2012 Sixth International Conference on Complex, Intelligent, and Software Intensive Systems

Intrusion Tolerance in Cloud Applications: the mOSAIC Approach

Massimo Ficco and Massimiliano Rak
Seconda Universita' di Napoli (SUN)
via Roma 29, 81031 Aversa, Italy
{*massimo.ficco, massimiliano.rak*}*@unina2.it*

***Abstract*—Cloud Computing is a recognized emerging solution for building Internet applications, which founds on delegation of every kind of resources to the network and on a pay-per-use business model. Cloud application, which runs consuming Cloud resources offered by Cloud Providers, offers open interfaces to their users, which access them from Internet and are often prone to Denial of Services attacks. This work focuses on the mOSAIC approach for development of Cloud applications, which offers a solution for gathering resources from many different providers. It shows how it is possible to enrich the mOSAIC platform with tools that, in a simple and transparent way, protect mOSAIC Cloud application from some well known Denial of Services attacks. The paper focuses on a single kind of attacks, called Deeply-Nested XML, in order to show the proposed approach and offer some preliminary results, which demonstrate the validity of the solution proposed.**

I. Introduction

Cloud applications are distributed applications which run consuming resources obtained from Cloud Providers. Such applications may use dedicated Cloud API that runs on the top of Cloud computing Platform as a Service (PaaS) solutions or can directly consume Cloud Infrastructure as a Service (IaaS), like Virtual Machines (VM) and Cloud storage systems. Cloud applications are offered typically *as a Service* to their users and are accessible from Internet, following the typical Cloud business model (pay-per-use and self-service management), *i.e.*, Cloud application owners buy resources from Cloud Provider and sell the application to their own users.

Cloud applications, due to their openness to the Internet, are prone to Denial of Services (DoS) attacks, that have a very high cost for the application owner. DoS attacks aim at reducing the service's availability by exhausting the resources of the service's host system, like memory, processing resources or network bandwidth. The application users usually does not pay for low level resource consumption, but for high level services, so the cost of additional resources consumption is completely on charge of the application owner. It is interesting to note that Cloud applications are usually subject to the same kind of attacks of the Web applications (in fact in most cases Cloud applications are common Web applications, which are deployed trough dedicated Cloud platforms), but the attacks have direct effects on the application costs and not only on the performance perceived by users.

This research is partially supported by FP7-ICT-2009-5-256910 (mOSAIC project).

In literature there are several approaches for development of such applications, In this paper we focus on the mOSAIC solution [1], [2], [3], which offers an API and a platform that enables the execution of Cloud application on a federation of Cloud Providers. Our goal is to show how it is possible to integrate security mechanisms in existing mOSAIC solution in order to protect applications from a given set of DoS attacks.

Using the programming model offered by mOSAIC, it is possible to add Intrusion Tolerance (IT) features in a mOSAIC application in a transparent way respect to final users (*i.e.*, respect to the application behavior) and with a minimal effort from Cloud application developer.

The remainder of this paper is organized as follows: next Section (section II) is dedicated to IT technologies and offer a basis for the description of the IT solutions adopted in our proposal. Section III is dedicated to describe the mOSAIC approach to Cloud application development (which is needed to fully understand the approach proposed), while Section IV, clarify the approach adopted to solve the problem. Section V describes the mOSAIC components developed in order to introduce IT features in Cloud applications, while Section VI describes a simple example of usage of such components in a real case study. Section VII concludes the paper with a summary of the results obtained and describing future works.

II. Detection and Tolerance of Cyber Attacks

In general, the security of a Cloud application against cyber attacks is hard to achieve for complexity, heterogeneity, and the high distribution of such systems [4].

Nowadays, the technologies available to protect network-based systems from attacks provide only a basic support for attack detection or intrusion prevention [5]. On the other hand, for business critical services such as Google, eBay, Amazon, it is preferable to provide mechanisms that prevent intrusions from leading to a system security failure. For example, it is preferable to continue the system operation after an intrusion, rather than shutting the whole system down along with the services being provided. The ability to diagnosis the intrusion effects is becoming increasingly important, in order to infer the least costly and most effective reaction aimed to avoid intrusions from generating a system failure.

On the other hand, Intrusion Detection Systems (IDSs), which are the main solution to protect networked systems, do not directly detect intrusions, but only the attacks symptoms, $i.e.$, the erroneous or anomalous behaviors of the system [6]. Intrusion Prevention Systems (IPSs) can be considered as an extension of IDSs. They simply aim at detecting and stop attacks by an on-line reaction ($e.g.$, by reconfiguring the targeted system). They usually include algorithms to perform more sophisticated traffic inspection, and to operate at the application layer in addition to performing classic network and transport layers processing. To achieve this functionality, developers have progressively incorporated scanning and checking mechanisms into firewalls. However, the capability of identifying the attack does not necessarily imply the ability to select the best recovery action. For example, an attack detection that does not correspond to an intrusion can produce a reaction that isolates system resources ($e.g.$, nodes), thus reducing system's availability and reliability.

Intrusion Tolerance (IT) is considered as an emerging paradigm for developing systems that are able to provide an acceptable level of service, even after intruders have broken in [7]. IT is the ability to react, counteract, recover and mask errors, which may lend to failures, if nothing is done to counter their effect on the system. IT aims to guarantee that the considered system provides the correct services even in the presence of malicious intentional actions, which may lead to violations of the system's security properties [8]. In order to enforce the intrusion tolerance several techniques can be used, including replication, rejuvenation, redundancy, indirection, reactive techniques, and reconfiguration [9], [10], [11], [12].

In next Sections, we present a mOSAIC solution that implements an indirection technique to tolerate specific attacks to the mOSAIC Applications.

A. Intrusion Tolerance technique

In this paper, we focused on DoS attacks that aim at exhausting the computational recourses of the target machine for extended periods time in order to reduce the application service availability. In particular, we consider the *Coercive Parsing* attack, also called *Deeply-Nested XML*, which is a resource exhaustion attack that exploits the XML message vulnerabilities by means of inserting of a large number of nested XML tags in the message body. The goal is to force the XML parser within the server application to exhaust the computational resources of the host VM, by processing numerous deeply-nested tags.

The defined approach aims at triggering a specific reaction, in order to mitigate the effects of the intrusions on the host VM, and improve the availability of service provided by the mOASIC application. In articular, the IT mechanism implements an *indirection* technique, which allows to insert a protection barrier between clients and applications transparently to the application development. We adopt an active monitoring approach, which consists of monitoring anomalous VM's CPU consumptions ($i.e.$, CPU overloading due to malicious XML messages), and correlating this information with symptoms of attack on the base of the temporal proximity. When an attack is detected, an adaptive reaction is triggered on the base of the target CPU load. Obviously, the kind of reaction cannot be absolute, but must be defined with respect to an intrusion model, that is, the specification of effects that the successful attack has on the target system ($i.e.$, the degree of success of the intruder in terms of CPU consumption).

In a transparent manner with respect to both the mOSAIC Application and the Final User, a threshold-based over-time monitoring mechanism is adopted to detect anomalous CPU consumption. It counts consecutive excessive CPU consumption, and when the count exceeds a fixed threshold an event is generated and a diagnosis activity is performed. The diagnosis consists of verifying the presence of Deeply-Nested XML attack symptoms by using an anomaly-based monitoring mechanism, $i.e.$, detecting the anomalous number of nested XML tags included within each message with respect to a training profile. If attack symptoms are observed, the reaction is performed. It filters each XML request that contains a number of nested tags greater than adaptive threshold, which is decreased until the CPU consumption fall below a severity level.

III. mOSAIC: Development of Distributed Cloud Applications

mOSAIC aims at offering a simple way to develop Cloud applications. The target user for the mOSAIC solution is a developer, which we call *mOSAIC User*. In mOSAIC a Cloud application is modeled as a collection of components that are able to communicate each other and that consumes/uses Cloud resources ($i.e.$, resources offered as a service from a Cloud Provider).

Cloud applications often are offered in the form of *Software as a Service* and can be accessed from users different from the mOSAIC developer. Users different from the mOSAIC User, which uses a Cloud application are defined *Final Users*. The mOSAIC User is able to act as a Service Provider toward the Final Users.

The mOSAIC solution is a framework composed of three independent components: *Platform*, *Cloud Agency* and *Semantic Engine*: the mOSAIC Platform enables the execution of application developed using mOSAIC API; the Cloud Agency acts as a provisioning system, brokering resources from a federation of Cloud Providers; and the Semantic Engine offers solutions for reasoning on the resources and application needing. For the needing of this paper, we need concepts related to Platform and Cloud Agency, while Semantic Engine will not be focused in this context.

mOSAIC solution can be adopted in three different scenarios: (i) a developer wants to develop an application that runs independently from the Cloud Providers, (ii) an IaaS Provider aims at offering enhanced services, and (iii) a single user (like a scientist) is interested to start his own application in the Cloud for computational purposes. In this paper we will focus on the first scenario ($i.e.$, the one dedicated to cloud applications which runs independently from cloud providers).

In such scenario, the mOSAIC User develops the application on its local resources, trough the mOSAIC development tools, then it deploys the application on the Cloud. The developer uses a local instance of the Cloud Agency in order to startup the process of remote resource acquisition, then it deploys the platform and the needed components on the chosen remote resources.

A. Programming with mOSAIC

A mOSAIC Application is defined as a collection of interconnected *mOSAIC Components*. Components may be offered by the mOSAIC Platform (i) as COTS (Commercial off-the Shelf) solutions, $i.e.$, common technologies embeded in a mOSAIC Component, (ii) as Core Components, $i.e.$, tools which enable application excution and composing the mOSAIC platform, (iii), as Service Components, $i.e.$, tools offered by mOSAIC Platform in order to perform predefined operations , or (iv) as new components that can be developed using mOSAIC API. In last case, a component is a *Cloudlet* running in a *Cloudlet Container* [13], [14].

mOSAIC Components are interconnected trough communication resources, such as queues or other communication mechanishs ($i.e.$, socket, web services, ...). The mOSAIC Platform offers some queuing system (rabbitmq and zeroMQ) as COTS components, in order to enable component communications. Moreover, mOSAIC Platform offers some Service Components in order to help Cloud application to offer their functionalities as a service, like an HTTP gateway, which accepts HTTP requests and forwards them on application queues. mOSAIC Components run on a dedicated operating system, thought to run on cloud-based VMs, named mOS (mOSAIC Operating System), which is a very small Linux distribution. The mOS hosts the mOSAIC Core Component, like the *Platform Manager*, which manages set of VMs hosting the mOS, as a virtual cluster on which the mOSAIC Components are independently managed. It is possible to increase and decrease the number of VMs dedicated to the mOSAIC Application, which scale in and out automatically.

The Cloud application is described as a whole in a file named *Application Descriptor*, which lists all the components and the Cloud resources needed to enable their communications. A mOSAIC User has the role of both develops new components and writes Application Descriptors that collect them together.

mOSAIC API, actually based on Java, enables the development of new components in the form of *Cloudlets*. Cloudlets are software components that self-scale on the above described Platform and consume every kind of Cloud resources, including queues, NO-SQL storage system (like KV store and columnar databases), independently from the technologies and API they adopt, trough a wrapping system.

B. mOSAIC offering for SLA-based applications

mOSAIC offers a set of features dedicated to SLA management. In [15] we proposed a first outline of the full architecture, while in [16] we proposed other case studies, dedicated to applications that offers security access configurations to a GRID environment in terms of SLA and to different mosaic scenarios. The components offered in the SLA architecture should help the mOSAIC User to implement a SLA-based architecture.

Figure 1 summarizes the global architecture offered in mOSAIC, showing the main modules and their respective roles. As shown in the picture, the main modules offered by mOSAIC are:

- *SLA Negotiation:* This module contains all the Cloudlets and components which manage the SLA documents and their formal management, $i.e.$, negotiation protocols, auditing, and so on.
- *SLA Monitoring:* This module contains all the Cloudlets and components needed to detect the warning conditions and generates alerts about the difficulty to fulfill the agreements. It should address both resources and applications monitoring. It is connected with the Cloud Agency.
- *SLA Enforcement:* This module includes all the Cloudlets and components needed to manage the elasticity of the application, and modules that are in charge of making decisions in order to grant the respect of the acquired needed to fulfill the agreements.

In this paper, we will use components that derive both from the Autonomic module (in order to take decisions and adapt the application behavior to the attacks) and from the Monitoring module (in order to detect the attacks).

C. SLA Enforcement: the SLA policy component

From the Enforcement module we will use only a single component: the SLA policy component, the overall module description is skipped, because it is out of the interest of this paper. The SLA Policy component acts mainly as a Policy Decision Point, $i.e$, it evaluates policies and emits messages on the basis of the policy evaluations. The component uses two different queues, the first one is dedicated to input messages and the second one for messages generated on the basis of the policy evaluated. Moreover, the component uses two different storages, one dedicated to Policies, the

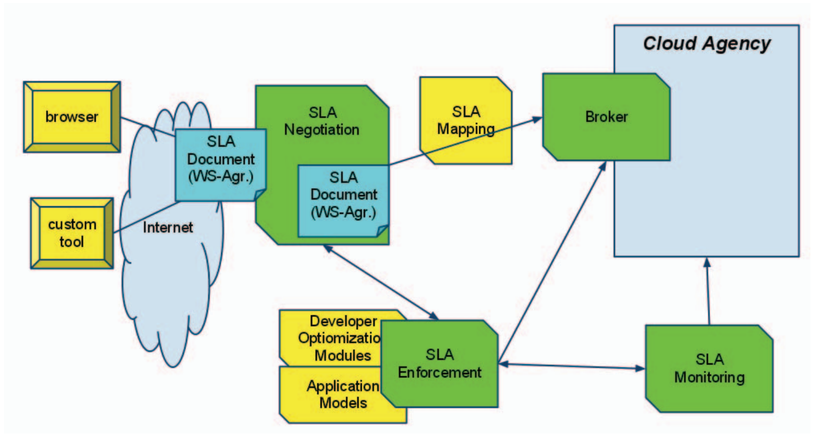

Figure 1: mOSAIC architecture

second one, shared with the monitoring system, on which the monitoring parameters are stored.

The input messages can be dedicated to policy management (add, delete, update a policy) or to policy evaluation. The policy are represented in a JSON format and are set of triples of kind [`<Parameters>, <Condition>, <Action>`]. Parameters are the keys needed to retrieve monitoring parameters from the Monitoring storage. Conditions is a boolean expression which can be evaluated using Parameters as variables. If Condition evaluation is true a message containing the Action value is sent on the output queue.

This component can be used by mOSAIC Applications in order to assume decisions about the evaluation of the application. In this paper we use this component in order to detect attacks on the basis of the monitored data.

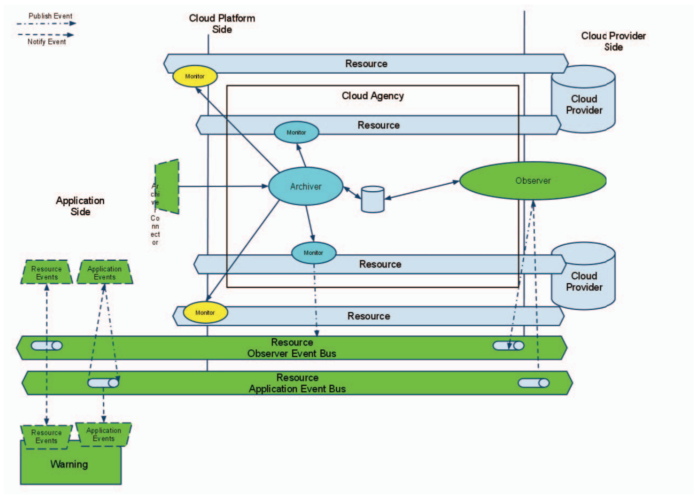

Figure 2: mOSAIC monitoring components architecture

D. mOSAIC offerings for application monitoring

In the context of mOSAIC Framework, the Monitoring System offers a set of tools (cloud components, resources, connectors), whose goal is to evaluate the global state of the distributed application and to generate warning when the target application and/or the associated resources are in conditions which may lead to a violation of the SLA. In the following, we briefly summarize the main concepts related to mOSAIC application monitoring, a deeper analysis and examples of the component usage can be found in [17].

The main duties of the Monitoring system are:

- monitor Cloud resources;
- monitor application components;
- verify the compliance of a set of rules, in order to discover warning conditions.

The Cloud resources monitoring is a feature strictly linked with the resource provisioning, so it is mainly managed from the Cloud Agency. Monitoring Cloudlets and components, instead, is strictly related to the application development and it should be supported by the API, trough ad-hoc solutions. Moreover, the Cloudlets and components (both user developed or offered by the Platform like SLA components) need a simple way to share the monitoring information and manage the related events.

In order to face the above described requirements, the proposed architecture is organized as illustrated in Figure 2, containing the following main elements:

1) *Monitoring Event Bus*, that collect the monitoring events from a large set of different sources;
2) *Connectors* dedicated to the Event Bus, which enable Cloudlets to intercept monitoring events and that generate events in the Event Bus from the application (they act like logging or tracing modules)
3) *Warning Component* that is a stand-alone Cloudlet, which offers the warning features to the Cloud application.

mOSAIC Users can use the connectors and other mO-

SAIC facilities in order to build up dedicated components able to retrieve from the Event Bus the monitoring information needed for their specific goals. An example of such component will be described in the next sections.

Communication between the Resource Monitoring system and the mOSAIC Application is based on the adoption of shared Event Bus. In the picture, we propose mainly two different Event Bus. The first bus is adopted in order to collect all the events from Cloud resources, the second one is adopted to collect events from applications. It is important to point out that it is up to the mOSAIC User in order to choose the number of bus to involve: all the events can be managed trough a single bus or split in as many bus as the developer consider necessary. In the general architecture, we just split them in two, considering that there are two main sources of events: the Cloud resources and the application components.

The resource monitoring is mainly based on the adoption of the Cloud Agency. It offers the *Archiver*, a monitoring agent that collects monitoring information from the agents distributed on all the monitored resources ($i.e.$, VMs), storing them in a storage system. In addition, the Monitoring agent should be able to collect monitoring information from common monitoring systems (like Ganglia, Nagios, SNMP-based applications) and publish them on the same storage. Applications are able to interact with the Archiver trough a Connector which enables Cloudlet to access Archiver information.

The Observer is a component that accesses to the storage filled from the Archiver and generates events on the resource Event Bus. It is the duty of the Observer to generate events in order to distribute *selected information* to all the interested components.

The Warning component acts on the basis of a policy-based approach: the developer loads a set of rules that identify the warning conditions to be verified on the events generated on the Event Bus. It uses the connectors offered by mOSAIC to access to the Event Bus.

IV. mOSAIC Approach to Intrusion Tolerance

A mOSAIC Application, as described above, is composed of a set of distributed components, each of them consuming resources, which cooperate trough messages exchange on message queue Cloud resources. mOSAIC Application are offered *as a service* to their users adopting dedicated components, which act as a gateway from external access to internal requests. An example of such components is the *HTTPgw*, which accepts HTTP requests from Internet and forward them on internal queues. mOSAIC components process such kind of messages preparing the replies, which are then forwarded to the HTTP gateway.

A very simple example of a mOSAIC Application is an application that receives messages from Internet containing an XML document and parses them for syntax validation. Figure 3 (side a) briefly summarizes the approach. This is a very simple example of an application which manages XML documents.

Note that even if the application described is a toy application, its behavior in terms of components interaction is very typical for Web application. It could be the basis for an XML document manager, which stores the received documents, classifies them and offers searching features.

Every Cloud application of this kind suffers of the typical XML DoS attacks, like the Deeply-Nested XML attack. Such an attack is based on submission of valid XML files with a great number of nested tags, which lead the XML validator to consume great amount of memory and CPU. Thanks to the flexibility of the mOSAIC approach, application is able to face the incremented workload due to the attack, acquiring new Cloud resources, in order to remain available to other service requests. As a side effect, the application consumes an increased amount of resource, on charge of the application owner. In other words the application owner has to face a clear trade-off: increased cost of the application vs reduced availability of the services offered.

The approach that we propose to face such kind of problems, is to offer dedicated mOSAIC components, which are able, on the basis of system monitoring, to adapt the behavior of the application, in order to reduce the effect of the attack, providing an acceptable availability, but with a reduced cost.

The proposed solution consists of adding an *ITSprotector* component in the communication between the HTTPgw and the application components. This additional component acts as filter, cutting away attack messages. As shown in Figure 3 (side b), the mOSAIC User does not have to modify the already developed application, but can simply introduce the

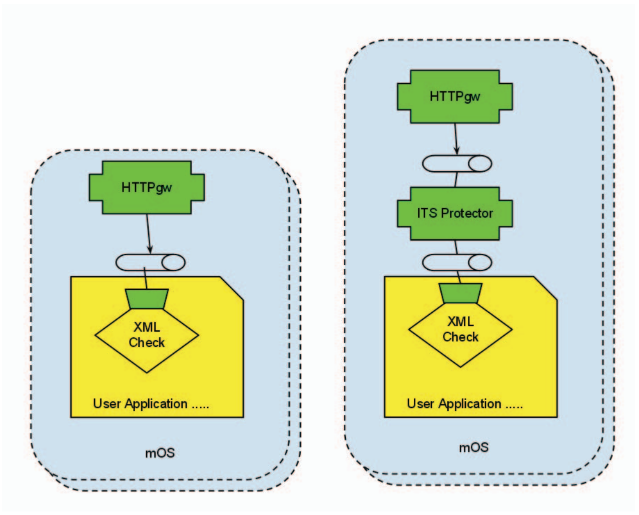

Figure 3: mOSAIC monitoring components architecture

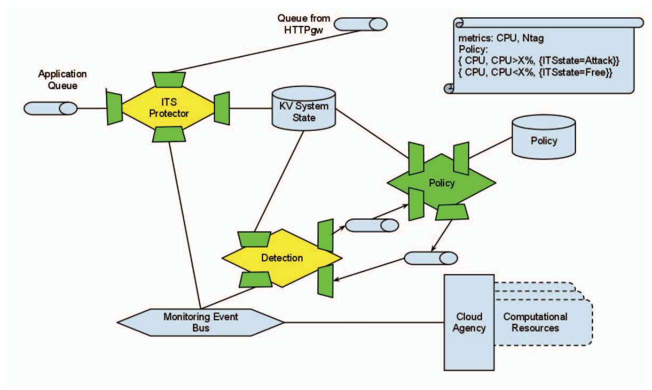

Figure 4: ITSprotector: the Intrusion tolerant component

new component in the middle of the communication. This approach can be used for all the attacks which can be faced trough a filtering system.

In the following Section, we will present an example of such component dedicated to Deeply-Nested XML attacks, showing how it can be integrated into the application and how it relates with the mOSAIC monitoring system.

V. ITSprotector: mOSAIC Componentents for Intrusion Tolerance

As described in Section IV, the proposed approach is to offer a component that acts as a filter, cutting away the attack messages that it is able to detect. Note that such a component has a behavior which depends on the global state of the application, so it needs to monitor the application evolution in order to identify the presence of attacks.

Figure 4 summarizes the architecture of the ITS filters. The lower side of the picture contains the components dedicated to monitor of the application evolution: the *Monitoring Event Bus* is adopted to share the information, the *Cloud Agency* collects data from the set of VMs (as an example CPU and memory usage), and the *Detection Cloudlet* collects data in order to maintain a global representation of the system state. In order to detect the attack, *ITSprotector Cloudlet* monitors the application requests and sends an event on the Monitoring Event Bus for each message received by the application. For example, for the Deeply-Nested XML attack, the ITSprotector sends out a message containing the number of nested tag, for each message received.

The Detection Cloudlet collects events from the Monitoring Event Bus and stores physical monitoring parameters in a shared KV store (KV System State in Figure 4). Moreover, it sends out messages to the *Policy component*, which applying the detection policy returns back a message when an attack condition is detected. The Detection Cloudlet updates the KV System State store with the attack state and the messages received from the Policy component.

The core of the solution is the ITSprotector, which applies the filtering conditions. The ITSprotector receives all messages that are directed to the application, and analyzes the messages in order to distinguish between attack and non-attack messages, following the approach described in section II-A

VI. An Example of ITS Components usage

In order to clarify how the proposed approach works, in this Section we briefly illustrate the adoption of the ITSprotector in a really simple case study. The numerical results here proposed are not original and of little interest for this paper, they are just proposed to clarify how the system works. In order to deeply study how the proposed filtering methodologies works, a detailed analysis is proposed in [8].

The application on which we focus is a simple XML checker. As Figure 3 shows, the ITS protector is added between the HTTPgw and the XML checker. The attack we aims to solve is the Deeply-Nested XML attack, which we detect and filter with the methodology summarized in II-A.

During the application execution, the application just receives messages from the Final Users. When an attack takes place, the CPU consumption increases abnormally. The typical reply for a Cloud application is to acquire new resources, in order to face the improved workload, but, as outlined in the previous sections, this has the side effect of increasing the cost of the application, due to additional resource consumption. When the ITSprotector is adopted, the monitoring system detects the presence of attacks, updates the system state, and actives the appropriate recovery action, avoiding malicious CPU consumption.

An example of the recovery effects is shown in Figure 5. It represents the number of application requests processed and the CPU variations with respect to the time, during an interval time of three minutes. The experiment consists of three temporal windows. During the first two windows the ITSprotector is disabled. In particular, the first 60 seconds show the number of requests processed and the CPU load in absence of the attack. The second 60 seconds show the intrusion effects ($i.e.$, the CPU load is about 100% and the number of requests processed is very low). Finally, during the last 60 seconds, the ITSprotector is enabled. When the Policy component detects the condition of a reaction ($i.e.$, the attack is in progress and the CPU consumption exceed the 90%), the ITSprotector filters all the suspicious messages. The reaction is applied until the CPU load falls below the 90%. On the other hand, since the attack is already in progress, the system's CPU resources are again exhausted by processing of new malevolous messages (the CPU consumption after a few seconds increasing again), therefore, the Policy component triggers a new reaction.

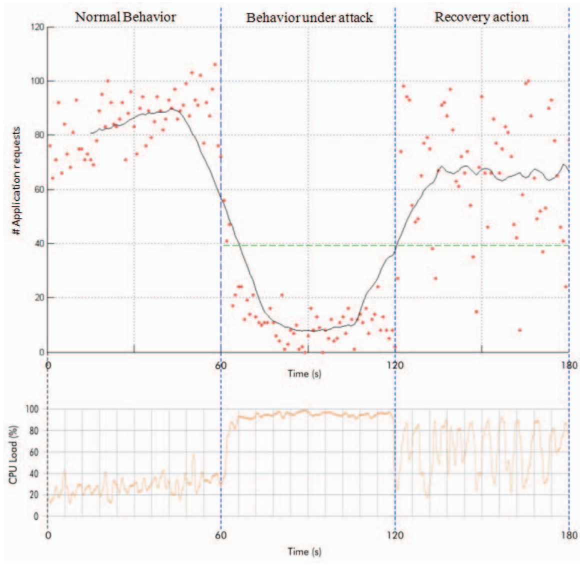

Figure 5: System evaluation during a recovery process

VII. Conclusions and Future Works

In this paper, we have shown how it is possible to enrich a mOSAIC-based Cloud Application in order to enable protection from a given set of attacks. We focused on a single kind of attacks (XML DoS Nested tags) in order to prove the validity of the approach, which can be generalized to a large set of different solutions. It is important to note that the approach proposed can be applied to already developed mOSAIC Cloud Application, without any kind of real changes on the original application code: in order to protect and existing application, it is needed just to stop it, apply the reconfiguration ($i.e.$, changes the connection to the consumed resources) and restart the application.

The work presented in this paper represent a first step in the direction of offering a complex set of Intrusion Tolerance features on the top of the mOSAIC Framework. In future works we will enrich the proposed components in order to offer protection to a larger set of attacks. Moreover, we will integrate the proposed solution with the SLA-related components, in order to offer protection from attacks in terms of Service Level Agreements. We will focus on a detailed evaluation of the overhead introduced from the protection components, offering clear indication in order to manage the trade-off between the additional cost due to overhead even in normal conditions and the costs due to the presence of an attack.

References

[1] mOSAIC Project, "mosaic: Open source api and platform for multiple clouds," http://www.mosaic-cloud.eu, 2010.

[2] CONTRAIL, "Contrail: Open computing infrastructres for elastic computing," http://contrail-project.eu/, 2010.

[3] "Optimis: the clouds silver lining," [Online]. Available:http://www.optimis-project.eu/, 2010.

[4] M. Jensen and N. Gruschka, "Flooding attack issues of web services and service-oriented architectures,," in *GI Jahrestagung*, ser. LNI, vol. 133, 2008, pp. 117–122.

[5] R. H. M. Jensen, N. Gruschka and N. Luttenberger, "Soa and web services: New technologies, new standards - new attacks," *European Conf. on Web Services,*, pp. 35–44, 2007.

[6] "Snort: an open source network intrusion prevention and detection system (ids/ips)," 2010. [Online]. Available: http://www.snort.org/

[7] B. J. Min and J. S. Choi, "An approach to intrusion tolerance for mission-critical services using adaptability and diverse replication," *Future Gener. Comput. Syst.*, vol. 20, pp. 303–313, Feb. 2004.

[8] M. Ficco and M. Rak, "Intrusion tolerant approach for denial of service attacks to web services," in *the 1st International Conference on Data Compression, Communications and Processing (CCP 2011)*, 2011, pp. 285–292.

[9] A. K. Caglayan and D. E. Eckhardt, "A theoretical investigation of generalized voters for redundant system," in *The Nineteenth International Symposium on Fault-Tolerant Computing*, 1989, pp. 444–451.

[10] A. P. Kouznetsov and P. Druschel, "The case for byzantine fault detection," in *The 2nd Workshop on Hot Topics in System Dependability*, 2006.

[11] P. Sousa and P. Verissimo, "Proactive resilience through architectural hybridization," in *The ACM Symp. on Applied-Computing (SAC'06)*, 2006.

[12] J. D. Heimbigner and A. Wolf, "The willow architecture: Comprehensive survivability for large-scale distributed applications," in *The Intrusion Tolerant System Workshop*, 2002, pp. 71–78.

[13] C. Craciun, M. Rak, and D. Petcu, "Towards a cross platform cloud api - components for cloud federation," in *1st International Conference on Cloud Computing and Services Science (CLOSER 2011)*, 2011, pp. 166–169.

[14] C. Craciun, M. Neagul, I. Lazcanotegui, M. Rak, and D. Petcu, "Building an interoperability api for sky computing," in *The Second International Workshop on Cloud Computing Interoperability and Services*, IEEE, Ed., 2011.

[15] M. Rak, S. Venticinque, R. Aversa, and B. Di Martino, "User centric service level management in mosaic application," in *Europar 2011 Workshop*, I. Press, Ed., 2011.

[16] M. Rak, L. Liccardo, and R. Aversa, "A sla-based interface for security management in cloud and grid integrations," in *Proceedings of the 2011 7th International Conference on Information Assurance and Security (IAS)*, A. Abraham *et al.*, Eds. IEEE Press, 2011.

[17] M. Rak, S. Venticinque, T. Mhr, G. Echevarria, and G. Esnal, "Cloud application monitoring: The mosaic approach," *Cloud Computing Technology and Science, IEEE International Conference on*, vol. 0, pp. 758–763, 2011.