52.5%	29.2%	72.2%	55.7%
iv	**OS, JVM, Net**	***value, diff***	**YES**	**221**	**33.5%**	**95.8%**	**69.8%**	**50.0%**	**86.7%**	**75.6%**

injection, instead, we obtained a correct error detection in the 86.7% of the runs.

It should be noted that with this configuration the framework provides an anomaly evaluation of the observed snapshot in 32.10 ± 5.99 milliseconds. This is the time needed by our framework to process each snapshot coming from the Target Machine.

Precision and Recall varying modules. We comment on the performances of the anomaly detector varying the modules and the anomaly checkers. From the top of Table 4 we summarize precision, recall and F-Scores obtained i) using the framework in [6], ii) introducing the network layer, iii) including the *diff* data series in addition to the default (*value*) for each indicator and iv) considering services information in combination with context awareness. Table 4 shows how using context awareness significantly raises the *F-Score*. Furthermore, as expected, introducing network probes significantly improves the *F-Score* in experiments with network errors.

Other framework configurations can be selected bringing to a higher balance between precision and recall. For example, considering *F-Measure* instead of *F-Score(2)* as reference metric we obtain a different set of anomaly checkers, ultimately resulting in precision of 41,0% and 80.2%, with recall of 58,3% and 73,3% respectively for the experiments with memory and network error injection.

5 State of the art and comparison with other solutions

Anomaly detectors have been proposed as error detectors [10] or failure predictors [2], based on the hypothesis that the activation of a fault (for error detection) or an error (for failure prediction) manifests as increasingly unstable performance-related behavior before escalating into a failure. The anomaly detector is in charge to observe these fluctuations providing a response to the administrator as soon as it can, triggering proactive recovery or dumping critical data. Reviewing state of the art it is possible to notice that the most used layers are the network [2], [3] and the operating system [6], [11]. This is not surprising since most of the systems include these layers: building solutions which fetch data from these layers allow building frameworks that fit in a very wide range of contexts. Regarding context-awareness, as highlighted in [16], in service-oriented architectures it usually refers to knowledge of the user environment to improve the performances of web services. For example, the *Akogrimo* project [17] aims at supporting mobile users to access data, knowledge, and computational services on the Grid focusing on user-context (such as user location and environmental information). In our work we refer to a server-side context-awareness, meaning that we do not require user information taking into account only runtime information about the services that are running in the SOA.

A detailed overview of anomaly detection frameworks can be found in [1]. Here we focus on three anomaly detection frameworks [2], [3], [14] addressing error detection/failure prediction where the authors reported the measurements for detection accuracy metrics (i.e., *precision* and *recall*). They observe indicators from multiple layers as the framework presented here does. We remark that these studies are exercised on *systems with low dynamicity*. *Tiresias* [3] predicts crash failures trough the observation of network, OS and application metrics by applying an anomaly detection strategy that is instantiated on each different monitored parameter. In *CASPER* [2], instead, the authors use different detection modules based on symptoms aggregated trough *Complex Event Processing* techniques based on the non-intrusive observation of network traffic parameters. Lastly, in [14] the authors aimed to detect anomalies due to the manifestation of hang, crash and content failure errors in an ATM system looking at OS indicators, exercising the framework on *Windows* and *Linux* kernels.

In Table 5 we reported the anomaly detection performance extracted from the surveyed studies. Detection performances (we show precision, recall and F-Score(2)) are strongly influenced by the characteristics of the target system: with low dynamicity it is easier to define a normal behavior, resulting in a significantly lower number of false detections (see [14], [2] and [3] in Table 5). Finally, looking at the performances of our framework we achieved a recall index that is competitive considering highly dynamic systems. Precision is low, meaning many false positives are generated, but in our setting we favoured recall since our aim is to minimize missed detections.

6 Conclusions and Future Works

In this paper we presented an anomaly detection framework for dynamic systems and especially SOAs. Assuming knowledge of the services that are running at time *t* on the observed machine gave us the opportunity to consider additional information that resulted fundamental to improve our anomaly detection capabilities.

As future works a sensitivity analysis directed to find the best *alpha* setup, a larger error model comprising *Liferay* software bugs, and an estimation of detection time varying number and type of observed layers will be investigated, along with strategies to reduce false positives. To further explore our context, we will focus on how changes in the user workload – and not in the services – can influence our detection capabilities and which strategies can be applied to maintain our solution working effectively. The basic failure model we considered for the experiments will be expanded including other items, to test the capabilities of the framework in different contexts.

Lastly, analysis aimed to understand the applicability of this solution when multiple SOA services are called simultaneously by different users will be investigated.

Table 5. Comparing performance indexes with similar studies.

	System Under Test			Precision	Recall	FScore(2)
	Characteristics	**Dynamicity**	**Layers**			
[14] (best UNIX)	ATM System	Very Low	OS	97.0%	100.0%	99.3%
CASPER [2]	ATM System	Very Low	Net	88.5%	76.5%	78.6%
TIRESIAS [3]	Emulab Distrib. Env.	Low	OS, Net	97.5%	n.p.	n.p.
Our Work - Memory	Secure! SOA	High	Net, OS, JVM	**33.5%**	**95.8%**	**69.8%**
Our Work - Network				**50.0%**	**86.7%**	**75.6%**

Acknowledgements. This work has been partially supported by the Joint Program Initiative (JPI) Urban Europe via the IRENE project, by the European FP7-ICT-2013-10-610535 AMADEOS project and by the European FP7-IRSES DEVASSES.

References

[1] Chandola, Varun, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey." ACM computing surveys (CSUR) 41.3 (2009): 15.

[2] Baldoni, Roberto, Luca Montanari, and Marco Rizzuto. "On-line failure prediction in safety-critical systems." Future Generation Computer Systems 45 (2015): 123-132.

[3] Williams, Andrew W., Soila M. Pertet, and Priya Narasimhan. "Tiresias: Black-box failure prediction in distributed systems." Parallel and Distributed Processing Symposium, IEEE 2007. IPDPS 2007.

[4] Tanenbaum, Andrew S., and Maarten Van Steen. Distributed systems. Prentice-Hall, 2007.

[5] Bose, S., S. Bharathimurugan, and A. Kannan. "Multi-layer integrated anomaly intrusion detection system for mobile adhoc networks." Signal Processing, Communications and Networking, 2007. ICSCN'07. International Conference on. IEEE, 2007.

[6] Ceccarelli, Andrea, et al. "A Multi-layer Anomaly Detector for Dynamic Service-Based Systems." Computer Safety, Reliability, and Security. Springer International Publishing, 2015. 166-180.

[7] Jyothsna, V., VV Rama Prasad, and K. Munivara Prasad. "A review of anomaly based intrusion detection systems." International Journal of Computer Applications 28.7 (2011): 26-35.

[8] Secure! project, http://secure.eng.it/ (last accessed 1st March 2016)

[9] Bondavalli, Andrea, et al. "Resilient estimation of synchronisation uncertainty through software clocks." International Journal of Critical Computer-Based Systems 4.4 (2013): 301-322.

[10] Modi, Chirag, et al. "A survey of intrusion detection techniques in cloud." Journal of Network and Computer Applications 36.1 (2013): 42-57.

[11] Shabtai, Asaf, et al. "“Andromaly”: a behavioral malware detection framework for android devices." Journal of Intelligent Information Systems 38.1 (2012): 161-190.

[12] Sokolova M., Japkowicz, Szpakowicz. "Beyond accuracy, F-score and ROC: a family of discriminant measures for performance evaluation." AI 2006: Springer Berlin Heidelberg, 2006. 1015-1021.

[13] Liferay, http://www.liferay.com (last accessed 1st March 2016)

[14] Bovenzi, Antonio, et al. "An OS-level Framework for Anomaly Detection in Complex Software Systems." Dependable and Secure Computing, IEEE Transactions on 12.3 (2015): 366-372.

[15] Erl, Thomas. Soa: principles of service design. Vol. 1. Upper Saddle River: Prentice Hall, 2008.

[16] Truong, Hong-Linh, and Schahram Dustdar. "A survey on context-aware web service systems." International Journal of Web Information Systems 5.1 (2009): 5-31.

[17] Loos, Christian. "E-health with mobile grids: The Akogrimo heart monitoring and emergency scenario." Akogrimo White Paper,(Online (2006).

[18] Esper Team and EsperTech Inc. “Esper Reference version 4.9.0”, Technical Report, 2012

[19] Valls, Marisol García, Iago Rodríguez López, and Laura Fernández Villar. "iLAND: an enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems." Industrial Informatics, IEEE Transactions on 9.1 (2013): 228-236.

[20] rclserver.dsi.unifi.it/owncloud/public.php?service=files&t=89f4b993136bda20ae9cfb3f32ac62da

[21] Thramboulidis, Kleanthis, Doukas, and Koumoutsos. "A SOA-based embedded systems development environment for industrial automation." EURASIP Journal on Embedded Systems (2008): 3.

[22] Bondavalli, Andrea, et al. "Differential analysis of Operating System indicators for anomaly detection in dependable systems: An experimental study."Measurement 80 (2016): 229-240.

[23] Zoppi, Tommaso. "Multi-Layer Anomaly Detection in Complex Dynamic Critical Systems.", Dependable Systems and Networks – Student Forum Session, DSN 2015.

[24] Cotroneo, Domenico, et al. "Failure classification and analysis of the java virtual machine.", ICDCS 2006. 26th IEEE International Conference on. Distributed Computing Systems IEEE, 2006.