Chapter 5
Monitoring in a Multi-cloud Environment

Marco Miglierina and Elisabetta Di Nitto

5.1 Introduction

The Cloud brings velocity to the development and release process of applications, however software systems become complex, distributed on multiple clouds, dynamic and heterogenous, leveraging both PaaS and IaaS resources. In this context, gathering feedback on the health and usage of services becomes really hard with traditional monitoring tools, since they were built for on-premise solutions offering uniform monitoring APIs and under the assumption that the application configuration evolves slowly over time. Still, visibility via monitoring is essential to understand how the application is behaving and to enable automatic remediation features such as the ones offered by MODAClouds.

Tower 4Clouds is a monitoring platform built with multi-cloud applications in mind. It offers a model-based approach that helps the user to focus on abstract concepts when configuring the monitoring activity of complex and heterogeneous applications running on multiple clouds. Configuration is done via a powerful rule language, which allows the user to instruct the platform once, predicating on the model of the application. Within a single rule the user will be able to configure what and how data should be collected, what aggregations should be performed, what condition should be verified and what actions should be executed. Tower 4Clouds is also highly composable. Custom metrics and third party monitoring tools can be easily integrated.

M. Miglierina · E. Di Nitto (✉)
Politecnico di Milano - DEIB, Piazza L. da Vinci 32, 20133 Milano, Italy
e-mail: elisabetta.dinitto@polimi.it

M. Miglierina
e-mail: marco.miglierina@polimi.it

E. Di Nitto et al. (eds.), *Model-Driven Development and Operation of Multi-Cloud Applications*, PoliMI SpringerBriefs,
DOI 10.1007/978-3-319-46031-4_5

5.2 Tower 4Clouds Architecture

In order to address the multi-cloud requirement, we could not rely on the monitoring infrastructure provided by a specific cloud provider. We therefore developed Tower 4Clouds as an open source modular platform. Figure 5.1 depicts the general architecture of the platform.

The core elements of the architecture are *Data Analyzers* which acquire data described as RDF tuples and perform filtering, aggregation, and statistical analyses on them. They receive data from multiple *Data Collectors* that can wrap preexisting monitoring tools. Examples of preexisting tools we managed to integrate with our platform are Sigar and Collectd.

Data Analyzers produce output metrics for *Observers* that subscribe for such data. Observers can be other Data Analyzers or external tools that may support, for instance, visualization of monitoring data or the execution of some application-specific actions in response of the occurrence of some events.

The typical configuration we have experimented with includes a Deterministic Data Analyzer (DDA), in charge of performing filtering and aggregation of data, connected to Observers such as a *Statistical Data Analyzer* (SDA), which executes prediction algorithms on data, Graphite or InfluxDB for storing time series data to be then used by graphing tools such as Grafana. The DDA core is the C-SPARQL engine, a general purpose RDF stream reasoner based on the C-SPARQL language [2], which we exploited to monitor applications [3].

As we anticipated, we are not monitoring static and slowly changing systems. Cloud applications are dynamic therefore we had to provide the platform with the elasticity required to reconfigure and update its internal model according to application changes. Such elasticity is obtained by giving data collectors the responsibility

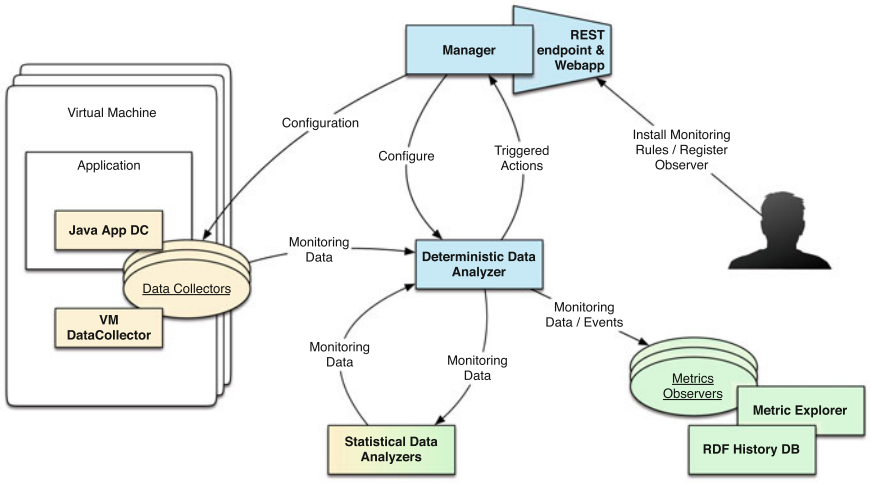

Fig. 5.1 Tower 4Clouds architecture

of registering to the central server (i.e., the Manager in Fig. 5.1) and notifying about resources they are monitoring, avoiding any central discovery mechanism. Collectors are supposed to be available as long as they periodically contact the server. After a predefined period of inactivity, the corresponding monitored resource is removed from the server internal model and considered unavailable.

Finally, we implemented a mono-directional communication protocol used by data collectors to register and to send monitoring data. Since the connection is always from data collectors to the server, there is no need to implement routing strategies and listen to ports at the client side. This allows to have fewer requirements on the XaaS services in charge of hosting the monitored application services.

5.3 Application Configuration Model

Metrics per se are dumb numbers, in order to actually understand where data is coming from and improve visibility, a model of the application is required to give semantic meaning to all its components and relationships among them. Different cloud providers, for example, are explicitly modeled so that per-cloud aggregations of data can be computed. The model, which is stored by the *Manager* component in Fig. 5.1, is maintained in sync in a distributed fashion by data collectors which are running on monitored hosts.

Figure 5.2 shows an example of model of a simple e-commerce webapp deployed on two different clouds (Flexiant and Amazon), providing 3 different methods (register, login and checkout).

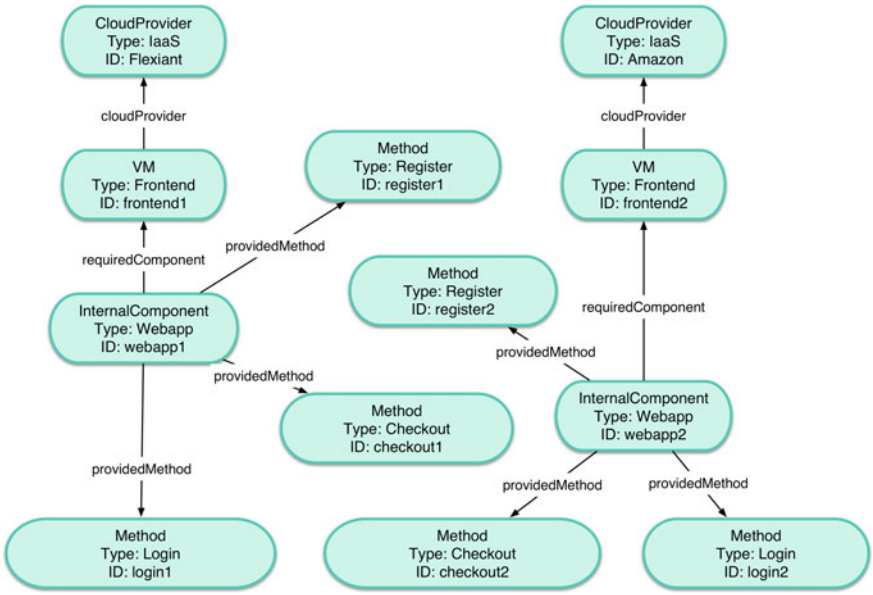

Fig. 5.2 Model example

5.4 Monitoring Rules

The configuration of the monitoring system is obtained via monitoring rules, which consist in recipes written by the QoS engineer describing the monitoring activity in a cloud-independent way. Monitoring rules can be automatically derived from QoS constraints specified during the design time and then customized according to users needs. A rule is composed of 5 building blocks:

- *monitoredTargets*, where a list of monitored resources is identified by either class, type or id;
- *collectedMetric*, where the metric to be collected is specified together with any data collector-specific parameter;
- *metricAggregation*, where the aggregation among average, percentile, sum, count, max, min of collected data is selected as well as whether the aggregation should be over all data or grouped by a specific class of resources (e.g., by cloud provider, or by vm);
- *condition*, where a condition to be verified can be expressed predicating on the aggregated value;
- *actions*, the action to be executed given the condition is satisfied (if any).

Table 5.1 Examples of monitoring rules

RTConstraint_Rule	*DetailedRAMRule*
– id: RTConstraint_Rule timeWindow: 60 timeStep: 60 enabled: true monitoredTargets: – type: Login – type: Register – type: Checkout collectedMetric: metricName: ResponseTime parameters: – samplingProbability: 1 metricAggregation: aggregateFunction: Percentile parameters: – thPercentile: 99 condition: METRIC > 10000 actions: – name: OutputMetric parameters: – metric: RTConstraint_Violation – name: EnableRule parameters: – id: DetailedRAMRule	– id: DetailedRAMRule timeWindow: 10 timeStep: 10 enabled: false monitoredTargets: – type: Frontend collectedMetric: metricName: RAMUtilization parameters: – samplingTime: 5 metricAggregation: aggregateFunction: Average groupingClass: VM actions: – name: OutputMetric parameters: – metric: AverageRAMUtilization

Table 5.1 provides two examples of rules that predicate over the example model provided in Fig. 5.2. The first rule (i.e., *RTConstraint_Rule*) instructs the platform to collect the response time of all three methods, compute the 99th percentile every 60 s, and check if it is lower than 10 s. In case the computed metric is over 10 s, the platform will produce a new metric named *RTConstraint_Violation*, which will be available as input of other rules and for observers, and will enable a second rule named *DetailedRAMRule*. This second rule is telling the platform to collect the average RAM utilization on all Frontend machines and produce a new metric named *AverageRAMUtilization* for each VM. *DetailedRAMRule* is not active in the initial monitoring configuration (in fact, its enabled attribute is set to false). This means that the data it needs are not collected. When the execution of *RTConstraint_Rule* activates it (that is, when the response time of the methods under monitoring is slow), data collectors are instructed to start sending the required metrics to the Data Analyzer that can then execute the rule. Thanks to this mechanism it is possible to increase or decrease the level of the monitoring, and the consequent overhead on the execution of the whole system, depending on the status of the system itself.

5.5 Conclusion

Tower 4Clouds is available as open source software.[1] It has been used as part of MODAClouds by all case studies owners that have been able to customize it for their purpose without a direct intervention of its main developers. Moreover, it is being used also within the SeaClouds project [1] where it has become one of the main infrastructural components. Thanks to its modularity, Tower 4Clouds has been incorporated within SeaClouds as it is, and SeaClouds partners have built around it the code needed to automatically derive monitoring rules and Data Collectors configurations from their design time specification of SeaClouds applications.

References

1. Brogi A et al (2015) CLEI Electron J 18(1):1–14
2. Barbieri DF et al (2010) C-SPARQL: a continuous query language for RDF data streams Int J Semantic Comput 4:3
3. Miglierina M et al (2013) Exploiting stream reasoning to monitor multi-cloud applications. In: 2013 ISWC 2nd international workshop on Ordering and Reasoning (OrdRing), 21–22 Oct 2013

[1] https://github.com/deib-polimi/tower4clouds.