A Trustworthy and Resilient Event Broker for Monitoring Cloud Infrastructures

Diego Kreutz, António Casimiro, and Marcelo Pasin

LaSIGE, Faculty of Sciences, University of Lisbon, Portugal
kreutz@lasige.di.fc.ul.pt, {casim,pasin}@di.fc.ul.pt

Abstract. We propose a fault- and intrusion-tolerant framework for datacenter and cloud infrastructure monitoring. In contrast to existing approaches, our framework is able to deal with Byzantine faults. It is based on a replicated event broker, lying in the core of the monitoring infrastructure, supporting the dissemination of all monitoring events. We describe the architecture and the behavior of the framework, explaining how we can deal with different requirements on QoS and QoP. We provide evaluation results as proof of concept for the proposed framework.

1 Introduction

Infrastructure security is a serious concern in cloud computing [1–3]. Cloud providers use various tools to monitor and control the behavior of their computing environment. These tools are fundamental to discover, diagnose and foresee problems, and then react whenever necessary to fix or avoid those problems, rendering security information event managers and cloud infrastructure monitoring systems especially relevant.

From a dependability perspective, relying on a single tool for system monitoring is not a good idea. A single vulnerability be exploited to affect the normal operation of the overall infrastructure. Given the criticality of the services being monitored, it is advisable to deploy multiple monitoring tools, creating the necessary diversity to reduce the number of common faults, and relying on redundancy to ensure that the monitoring system itself (and consequently the cloud environment) will resists to faults and intrusions.

In this paper we propose a fault and intrusion tolerant (FIT) framework to support a trustworthy, flexible and efficient dissemination of monitoring information and thus facilitate the deployment of redundancy in monitoring. The framework handles event messages produced by monitoring probes, delivering them to all interested monitoring consoles, and doing so in a resilient and trustworthy way. One fundamental contribution of our work is that we consider Byzantine (intentional and malicious) faults in the fault model, and provide the mechanisms and protocols to handle these faults. To the best of our knowledge, we are the first to provide a framework for Byzantine fault tolerant event dissemination.

In our design we strive to provide simple interfaces that can be easily used in existing and in new applications. One application example could be a redundant ArchSight security information event manager (SIEM) [4], where each

K.M. Göschka and S. Haridi (Eds.): DAIS 2012, LNCS 7272, pp. 87–95, 2012.

replica receives all the events from all probes to ensure that a correct monitoring view is still possible even if one of the instances is compromised. We resort to replication techniques in the design of the dissemination framework to achieve trustworthiness and ensure that tampering with a single flow of event messages will not be enough to affect the system correctness, as it could happen with a non-replicated solution.

Following this introduction, we have a section to review related work. Next, we present the design and architecture of the proposed framework. After that we detail the proof of concept prototype, present evaluation results and conclude with final remarks.

2 Background and Related Work

Recent surveys show that there are different threats in cloud infrastructure-as-a-service [3, 2]. Seven threats were already identified and characterized [1]: (1) abuse and nefarious use of cloud computing; (2) insecure interfaces and APIs; (3) malicious insiders; (4) shared technology issues; (5) data loss or leakage; (6) account or service hijacking; and (7) unknown security profile. Those threats go from the hardware level to the top level software.

Security information event managers are able to process and correlate security logs and events generated by different and distributed sources. Events are generated by probes or by agents that are distributed, attached to most system components (devices, services, entire systems) to collect security related data. These events are propagated to a SIEM engine for processing, analysis, visualization, archival and possibly also automated reactive actions.

Some of the most widely used and known monitoring tools for cloud computing infrastructures [4] include Amazon CloudWatch, VMware vFabric Hyperic and LogicMonitor. Most of the tools target the basic health of the system, such as workloads, network traffic, components availability and general system state. They essentially provide monitoring data for further processing, using their own agents to gather data from the infrastructure. As far as we understand them, none is capable of gathering and providing baseline data that is enough to address all types of threats, attacks and vulnerabilities found in cloud infrastructures-as-a-service. Among all analyzed tools, CloudSec, PCMONS and Middleware for Assured Cloud present some interesting characteristics worth being noted. CloudSec [5], differently from the other tools, addresses only security problems at the virtual machine level. On the other hand, PCMONS [6] was designed for the integration of existing open source tools, such as Nagios, to address the problem of monitoring the IaaS. However, it is not clear how this integration is effectively done. Moreover, it seems to be costly to write extensions to different kinds of tools to be used in the PCMONS architecture. One work that effectively starts to address the problem of securing cloud infrastructure monitoring systems is the Middleware for Assured Cloud [7]. Their basic idea is to distribute the monitoring system across the domain and use information redundancy to provide assured monitoring data.

Discussion. There is no single solution capable of covering all real and existing security threats in IaaS environments. There are many security information events managers and monitoring systems, as well as other smaller and more specific solutions, which can help to improve the overall infrastructure security.

We envision an environment based on the growing use of diverse and heterogeneous solutions. With that in mind, we intend to provide a resilient and trustworthy framework to assure the transport of monitoring events from probes to consoles. Probes can be of any kind, including existing systems. Similarly, consoles can be of any kind, from traditional screen consoles to advanced event correlation engines. Furthermore, a single monitoring event could be delivered to multiple consoles at the same time. This allows for improved Quality of Protection (QoP), while permit different and richer analysis at the same time. Two different SIEMs, for instance, may detect more security threats and vulnerabilities than just one.

Finally, as far as we know, besides the Middleware for Assured Cloud [7] that uses data redundancy for fault tolerance, most of the existing SIEMs and monitoring solutions do not offer fault and intrusion tolerance in their design. This means that their single communication channels between probes and backend monitoring system itself (or consoles) can be compromised by a single attack. Clearly, existing solutions do not offer support for Byzantine fault tolerance, which is one of the basic building blocks to achieve a resilient and trustworthy event message transport.

3 FIT Event Broker

This section contains an architectural description of our FIT event broker. It is a service that provides a reliable and trustworthy communication layer for transporting event messages from probes to consoles using replication. Each event broker replica is like a state machine, and all correct replicas have the same state. For increased QoP, each replica can be implemented using diversity techniques, such as different operating systems.

3.1 Fundamental Assumptions

Network Model. We assume a fully connected network, following a TCP/IP model. Thus, all probes and consoles can reach all broker replicas.

Synchrony Model. We assume a partially synchronous model like the timed asynchronous model [8]. In this model systems can make progress when there is enough synchrony to detect omissions and performance failures, allowing problems such as consensus, membership and leader election to be solved. Most computing systems, such as the ones we consider, have high-precision quartz clocks that render the assumption reasonable enough.

Fault Model. We consider that two fault models can be assumed, and provide solutions adequate to both: one that includes only accidental faults and anther

that includes both accidental and malicious faults. In the first model, the system can tolerate f accidental faults if there is at least one service replica forwarding event messages. In the second, the system can tolerate f Byzantine faults if it employs integrity or confidentiality facilities and there are enough replicas forwarding event messages to be voted by subscribers.

3.2 System Requirements

We consider functional and non-functional requirements for our FIT event broker. From a functional perspective we need to: (a) Ensure the correct event forwarding from probes to consoles, in a decoupled mode using replication mechanisms. We opted for a publish/subscribe model for communication and event handling [9], where we provide a topic-based message routing; (b) Support diverse requirements for Quality of Service (QoS) and Quality of Protection (QoP). There are various metrics for both QoS and QoP [10, 11], so in our work we focus on the following. Regarding QoS we provide support for the specification of message urgency, ordering and persistence requirements. Urgent messages imply the notion of priority among events being handled by the FIT broker. When ordering is required, additional algorithms need to be used, with implications on performance. Persistence can be achieved through dynamic queues, with upper bounds for persistence requirements configured by clients. Regarding QoP, requirements for simple crash fault tolerance (CFT) and Byzantine fault tolerance (BFT) are supported and selectable by clients. BFT provides increased QoP at the possible cost of performance, since BFT state-machine replication has to be used in stricter cases, such as when registering channel subscribers to multiple replicas or when several SIEM (subscribers) need to receive the events in the same strict order. In addition to CFT and BFT requirements, integrity and confidentiality of events can also be user-defined, and is supported through various techniques, such as simple identification (SI), hash functions (HA), public-key signatures (PKS) and public-key signature with encryption (PKE).

From a non-functional perspective, we consider the following fundamental requirements: (a) Reasonable performance for the provided QoP. The FIT broker overhead must be acceptable for each selected operational mode. In particular, when basic QoP (CFT) is required, it should be possible to achieve performance levels compatible to baseline event broker systems. We achieve this requirement using a modular design that separates concerns within the FIT broker; (b) Easy integration in existing environments. We assume that existent monitoring tools can have their event messages encapsulated for transferring. Such datagram encapsulation can be transparently provided by our broker. End applications need only to use the broker's interfaces to transport event messages.

3.3 Architectural Components

Publishers and subscribers are both clients of our event broker. A probe is an example of a publisher that provides messages related to security events as, for

example, behavioral changes on network and processing resources usage, or security threats. A console is an example of a subscriber that receives event messages and presents monitoring information or alerts to system administrators.

The FIT event broker can be decomposed into three communication layers, as presented in Figure 1. The first is the event broker interface (L1), which provides the basic primitives for the interaction between clients and the broker service.

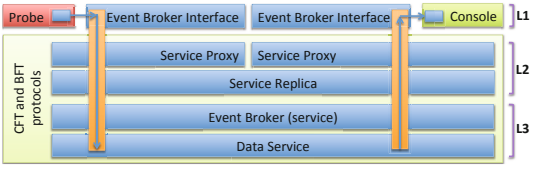

Fig. 1. System communication layers

The second layer of our FIT broker contains the service proxy and service replica (L2). It is used to establish the connection between a client and service replicas, as well as to transport events and to invoke request actions in replicas. The lower layer is the event broker service algorithm (L3), which provides channel control, event management and routing.

Channels are event streams where event messages are published, managed and routed to the final subscriber queues. They are created by service replicas through configuration properties and are managed based on a control table. Such control table contains, for each channel, a list of authorized clients (publishers and subscribers), a topic and a QoS and QoP class. The event message flow, from publishers to subscribers, is presented in Figure 2.

Event messages are created and sent by publishers and are inserted in channels. Once a message is in a channel, it is filtered and routed to output queues associated with clients accordingly to the control table. Each subscriber has its own output queue.

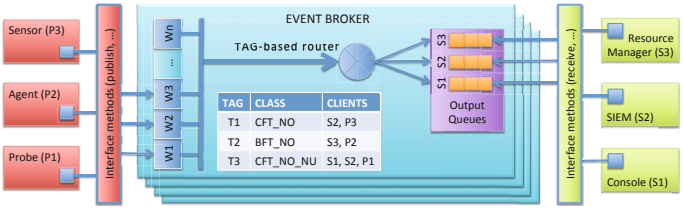

Fig. 2. Broker internal components and event message flow

Some published events require immediate effect, as for example, alarms for system administrators. Others need total order deliver within a channel or among different channels, as for example, for root cause correlation. QoS and QoP classes are provided in order to support such diverse use cases of our tool.

Channels and output queues have limited size, which can temporarily maintain a certain amount of information. After a predefined time to live, oldest events start to be discarded if subscribers do not process events as fast as publishers create them. This mechanism can also work as a buffer for slow subscribers to deal with event bursts and to improve event message delivery reliability.

Operating under CFT requirements and using CFT protocols implies publishing messages in all service broker replicas and subscribers receiving the event message from the fastest service replica. In this case, there is no communication among replicas and the service is available if at least one replica forwards the message to subscribers.

When the Byzantine fault tolerant protocol is employed, it allows to the service tolerate even malicious faults. If ordering is required, we opted for using BFT-SMaRt [12], which implements a BFT protocol with state machine replication. The entire control table becomes part of the state in this case, given that all replicas have to agree on the list of publishers and subscribers. A voting mechanism is provided on the subscriber side to receive message from all broker replicas and verify which is the correct answer.

4 Implementation and Evaluation

We developed a first prototype for the FIT event broker as a proof of concept. Now we describe the main building components, use case scenario and results.

Main Components

Basic data units. The implementation is based on the architectural design. The main data units are represented by events, requests and channels. An event basically contains: the publisher id and signature (if it applies), the event id, a clock-base timestamp and the content, which represents monitoring data and information. An event is uniquely identified in the FIT service by composing the sender id with the event id. The clock-based timestamp is used to verify the event time to live: once expired, the event is dropped. This can happen in the service replicas or in the receiver's local event buffer used by the BFT voter. A request, which is used between the event broker client and service, contains a channel tag, a method to be invoked on the service broker, a sequence number, the number of events to fetch, the events (in the case of a receive method invoked by a client) and the operation status. This status describes for the client application what happened with the requested on the server side. The channel contains a tag (or id), a class (specifying QoS parameters), a map of subscribers to output queues, and a list of publishers. Each channel is an independent data unit that can be accessed, both for read and write, in parallel.

Event Broker. The event broker interface implements all common methods of a publish-subscribe system, such as register, subscribe, publish and receive. When a client invokes one method, a request is automatically created and sent to all service replicas. Each service replica receives the request and delivers it to the event broker service, invoking the corresponding method defined in the request.

For instance, if it is a receive method, the event broker service will get the requested events from the corresponding output queue and send them to the caller client.

Service Proxy and Replica. The service proxy encapsulates all the communication between clients and replicas. It only provides a simple invoke interface (`Reply invoke(Request)`) to the event broker with a request to be sent to all service replicas. Each replica is going to send back an appropriate response, according to what has been requested by the client.

Results and Evaluation

Our evaluation goal was to verify if the requirements for a specific use case scenario [4] are fullfiled. The environment is composed of event aggregators with a total throughput of 30,000 events per seconds sent to ArcSight. The throughput scales according to the number of SIEM instances. As an example, we need 60,000, 90,000 and 120,000 events per second for 2, 3 and 4 SIEMs, respectively.

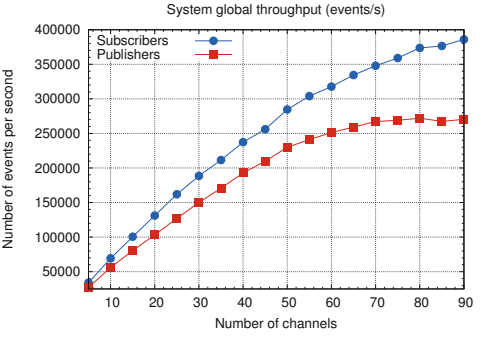

Fig. 3. Throughput for up to 90 channels

Environment and basic info. To test our software, we used 4 x86-based computers, with 2 quad-core CPUs supporting up to 16 threads in parallel. Each computer has 32GB of RAM and Gigabit Ethernet interfaces, connected to a Gigabit switch. Two computers where used to run the replicas and the other two to run the clients. To test BFT throughput, we used 4 replicas, 2 per computer.

Byzantine fault tolerance throughput. We present some results achieved for BFT without ordering. As we have four replicas, we can tolerate up to one faulty replica. On Figure 3 we can see the measured throughput from 5 to 90 channels. With 50 channels we achieve a publishing throughput of over 200,000 events per second. At the same time, the throughput of the subscribers gets close to 300,000 events per second. This difference resides mainly on three facts: (1) the publishers have to collect data and generate the events; (2) the publishers use blocking methods, which means that they wait for the service replica to answer before going on; (3) buffer contention has not been measured, thus,

subscribers can actually be faster than publishers. Otherwise, with buffer contention, subscribers should have equal or less performance than publishers. Furthermore, the publisher throughput stabilization for more than 70 channels is due to resources exhaustion. With more resources we would be able to keep increasing the throughput because the channels are managed independently.

5 Conclusion and Future Work

In this paper we proposed a trustworthy and resilient event broker middleware for monitoring cloud infrastructures. The architecture supports different levels of quality of service and quality of protection. It is designed to allow the use of existing probes, consoles, SIEMs, monitoring tools, management engines, among other systems. The idea is to provide a resilient and trustworthy event broker and, at the same time, facilitate the use of multiple tools, allowing broader threat analysis coverage.

The first evaluation results focused on basic requirements of a real use case, considering a throughput of 30,000 events per second. We showed that the first prototype is able to deliver more than 50,000 events per second with 10 channels. It is also capable of delivering more than 250,000 events per second with 55 or more channels.

Acknowledgements. This work is supported by the EC, through project SecFuNet FP7-ICT-STREP-288349 and by FCT, through project TRONE CMU-PT/RNQ/0015/2009.

References

1. Vaquero, L., Rodero-Merino, L., Morn, D.: Locking the sky: a survey on iaas cloud security. Computing 91, 93–118 (2011)
2. Takabi, H., Joshi, J., Ahn, G.: Security and privacy challenges in cloud computing environments. IEEE Security Privacy 8(6), 24–31 (2010)
3. Mansfield-Devine, S.: Danger in the clouds. Network Security (12), 9–11 (2008)
4. Padhy, S., Kreutz, D., Casimiro, A., Pasin, M.: TRONE - First Specification of the Architecture. Technical report, FCUL (October 2011), `http://trone.di.fc.ul.pt`
5. Ibrahim, A.S., et al.: Cloudsec: a security monitoring appliance for virtual machines in the iaas cloud model. In: Proceedings of the 5th International Conference on Network and System Security. IEEE (2011)
6. De Chaves, S., Uriarte, R., Westphall, C.: Toward an architecture for monitoring private clouds. IEEE Communications Magazine 49(12), 130–137 (2011)
7. Campbell, R.H., Montanari, M., Farivar, R.: A middleware for assured clouds. Journal of Internet Services and Applications (December 2011)
8. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems 10, 642–657 (1999)
9. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Survey 35, 114–131 (2003)

10. Wang, H., Liao, C., Tian, Z.: Providing quality of service over time delay networks by efficient queue management. In: 2011 IEEE 36th Conference on Local Computer Networks (LCN), pp. 275–278 (October 2011)
11. Foley, S.N., et al.: Multilevel Security and Quality of Protection. In: Gollmann, D., Massacci, F., Yautsiukhin, A. (eds.) Quality of Protection. Advances in Information Security, vol. 23, pp. 93–105. Springer, US (2006)
12. Bessani, A.N., et al.: BFT-SMaRt - High-performance Byzantine-Fault-Tolerant State Machine Replication (2011), `http://code.google.com/p/bft-smart/`