Integrated Monitoring of Infrastructures and Applications in Cloud Environments

Roberto Palmieri*, Pierangelo di Sanzo*, Francesco Quaglia*, Paolo Romano†, Sebastiano Peluso†, and Diego Didona†

Dipartimento di Informatica e Sistemistica, Sapienza Rome University, Italy(*);Distributed Systems Group, INESC-ID, Lisbon, Portugal(†)

Abstract. One approach to fully exploit the potential of Cloud technologies consists in leveraging on the Autonomic Computing paradigm. It could be exploited in order to put in place reconfiguration strategies spanning the whole protocol stack, starting from the infrastructure and then going up to platform/application level protocols. On the other hand, the very base for the design and development of Cloud oriented Autonomic Managers is represented by monitoring sub-systems, able to provide audit data related to any layer within the stack. In this article we present the approach that has been taken while designing and implementing the monitoring sub-system for the Cloud-TM FP7 project, which is aimed at realizing a self-adapting, Cloud based middleware platform providing transactional data access to generic customer applications.

1 Introduction

As well known, Cloud based technologies are making a revolutionary change in the way systems and applications are built, configured and run. In particular, the ability to acquire computational power and storage on-the-fly has opened the possibility to massively put in place Autonomic Management schemes aimed at optimizing performance/availability indexes vs specific cost metrics.

A relevant reflection of such a revolutionary change is in that several projects targeting Cloud oriented software platforms and applications aim at designing/integrating multi-modal operating modes. In particular, the target is to make differentiated protocols coexist within both the platform and the application layer in order to dynamically select the best suited protocol (and well suited parameter settings for it) depending on specific environmental conditions, such as the current workload profile. Consequently, the need arises for defining/implementing frameworks and systems supporting audit and monitoring functionalities spanning the whole set of differentiated layers within the Cloud based system.

At current date, several proposals exist in the context of monitoring the usage of infrastructure level resources (e.g. CPU and RAM) [1]. These are mostly suited for Infrastructure-as-a-Service (IaaS) customers, to whom the possibility to trigger infrastructure level reconfigurations either automatically or on demand, based on the monitoring outcomes, is provided. On the other hand, Cloud providers offer the possibility to monitor the level of performance provided by specific, supported platforms [4], such as Web based platforms, in order to enable, e.g., auto-scale facilities aimed at dynamically resizing the offered computational platform. This is suited for Platform-as-a-Service (PaaS) customers, who aim at delivering specific performance levels, while relying on facilities already offered by their reference Cloud providers.

In this paper we describe the approach we have taken in the design/development of a Workload and Performance Monitor (WPM) that provides audit data for both infrastructure resources and platform (or application) level components in an integrated manner. The main distinguishing feature of our solution is that it does not target any specific platform or application. Instead, it is flexible and adaptable so to allow integration with differentiated platform/application types. On the technological side, our design comes from the integration of the Lattice framework (natively oriented to infrastructure monitoring), which has been largely exploited in the context of the RESERVOIR project [2], and the JMX JAVA oriented framework (suited for the audit of JAVA based components). The whole design/implementation has been tailored for integration within the platform targeted by the Cloud-TM FP7 project [3]. This project aims at designing/developing a self-adaptive middleware level platform, based on the Infinispan in-memory data management layer [6], providing transactional data access services (according to agreed upon QoS vs cost constraints) to the overlying customer applications.

2 Technological Background

2.1 The Lattice framework

Lattice relies on a reduced number of interacting components, each one devoted (and encapsulating) a specific task in relation to distributed data-gathering activities. In terms of interaction abstraction, the Lattice framework is based on the producer-consumer scheme, where both the producer and consumer components are, in their turn, formed by sub-components, whose instantiation ultimately determines the functionalities of the implemented monitoring system. A producer contains data sources which,

in turn, contain one or more probes. Probes read data values to be monitored, encapsulate measures within measurement messages and put them into message queues. Data values can be read by probes periodically, or as a consequence of some event. A message queue is shared by the data source and the contained probes. When a measurement message is available within some queue, the data source sends it to the consumer, which makes it available to reporter components. Overall, the producer component injects data that are delivered to the consumer. Also, producer and consumer have the capability to interact in order to internally (re)configure their operating mode.

Three logical channels are defined for the interaction between the two components, named

- data plane;
- info plane;
- control plane.

The data plane is used to transfer data-messages, whose payload is a set of measures, each kept within a proper message-field. The structure of the message (in terms of amount of fields, and meaning of each field) is predetermined. Hence, message-fields do not need to be explicitly tagged so that only data-values are really transmitted, together with a concise header tagging the message with very basic information, mostly related to source identification and timestamping. Such a structure can be anyway dynamically reconfigured via interactions supported by the info plane. This is a very relevant feature of Lattice since it allows minimal message footprint for (frequently) exchanged data-messages, while still enabling maximal flexibility, in terms of on-the-fly (infrequent) reconfiguration of the monitoring-information structure exchanged across the distributed components within the monitoring architecture.

Finally, the control plane can be used for triggering reconfiguration of the producer component, e.g., by inducing a change of the rate at which measurements need to be taken. Notably, the actual transport mechanism supporting the planes is decoupled from the internal architecture of producer/consumer components. Specifically, data are disseminated across these components through configurable distribution mechanisms ranging from IP multicast to publish/subscribe systems, which can be selected on the basis of the actual deployment and which can even be changed over time without affecting other components, in term of their internal configuration. The framework is designed to support multiple producers and multiple consumers, providing the chance to dynamically manage data

source configuration, probe-activation/deactivation, data sending rate, redundancy and so on.

2.2 Portability issues

The Lattice framework is based on JAVA technology, so that producer/consumer components encapsulate sub-components that are mapped onto a set of JAVA threads, each one taking care of specific activities. Some of these threads, such as the data-source or the data-consumer, constitute the general purpose backbone of the skeleton provided by Lattice. Other threads, most notably the probe-thread and the reporter-thread, implement the actual logic for taking/reporting measurement samples. The implementation of these threads can be seen as the ad-hoc portion of the whole monitoring infrastructure, which performs activities tailored to specific measurements to be taken, in relation to the context where the monitoring system operates.

By the reliance on JAVA, portability issues are mostly limited to the implementation of the ad-hoc components. As an example, a probe-thread based on direct access to the "proc" file system for gathering CPU/memory usage information is portable only across (virtualized) operating systems supporting that type of file system (e.g. LINUX). However, widening portability across general platforms would only entail reprogramming the internal logic of this probe, which in some cases can even be done by exploiting, e.g., pre-existing JAVA packages providing platform-transparent access to physical resource usage.

The aforementioned portability considerations also apply to reporter-threads, which can implement differentiated, portable logics for exposing data to back-end applications (e.g. by implementing logics that store the data within a conventional database).

3 Architectural Organization

Figure 1 shows the general architectural organization we have devised for WPM. It has been defined according to the need for supporting the following two main functionalities:

- statistical data gathering (SDG);
- statistical data logging (SDL).

The SDG functionality maps onto an instantiation of the Lattice framework. In our instantiation, the elements belonging to the monitored

infrastructure, such as Virtual Machines (VMs), can be logically grouped, and each group will entail per-machine probes targeting two types of resources: (A) hardware/virtualized and (B) logical. Statistics for the first kind of resources are directly collected over the Operating System (OS), or via OS decoupled libraries, while statistics related to logical resources (e.g. the data-platform) are collected at the application level by relying on the JMX framework for JAVA components.

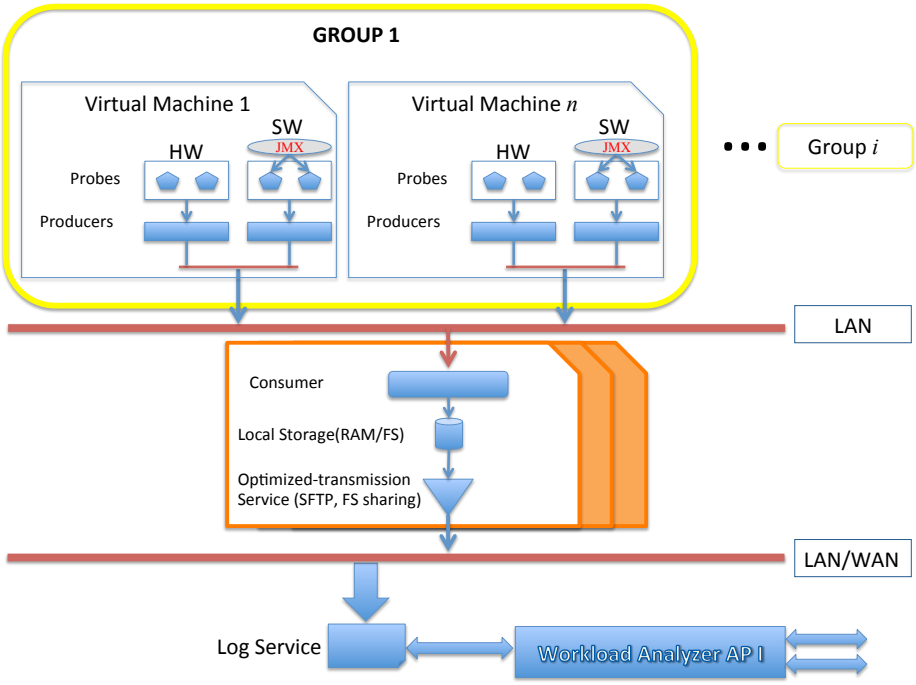

Fig. 1. WPM Architectural Organization.

The data collected by the probes are sent to the producer component via the facilities natively offered by the Lattice framework. Each producer is coupled with one or many probes and it is responsible of managing them. The consumer is the Lattice component that receives the data from the producers, via differentiated messaging implementations, which could be selected on the basis of the specific system deployment. We envisage a LAN based clustering scheme such that the consumer is in charge of handling one or multiple groups of machines belonging to the same LAN. Anyway, in our architectural organization, the number of consumers is not

meant to be fixed, instead it can be scaled up/down depending on the amount of instantiated probes/producers. Overall, the consumer can be instantiated as a centralized or a distributed process. Beyond collecting data from the producers, the consumer is also in charge of performing a local elaboration aimed at producing a suited stream representation to be provided as the input to the Log Service, which is in turn in charge of supporting the SDL functionality.

We decided to exploit the file system locally available at the consumer side to temporarily keep the stream instances to be sent towards the Log Service. The functional block which is responsible for the interaction between SDG and SDL is the so called *optimized-transmission service*. This can rely on top of differentiated solutions depending on whether the instance of SDL is co-located with the consumer or resides on a remote network. Generally speaking, with our organization we can exploit, e.g., SFTP or a locally shared File System. Also, stream compression schemes can be actuated to optimize both latency and storage occupancy.

The *Log Service* is the logical component responsible for storing and managing all the gathered data. It must support queries from any external application so to expose the statistical data for subsequent processing/analysis. The Log Service could be implemented in several manners, in terms of both the underlying data storage technology and the selected deployment (centralized vs distributed). As for the first aspect, different solutions could be envisaged in order to optimize access operations depending on, e.g. suited tradeoffs between performance and access flexibility. This is also related with the data model ultimately supported by the Log Service, which might be a traditional relational model or, alternatively, a <key,value> model. Further, the Log Service could maintain the data onto a stable storage support or within volatile memory, for performance vs reliability tradeoffs. The above aspects could depend on the the functionality/architecture of the application that is responsible for analyzing statistical data, which could be designed to be implemented as a geographically distributed process in order to better fit the WPM deployment (hence taking advantage from data partitioning and distributed processing).

3.1 Implementation of infrastructure oriented probes

In this section we provide some technical specification for the probes developed in WPM. The design and the implementation of the infrastructure oriented probes has been tailored to the acquisition of statistical data in relation to (virtualized) hardware resources with no binding on

a specific Operating System. This has been done by implementing the JAVA code associated with the probe on top of the SIGAR cross-platform JAVA based library (version 1.6.4) [5]. Infrastructure oriented probes are in charge of gathering statistical data on

1) CPU (per core): %user, %system, %idle.
2) RAM: kB free memory, kB used memory.
3) Network interfaces: total incoming data bytes, total outgoing data bytes, inbound bandwidth usage, outbound bandwidth usage.
4) Disks: %Free space (kB), %Used space (kB), mountPoint or Volume.

For all of the above four resources, the associated sampling process can be configured with differentiated timeouts whose values can be selected on the basis of the time-granularity according to which the sampled statistical process is expected to exhibit non-negligible changes.

3.2 Implementation of data platform oriented probes

The implementation of the data platform oriented probes has been extensively based on the JMX framework [7], which is explicitly oriented to support audit functionalities for JAVA based components. Essentially, each data platform oriented probe implements a JMX client, which can connect towards the JMX server running within the process where the monitored component resides. Then, via the JMX standard API, the probe retrieves the audit information internally produced by the monitored JAVA component in relation to its own activities. Anyway, the adoption of JMX Framework as a reference technology for implementing application level probes is not necessarily tied to a JAVA component. This is because a generic JMX probe can retrieve data form a JAVA component that wraps any possible monitored application, also written using any programming language.

As an instantiation of application level probes, in our implementation we developed a data platform probe that accesses the internal audit system of single Infinispan [6] caches ([1]), in order to sample some parameters such as the Number of Commit, Number of Rollback, the Commit latency, etc.

3.3 Startup and base message tagging rules

Particular care has been taken in the design of the startup phase of WPM components, in relation to the fact that each probe could be deployed

[1] We recall that Infinispan has been selected as the data layer within the Cloud-TM project, for which WPM constitutes one of the building blocks.

within a highly dynamic environment, where the set of monitored components (either belonging to the infrastructure or to the data platform) and the related instances can vary over time.

As pointed out, WPM will be a part of the Autonomic Manager of the Cloud-TM platform, which will rely on a Repository of Tunable Components where an XML description for each component currently taking part to the Cloud-TM platform is recorded at component startup time. In the design of the WPM we rely on this repository, by exploiting it as a registry, where each probe can automatically retrieve information allowing it to univocally tag each measurement message sent to the Lattice consumer with the identity of the corresponding monitored component instance, as currently maintained by the registry. This will allow supporting a perfect matching between the measurement message and the associated instance of component, as seen by the overall infrastructure at any time instant. Such a process has been supported by embedding within Lattice probes a sensing functionality, allowing the retrieval of basic information related to the environment where the probe is activated (e.g. the IP number of the VM hosting that instance of the probe), which has been coupled with a matching functionality vs the registry in order to both

(a) retrieve the ID of the currently monitored component instance;
(b) retrieve information needed to correctly carry out the monitoring task, in relation to the target component instance.

Such a behavior is shown in Figure 2, where the interaction with the registry is actuated as a query over specific component types, depending on the type of probe issuing the query (an infrastructure oriented probe will query the registry for extracting records associated with VM instances, while a data platform oriented probe will query the registry for extracting records related to the specific component it is in charge of).

As for point (b), data platform probes rely on the use of JMX servers exposed by monitored components. Hence, the information requested to correctly support the statistical data gathering process entails the address (e.g. the port number) associated with the JMX server instance to be contacted. The information associated with point (b) is a "don't care" for infrastructure oriented probes since they do not operate via any intermediary (e.g. JMX server) entity.

3.4 Implementation of the Optimized-transmission Service

In the current implementation, the optimized-transmission service has been implemented by relying on the use of zip and SSL-based file trans-

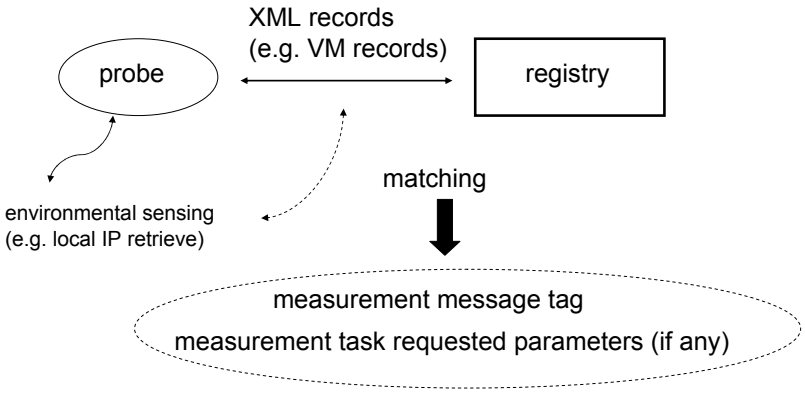

Fig. 2. Interaction between the Probes and the Registry.

fer functionalities. Each data stream portion assembled by the Lattice consumer is locally logged within a file, which is then zipped and sent towards the Log Service front-end via SSL. Exactly-once transmission semantic has been guaranteed via well known retransmission/filtering schemes, which have been based on a univocally determined name for each transmitted zipped file. Specifically, each Lattice consumer is univocally identified via a $consumer\_ID$, which has been used to generate unique file names in the form

$$consumer\_ID + start\_timestamp + end\_timestamp$$

where start and end timestamp values within the file name identify the time interval during which the statistical data have been gathered by the consumer. These timestamp values are determined by exploiting the local clock accessible at the consumer side via the `System.currentTimeMillis()` service.

3.5 Implementation of the Log Service

As for the Cloud-TM data layer, the Log Service has been implemented by still relying on Infinispan [6], specifically by instantiating it as an Infinispan application that parses the input streams received from the Lattice

consumer, and performs put operations on top of an Infinispan cache instance. The keys used for put operations correspond to message tags, as defined by the Lattice producer and its hosted probes. In particular, as explained above, each probe tags measurement messages with the unique ID associated with the monitored component. This ID has been used in our implementation to determine a unique key, to be used for a put operation, formed by:

$$component\_ID + type\_of\_measure + measure\_timestamp$$

where the $type\_of\_measure$ identifies the specific measure carried out for that component (e.g. CPU vs RAM usage in case of a VM component), and the value expressed by $measure\_timestamp$ is again generated via the local clock accessible by the probe instance producing the message. Currently, the Log Service exposes to the external applications the Infinispan native <key,value> API, which does not prevent the possibility of supporting a different API in future releases.

4 Summary

In this article we have presented the architecture and the implementation of a Workload and Performance Monitor to be integrated within the architectural design of the Cloud-TM FP7 project platform. Our monitoring system provides integrated supports for gathering samples related to both hardware/virtualized resources and logical resources. It relies on the integration between the Lattice framework and JMX.

References

1. Stuart Clayman, Alex Galis, Clovis Chapman, Giovanni Toffetti and Luis Rodero-Merino and Luis M. Vaquero and Kenneth Nagin and Benny Rochwerger, Monitoring Service Clouds in the Future Internet. IOS Press, 2010.
2. http://www.reservoir-fp7.eu/index.php?page=open-source-code
3. http://www.cloudtm.eu/
4. http://aws.amazon.com/ec2/
5. http://www.hyperic.com/products/sigar
6. http://www.jboss.org/infinispan
7. Java Management Extensions (JMX) Technology, http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html