Exploiting Stream Reasoning to Monitor multi-Cloud Applications

Marco Miglierina, Marco Balduini, Narges Shahmandi Hoonejani, Elisabetta Di Nitto, and Danilo Ardagna

Politecnico di Milano, Italy
marco.miglierina@polimi.it, marco.balduini@polimi.it, narges.shahmandi@mail.polimi.it, elisabetta.dinitto@polimi.it, danilo.ardagna@polimi.it

Abstract. This demo shows how we have used a stream reasoning mechanism, C-SPARQL, as the main component of a Monitoring Platform for multi-Clouds applications, that is, applications replicated or distributed on multiple Clouds. The C-SPARQL engine executes *monitoring queries* on data gathered both from application-level components deployed on the Cloud and from Cloud-level resources. We show how, through our Monitoring Platform, we can enable and disable monitoring queries on demand. Thus, we increase/decrease the monitoring granularity depending on the overall system status, and therefore limit the monitoring overhead when possible.

Keywords: multi-cloud applications, cloud computing, stream reasoning, monitoring, c-sparql

1 Introduction

Cloud Computing is a novel paradigm based on the idea that computation, storage and communication resources are offered as a service to any user. Users can exploit the resulting services based on their needs, adopting a pay-per-use cost model. This makes the adoption of Cloud Computing a very interesting business opportunity, especially for those companies who are rapidly growing or expecting to grow up in the future.

Today research is investigating various fronteers of Cloud Computing, one of which is the possibility of designing and managing an application on multiple Clouds. This is an interesting problem as it could limit an issue called *customer lock-in*, that is, the difficulty for a user to change Clouds given the potentially high costs of migration. Moreover, it could open up the possibility of significantly increasing the availability of an application.

In the MODAClouds project[1], among the various issues, we focus on the problem of how to monitor *multi-Cloud applications* in a scenario where applications are replicated on different Clouds and monitoring data of various kinds need to be collected and analyzed from these replicas.

This demo, in particular, shows how we have used a stream reasoning engine, that is, C-SPARQL [1], as the main component to execute *monitoring queries* on data gathered both from application-level components deployed on

[1] www.modaclouds.eu

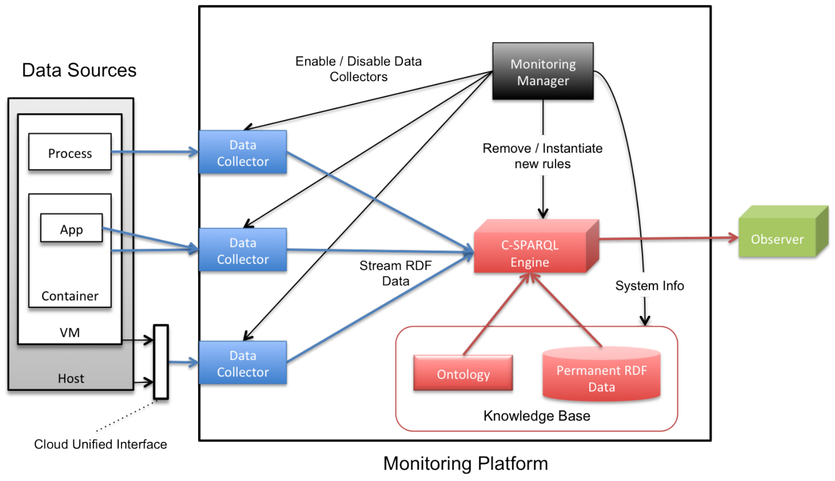

Fig. 1. The MODAClouds Monitoring Architecture.

the Cloud and from Cloud-level resources. We show how, through our monitoring platform, we can enable and disable monitoring queries on demand thus increasing/decreasing the monitoring capabilities of the system depending on its overall status, and therefore limiting the overhead of monitoring when possible and its cost (note that also the network traffic is charged by many Cloud providers).

2 MODAClouds Monitoring Platform

The MODAClouds monitoring architecture is illustrated in Figure 1. *Data Collectors* are in charge of gathering monitoring data from different data sources where the application is running. In particular, they associate semantic meaning to data, and send this information to the C-SPARQL engine in the RDF format. C-SPARQL is an extension of SPARQL language that enables continuous queries over streams of RDF data. *Monitoring queries* are evaluated by the engine according to sliding time windows so that reasoning is performed on knowledge evolving over time. The *Knowledge Base* contains the *Ontology*, which is a formal specification of the common abstractions needed to represent and monitor Cloud systems and applications, and the *Permanent RDF Data*, which contains information about the deployed system. Any *Observer* can finally subscribe for queries so to receive their results when available. Last, the *Monitoring Manager* is responsible for enabling or disabling data collectors, registering or unregistering queries and keeping the Knowledge Base up to date.

3 The Demo

The objective of this demo is to show how the Monitoring Platform behaves when monitoring a commercial Web application deployed on two different Clouds, Amazon Web Services and Eucalyptus@iEAT (a private Cloud based on Eucalyptus 3.0).

In particular, monitoring focuses on the execution time on the server side. The commercial application we use for the demo is Apache Open For Business

(Apache OFBiz), an open source enterprise resource planning (ERP) system. It provides a suite of enterprise applications that integrate and automate many of the business processes of an enterprise.

We implemented Data Collectors by instrumenting the Control Servlet code, which is in charge of handling incoming requests, and deployed it on the two Clouds (located in the Ireland Amazon Region and Romania, respectively). Each replica (which includes an application server and an Apache Derby DB) is installed on a Virtual Machine in each Cloud. We deployed the C-SPARQL engine on another Amazon VM (located in the Ireland Region but in different availability zone). The replicas are stressed through a load injector (we use Apache JMeter) running in Italy and sending requests to both of them.

We registered two different queries on the C-SPARQL engine:

– *Query Q1* checks the average execution time of requests arriving to the whole system, composed by the two OfBiz replicas. If this value, within a 60 sec time window –which is updated with a 10 sec step– is above a 5 sec threshold, the query produces a *Violation Event* on its output stream. In this case we say that the data on the input stream have *violated* the monitoring query.
– *Query Q2* works exactly as query Q1 (and filters the same input) except for the fact that the average is computed for each replica and for each type of request. Thus, the query provides finer grained results and allows the system administrator to identify the Cloud installation that is not performing as expected.

The execution of Q2 requires a larger number of computational and bandwith resources (and hence incurs also in higher cost since the incoming Amazon traffic is charged). Indeed, in case of violations, Q1 sends a violation event every 10 sec, while Q2 sends an event every 10 sec for every type of request in violation. We therefore configured the Monitoring Platform to activate Q2 automatically only in case query Q1 detects a violation.

The Knowledge Base is stored in the VM containing the C-SPARQL engine, and is fed with an ontology that contains both Monitoring Platform-specific concepts and those OfBiz concepts that are relevant for the monitoring activity. The Knowldge Base also contains information about the system deployment.

The Monitoring Manager registers Q1 and Q2, enables the two Data Collectors and attaches a text viewer as Observer of the two queries so the user can visualize the data that are streamed out of them.

In the demo we load the system by means of the load injector so to send requests uniformly to both VMs, increasing the rate linearly (up to 180 requests/minute reached after 10 minutes) so to cause a violation of Q1, the activation of Q2 and then its violation. Q1 violations occur after 2 minutes, while Q2 violations start immediately (after the 60 sec time window) in the two deployments.

4 Discussion

The purpose of this demo is to show an application of stream reasoning on monitoring of Cloud-based applications. While some Cloud providers offer their own monitoring mechanisms, when considering monitoring of multi-Cloud applications new and flexible monitoring platforms are needed. Stream reasoning can be at the core of such kinds of platform thanks to its ability to acquire large quantities of flowing data and reason on them in a flexible way.

Information produced by monitoring platforms vary depending on the QoS constraints posed on the application under analysis. While in this demo we focus on execution time, other kinds of metrics could be more interesting in other cases, for instance the number of requests answered without errors, the average CPU utilization, the number of accesses to data storage, etc. The possibility of programming the behavior of the C-SPARQL engine through queries and the use of ontology certainly offers a way to support the selection, filtering, correlation of such different types of data (that can be also semantically different in different providers). Of course, the engine should also be connected to proper Data Collectors able to provide the right data to reason on.

Disabling and enabling queries depending on the status of the monitored application is an important mean to increase the level of accuracy of monitoring when needed and decreasing it when it would only imply a large overhead and cost on the execution of the application itself. Issues to be considered in the evolution of our platform include the definition of guidelines for the development of Data Collectors. The Data Collector we used in the demo has been hard coded as part of the servlet of the OfBiz application. While this approach may be the only solution in the worst case, less invasive approaches should be pursued in general. A promising approach we are investigating concerns the adoption of aspect-oriented programming [2] as a way to build monitoring code fragments that are then weaved within the application code. This enables separation of concerns still allowing the programmer to instrument the application code. In the case of monitoring resources in a lower level than the application (e.g., the operating system, the virtual machine, the application container, ...), data collection can be achieved only by exploiting specific (and often proprietary) APIs, if any (e.g., Amazon CloudWatch). The interesting characteristics of our Monitoring Platform is that we keep these issues completely decoupled from the logic needed to analyze the data. This certainly simplifies the design, even in presence of many special cases. Another important issue is how to support the designer in the definition of monitoring queries starting from the definition of QoS requirements and constraints typically stated during the design of an application. To support this issue we are experimenting with the semi-automatic generation of monitoring queries by adopting a model-driven approach.

Acknowledgments. This research has been partially supported by the European Commission, Grant no. FP7-ICT-2011-8-318484 MODAClouds project.

References

1. Barbieri, D. F., Braga, D. Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: A continuous query language for RDF data streams. Int. Journal of Semantic Computing (4),1, 3-25. 2010
2. Elrad, T., Filman, R. E., Bader, A: Aspect-oriented programming: Introduction. Commun. ACM (44), 10, 29-32, 2001.