View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Florence Research

Multi-Layer Anomaly Detection in Complex Dynamic Critical Systems

Tommaso Zoppi
Department of Mathematics and Informatics
University of Florence
Viale Morgagni 65, Florence, Italy
tommaso.zoppi@unifi.it

***Abstract*— Revealing anomalies to support error detection in complex systems is a promising approach when traditional detection mechanisms (i.e., based on event logs, probes and heartbeats) are considered inadequate or not applicable: anomalies in data usually suggest significant, and also critical, actionable information in a wide variety of application domains. The detection capability of such complex system can be enhanced observing different layers and indicators to achieve richer information that describe the system status. The paper describes the context and the state of the art in association with the current research direction of the author with the aim to highlight the challenges and the future works that the student aims to perform in the next years.**

Keywords—anomaly detection, error detection, monitoring, fault injection, SOA, Secure!

I. Introduction

Large-scale software systems as for example Service Oriented Architectures (SOAs) or cyber-physical infrastructures in general are composed of several different components, software layers and services. These systems are characterized by a dynamic and evolutionary behavior, which leads to changes in part of the system, as well as their services and connections. Recent trends show the increasing introduction in these systems of safety-critical requirements, as for example in crisis management systems where rescue personnel on-the-field is remotely guided [1]. To deploy monitoring solutions of the system and its services to timely detect failures we must consider the complexity of software and of the dynamic and evolutionary behavior of the whole system, make the definition and instrumentation of a monitoring solution an open challenge [2].

The complexity of the code, the management and the evolution of their services, make these systems exposed to residual software faults. The activation of that faults can result in failures and services downtime which ultimately may lead to huge safety violations, financial losses and consumer dissatisfaction, so it is very important to continuously check the services' status. The dynamic and evolutionary characteristics of these systems call for monitoring solutions which are as much independent as possible from the services running on the application layer, in order to *not*: i) require information on the services, ii) need to instrument the services with monitoring probes, iii) be forced to reconfigure the monitor each time services are updated, added or removed.

In this paper we focus on anomaly detection, which refers to the problem of finding patterns in data that do not conform to the expected trend. Such patterns are changes in the observed indicators characterizing the behavior of the system caused by specific and non-random factors i.e., pattern changes can be due to a system overload, or to the activation of faults. Anomaly detectors may be able to infer the status of a service without directly observing it, but observing the "surroundings", or rather the lower abstraction levels we chose to instrument. This has been proven relevant and useful for dynamic software and systems which are subject to frequent changes or when the instrumentation with probes of the target services is not allowed (e.g. the source code is not available) or unfeasible.

The paper is organized as follows. Section II presents the background and related work, Section III highlights our research contribution, the structure of the examined framework and the related open challenges while in Section IV we describe a case study in which we applied our anomaly detector. Finally, Section V summarizes some key aspects and defines the future works.

II. Background and Related Work

Several approaches can be identified in the state of the art for anomaly detection in complex systems. For example, in [3] the authors summarize some useful techniques – especially based on statistical and clustering algorithms – aimed to reveal anomalies in distributed wireless sensor networks, while in [4] the authors describe how this approach is useful for intrusion detection purposes in cloud systems. Anyway, all the anomaly detection techniques need information on the system behavior, obtained by monitoring the system during its operational life. This technology is widely used in many kinds of software: Web services and SOA monitoring is executed in parallel with the normal executing processes without interrupting them.

Regarding our purposes, it is also interesting to highlight some recent works related to the observation of different specific abstraction levels of a complex system. In [5] we can observe that the authors tried to summarize what abstraction levels can be monitored in a generic cloud system, depending on the characteristics of that system (SaaS, PaaS, IaaS). The

work in [6] focus the attention on a configurable detection framework aimed to reveal anomalies in the Operating System (OS) behavior: the detector is based on online statistical analysis techniques, and it is designed for systems that operate under variable and non-stationary conditions. The framework is evaluated in order to detect the activation of software faults in a distributed system for Air Traffic Management (ATM).

To remark the utility and the wider applicability of this approach we can see that also in different application contexts, like the optimization of business processes [7], different abstraction layers can be observed to reach different goals based on the granularity of the patterns that we can use for recognize specific situations.

As abovementioned, monitoring a system gives the opportunity to understand the behaviour of the observed components when a specific activity was executed. However, since we are interested in detecting anomalies due to software errors, we need to understand how those components react when an error is activated in the system. One way to support and speedup that "error observation" consists in deliberately insert the chosen fault in the monitored system, with the aim to observe its reaction: this is a well-known process that is called fault injection [8]. With this technique, we become able to observe the reaction of the system when a fault is injected at specific abstraction level as defined in the fault library.

III. Contribution and Challenges

The novelty of our approach consists in shifting the observation perspective from the services at the application layer to the underlying layers; currently we are considering the middleware or Application Server (AS hereafter) and the Operating System (OS hereafter). Updating the requirements in this way allows to: i) get monitored data from different (but not independent) abstraction levels; ii) observe systems in which Off-The-Shelf components are running at the application layer and, mainly, iii) build a monitoring structure with an high level of flexibility with respect to the services that are executed at the application layer.

All these characteristics make this solution very suitable in contexts like SOAs, in which a lot of different services are executed on a set of server machines that can also share the lower abstraction levels, as well as distribute *FileSystem*, or cloud environments.

A. The Framework

As described before, the anomaly detector needs a monitoring structure that retrieves data from the target system. As we can see in Fig. 1, the retrieved data is separately processed by different instances of the detection algorithm, whose results are collected and weighted by the voting function. Once an anomaly is found in the examined observation set, an alarm is thrown to a module (the operating center) that is in charge to perform the appropriate actions to face up the suspicious situation. These modules – monitor, detector and operating center – are the main components of a framework that could be installed on all the systems that needs of an additional control of their behavior. Suppose you have a server farm in which the malfunction of one machine influences the entire system. Installing the framework on the servers could help to detect malfunctions on each machine, protecting the other servers from its imminent failure.

An important role is played from the tuning process of all parameters that characterizes that framework: depending on the context, some OS and AS attributes are more relevant than others, and the reaction policies must be calibrated based on the requirements of the target system. During the setup phase of the framework, an observation of the behavior of the system must be performed to define i) the set of monitored attributes, ii) the detection function parameters and iii) the choice of the reaction strategies implemented in the operating center. This framework is designed to work independently with respect to the applications or the services that are running in the target system, with a lot of advantages in terms of adaptability and wider applicability. Anyway this generality leads us to some difficulties mainly related to the choice of the parameters to observe or the type of detection function to use. We must conclude that a tuning phase is requested in each installation of the framework, to tailor the parameters on the specific system to improve the effectiveness of the system.

B. Performance Improving Challenges

The main challenges related to this approach are essentially linked to i) study the context with the aim to discover the most frequent (or dangerous) errors that can affect the target system, and ii) understand which set of indicators - coming both from the OS and the AS - is more useful than others to perform the detection of the anomalies produced by those errors. This utility is strongly affected by the choice of the fault model (defined with the support of the studies at i)), which summarizes the types of errors we want to detect: i.e., monitor the HTTP traffic is probably less useful than tracing RAM usage if you want to recognize a memory overload. The fault model must be built according both to the vulnerabilities of the system and the likelihood of the faults that can affect the context; we can think about errors due to hardware faults, human mistakes, external/internal attackers, software bugs.

Once the set of monitored attributes is defined, the data is processed by the detection algorithm. This algorithm has to work in a dynamic context, in which a lot of different applications can be used; because of this, all the techniques that are built to work in semi-static scenario (i.e., some types of algorithm based on pattern recognition) do not fit with our context. As we shall see in the next section, for our first experiments we chose a statistical detection strategy that rise alerts when an observation is out of an expected range built dynamically depending on the last collected observations and

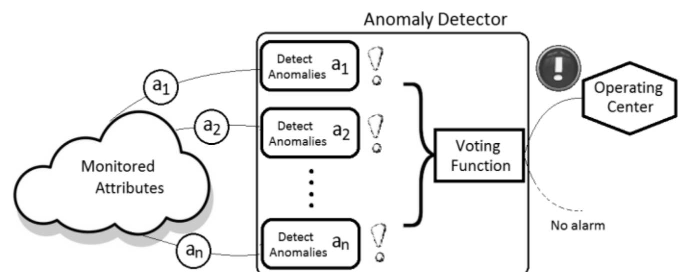

Fig. 1. Main components of the monitoring framework

not from some static data or patterns. All the parameters of the chosen algorithm, in addition to the monitored attributes, need to be tuned to guarantee the maximum efficiency of the anomaly detection process. Fault injection experiments should be conducted with the aim to collect a lot of data series that can be used as input values of a parameter optimization process that we can summarize as follows.

Let $f : \langle Ma, Ap, V\rangle \rightarrow fd$ be a function that builds detection functions based on three input sets where *Ma* represents the set of monitored indicators, *Ap* denotes the set of parameters of the detection algorithm and *V* points to the voting strategy. The output is a function $fd : Oi \rightarrow Ar$ that takes an observation (*Oi*) at the time instant i of the entire set of the monitored indicators and signals an anomaly rate value – *Ar*– that indicates if an anomaly is suspected for this specific input. Using the data collected with the abovementioned experiments, our objective is to find the *fd* function - or rather, the triple $\langle Ma, Ap, V\rangle$ - that minimizes the number of false alarms raised by the process (both false positives/negatives). As we can see, this process cannot have a unique resulting function for all systems in which the framework is going to work because the *f* function inputs, especially *Ma* and *Ap*, are strongly influenced by the context.

Last, every system must implement the primitives that allow the operating center to react in a crisis situation, to take back the system in a proper (not anomalous) state when the detector signals an alteration from the expected behavior.

IV. The Secure! Case Study

The Secure! Framework, which is being developed in the context of the Secure! Project [10], is a novel Decision Support System (DSS) for crisis and emergency management. It exploits information retrieved from a large quantity and several types of sensors, including crowd sensing, in order to detect critical situations and perform the corresponding reaction. Users will have the opportunity to interact with the Secure! framework using their mobile devices to provide and receive information about real events or dangerous situations in which they could be involved. Secure! should also be able to detect critical situations before they happen analyzing real events provided by the social media and correlating them with historical data and events incoming from other sources.

A. The Secure! Framework

The system is organized as a SOA structure and basically divided into four distinct levels each of them comprises logical components and services based on the input data coming from different sources (social media, web sites, mobile devices …). Starting from the bottom level, data are received, collected, homogenized, correlated and aggregated in order to produce the Secure! situation.

The services running on the Secure! system have different ownerships and authorships, and may incur in frequent updates and removal, or even new services may be introduced (i.e., addition of another input processing technique), together with modification to their orchestration. Thus, while instrumenting with probes and monitoring each service is unfeasible, the opportunity to observe the underlying layers (AS and OS) is offered and a more accurate detection of errors become available thanks to the combined usage of AS and OS monitoring. We can also notice that the services are running on a heterogeneous set of virtual machines that have the same type of OS and AS, although with different settings and computational ability. However, all of these machines share the requirement to process the data in a proper way, because of the criticality of the tasks needed in this DSS.

B. Anomaly detection solution for the Secure! system

Because of this necessity to carefully control the behavior of each virtual node, we install a basic version of the framework on the machines, in which a prototype of Secure! was already running. Due to the dynamicity of the Secure! scenario, we chose a statistical detection algorithm (*Statistical Prediction and Safety Margin*, SPS [11]) that every time the monitor collects information about the value of a set of indicators, executes a prediction calculation aimed to identify an interval in which each of the observed values must fall. Following the structure in Fig 1, if the observed value is out of this range, an anomaly is suspected for such indicator and a notification is thrown to the voting module. The voter executes a simple sum of the notifications that come from the instantiation of SPS for each indicator, and raises an alarm to the operating center if a static threshold value is reached.

Another important element we must define in order to adapt the anomaly detection process to the scenario is the set of faults that could involve the system under observation (e.g. external attacks, software bugs …). Our main interest in this study was to understand the impact that software bugs (e.g., programmer errors) located in the source code of the applications might have on the entire node. In [9] the authors, after examined some real case studies, summarized the most frequent software bugs that can affect a code; this collection of results perfectly fits with our requests because of the generality of the listed faults, so we adapted it as our fault model.

C. Assessment of the Anomaly Detection Solution

Once defined the fault model, a testbed [12] was built in order to conduct assessment experiments based on fault injection aimed to collect a huge amount of data related to the behavior of the indicators when a fault is activated in the Secure! prototype. As described in [12], the monitor is able to collect data related to 50 different indicators, coming both from OS and AS. With an analysis of the data collected with the support of the testbed we were able to reduce this starting set applying the SPS algorithm to the experiment traces. The aim is to tune the parameters (essentially 6, as shown in [11]) of the statistical detection algorithm with a kind of supervised learning process based both on the traces and on the information about the faults we injected in each experiment.

The output of that process is an *fd* function that was tailored to the Secure! prototype from such tuning process in which the *Ma* set is composed from 20 AS indicators and 9 OS ones, *Ap* is a set of 29 sextuples (that represents the more performing instantiation of SPS for a specific single indicator) and *V* is a simple sum function.

D. Implementation of the Testbed

In Fig. 2 we depict the structure of our testbed. In the bottom we can notice the modules owned by the target system, that is a virtual node equipped with *Linux CentOS* 6 and *Apache Tomcat 7.0.40* as middleware layer above which a prototype of the Secure! system is running. Probes were installed with the aim to retrieve data from OS and AS and send them to the system monitor, which is located in another machine that we use to run experiments. On this machine a storage module called *DataLogger* is invoked by the monitor once the experiment is finished in order to aggregate, filter and finally store the collected data into a *MySQL* database.

Experiments were conducted with the support of two modules: the *Workload Generator*, that builds an XML workload file based on configurable user settings, and a *Fault Injector*, that is able to perform a compile-time injection of the faults summarized in [9] giving the possibility to dynamically activate a subset of them depending on the experiment needs. The injector, that is actually available on our company for research purposes, is able to process any *Java* source in order to identify all the possible injection points and change the chosen parts of the code in which a fault is going to inject.

Each experiment is conducted in the following way: first we decide the fault set that we want to activate, then a workload is built and executed on the target system, in which a modified version of the Secure! system is running. During this process, probes retrieve data both from OS and AS and send it to the experiment machine through system pipe; data is collected, aggregated and stored in a database at the end of the execution of the chosen workload.

V. Conclusions and Future Works

The installation and the utilization of a first version of our framework in a context like Secure! gave us the opportunity to test our idea in a real context with critical requirements. Results of the efficiency of the implementation described in section IV.B are currently in a finalization and summarization phase, and will be presented soon to the community.

Just to give a brief idea of these outcomes, we observed that in a lot of cases the activation of the faults defined in [9] do not have effects that significantly change the trend of the observed variables. To limit this problem, we tried to update the parameters of the detection algorithm to obtain a higher sensibility and detect also the little variations. This approach gave us a higher number of false alarms, so further analysis aimed to find a sensitiveness tradeoff are under investigation.

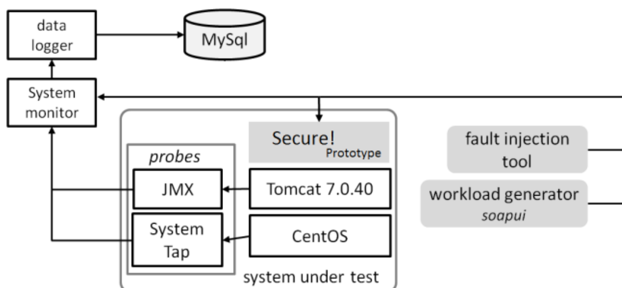

Fig. 2. Implementation of the testbed [12]

Future works we want to perform are aimed to understand the potential, the limits and the range of applicability of this cross-level anomaly detection: what are the most fitting detection algorithms, the metrics that notify fewer false alarms, the abstraction levels that give more accurate information than others and especially the evaluation of the effective support that this techniques give to the safety of the systems in exam. These studies will be also useful in order to automatize as best as we can the installation and tuning process of our framework, to deploy a tool that can be installed on a wide set of systems with default settings and high personalization capabilities.

To improve the effectiveness of the anomaly detection framework we also have to investigate different types of critical systems to understand all the possible relationships between them: for example, if an indicator (related to memory utilization, CPU consumption …) is useful in a lot of examined systems or if a voting strategy have very good performances in terms of effectiveness optimization of the detection process, probably might be included in the base setup of the framework.

Acknowledgment

This work has been partially supported by the European Project FP7-PEOPLE-2013-IRSES DEVASSES, the Regional Project POR-CREO 2007-2013 Secure!, and the TENACE PRIN Project (n. 20103P34XC) funded by the Italian Ministry of Education, University and Research.

References

[1] S. B. Eom, S. M. Lee, E.B. Kim, and C. Somarajan, "A survey of decision support system applications," Journal of the Operational Research Society, pp. 109-120, 1998.

[2] A. Bovenzi, F. Brancati, S. Russo, A. Bondavalli, “An OS-level Framework for Anomaly Detection in Complex Software Systems”, IEEE Transactions on Dependable and Secure Computing, (in press).

[3] Xie, Miao, et al. "Anomaly detection in wireless sensor networks: A survey." Journal of Network and Comp. Applications 34.4 (2011): 1302-1325.

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

[5] Spring, Jonathan. "Monitoring cloud computing by layer, part 1." Security & Privacy, IEEE 9.2 (2011): 66-68.

[6] A. Bovenzi, S. Russo, F. Brancati, A. Bondavalli, "Towards identifying OS-level anomalies to detect application software failures," IEEE Int. Workshop on Measurements and Networking (M&N), pp. 71-76, 2011.

[7] Schumm, David, Gregor Latuske, and Frank Leymann. "State Propagation for Business Process Monitoring." Proceedings of the 19th European Conference on. 2011.

[8] Hsueh, Mei-Chen, Timothy K. Tsai, and Ravishankar K. Iyer. "Fault injection techniques and tools." Computer 30.4 (1997): 75-82.

[9] Duraes, Joao A., and Henrique S. Madeira. "Emulation of software faults: a field data study and a practical approach. "Software Engineering, IEEE Transactions on 32.11 (2006): 849-867.

[10] Secure! Project, http://secure.eng.it/.

[11] Bondavalli, Andrea, Francesco Brancati, and Andrea Ceccarelli. "Safe estimation of time uncertainty of local clocks."Precision Clock Synchronization for Measurement, Control and Communication, 2009. ISPCS 2009. International Symposium on. IEEE, 2009.

[12] Ceccarelli, Andrea, et al. "A Testbed for Evaluating Anomaly Detection Monitors Through Fault Injection." Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on. IEEE, 2014.