Politecnico di Milano

Facoltà di Ingegneria dell'Informatica

Corso di Laurea in
Ingeneria Informatica
Dipartimento di
Electronica e
Informatzione e Bioingeneria

A Methodology and a Platform for Monitoring Multi-Cloud Applications

Supervisor: Prof. Elisabetta Di Nitto
Assistant supervisor: Prof. Danilo Ardagna

Master's thesis of: Narges Shahmandi Hoonejani matr.764761

Academic Year 2012-2013

Abstract

Owing to the rapid progress in the Information and Communications Technology (ICT) and the subsequent demand for computing power, Cloud Computing, as an efficient method for handling workloads of great diversity and massive scale, has received increasing attentions. Cloud Computing has been increasingly adopted in diverse areas, such as e-commerce, retail industry and academic environment due its various advantages including No up-front investments, lower operating costs, high scalability and elasticity, easy access and reduction of business risks and maintenance expenses.

However, despite the undeniable advantages and considerable spread of Cloud Computing, there are still many related challenges and issues which should be dealt with in order to ensure the efficient and proper operation of this new computing paradigm. Specifically in order to ensure the quality of service and health of the running applications of the Cloud, monitoring the Cloud infrastructures and applications is a crucial task. The solution that has been proposed is defining Data Collectors that are gathered data from resources and produce as the output RDF triples data, and the Data Analyzer that at a high-level could be seen as accepting in input RDF stream data, and producing one or more streams of data in RDF format as the output. In this thesis the DA is implemented by using C-SPARQL (continues-SPARQL) that is a query engine for RDF data.

In the present thesis, in the context of MODAClouds project, a monitoring framework will be developed which provides the enabling mechanism to trigger runtime adaption and to fill the gap between design-time modules and run-time application behavior. Aspect Oriented Programming, owing to its capability of keeping separate the cross-cutting features, has been chosen for developing the monitoring framework. Furthermore, meta-model for specifying the QoS, and meta-model and language for monitoring rules at design time. Finally, a methodology for the automatic/semi-automatic translation of QoS constraints into monitoring rules for translating them into queries is defined.

1 INTRODUCTION

In a world characterized by rapid progress, demand for computing power has been increasing over the last half century. The penetration of Information and Communications Technology (ICT) in our daily social and personal interactions with the world, encompassing business, commerce, education, manufacturing and communication services, has led to the necessity of handling workloads of great diversity and enormous scale in all the crucial fields of today's society.

With the rapid development of processing and storage technologies, and owing to the success of the Internet, computing resources have become cheaper, more powerful and more universally available than ever before. In such a context, dynamic systems are required to provide services and applications which are more competitive, scalable and responsive in comparison to the classical systems. This technological trend has enabled the realization of a new computing paradigm called Cloud Computing, in which resources (e.g., CPU and storage) are provided as general utilities that can be leased and released by users through the Internet in an on-demand approach.

In a Cloud Computing environment, the traditional role of service provider is played by the infrastructure providers who manage Cloud platforms and lease resources according to a usage-based pricing model, and service providers, who rent resources from single or multiple infrastructure providers to serve the end users. The emergence of Cloud Computing has made a tremendous impact on the Information Technology (IT) industry over the past few years, where large companies such as Google [9], Amazon [2] and Microsoft [12] strive to provide more powerful, reliable and cost-efficient Cloud platforms, while business enterprises seek to reshape their business models to gain benefit from this new paradigm. Indeed, Cloud Computing provides several compelling features which takes the attentions of the business owners.

The new mechanism is increasingly adopted in many areas, such as e-commerce, retail industry and academic environment for its various advantages including No up-front investments, lower operating costs, high scalability and elasticity, easy access and reduction of business risks and maintenance expenses.

Cloud Computing uses a pay-as-you-go pricing model. A service provider does not need to invest in the infrastructure to start gaining benefit from Cloud Computing. It simply rents resources from the Cloud according to its own needs and pay for the usage. As a consequence, cloud model is cost-effective because customers pay for their actual usage without up-front costs.

Beside, resources in a Cloud environment can be rapidly allocated and deallocated on demand. Hence, a service provider no longer needs to provision capacities according to the peak load. This ensures huge savings since resources can be released to save on operating costs when service demand is low. In this way, costs are claimed to be reduced and in a public Cloud delivery model capital expenditure is converted to operational expenditure.

Additionally, Infrastructure providers pool large amount of resources from data centers and make them easily accessible. Scalability is possible via dynamic ("on-demand") provisioning of resources on a fine-grained, self-service basis near real-time, without users having to engineer for peak loads (surge computing). Indeed, a service provider can easily expand its service to large scales in order to handle rapid increase in service demands (e.g., flash-crowd effect).

Furthermore, services hosted in the Cloud are generally web-based. Therefore, they are easily accessible through a variety of devices with Internet connections. These devices not only include desktop and laptop computers, but also cell phones and smart devices. Agility improves with users' ability to re-provision technological infrastructure resources.

Moreover, by outsourcing the service infrastructure to the Clouds, a service provider shifts its business risks (such as hardware failures) to infrastructure providers, who often have better expertise than many customers and are better equipped for managing these risks. In this way Cloud Computing guarantees business continuity and disaster recovery. In addition, maintenance of Cloud Computing applications is easier, because they do not need to be installed on each user's computer and can be accessed from different places. Consequently, a service can cut down the hardware maintenance and the staff costs.

However, despite the considerable development and spread of Cloud Computing, it also brings many challenges and new problems in terms of quality of service (QoS),

Service Level Agreements (SLA), security, compatibility, interoperability, costs and performance estimation and so on. These issues have been analyzed and studied in the last few years but still a lot of investigation needs to be carefully addressed.
Accordingly, in order to ensure the quality of service and health of the running applications of the Cloud, monitoring the Cloud infrastructures and applications is a crucial task. Cloud monitoring also allows us to get insights into the system and to gather information and scalability and consequently coming up with adaptation decisions based on the monitoring data.
In general, the monitoring approaches differ from various points of view including monitoring actor, monitored object, timing of the monitoring procedure, monitoring mechanism, monitoring environment architecture and monitoring constraints and output.
One of the important monitoring challenges is to tackle the different monitoring constraints imposed by different target clouds. IaaS and PaaS platforms offer very different metrics and therefore a multi-cloud monitoring platform has to cope with this heterogeneity.

MODAClouds, Model-Driven Approach for design and execution of applications on multi clouds, is a European project which is focused on providing method, a decision support system, an open source IDE and run-time environment for the high-level design, early prototyping, semi-automatic code generation, and automatic deployment of applications on multi-Clouds with guaranteed QoS. MODAClouds includes a *Runtime Environment* that will implement the MAPE-K loop (monitor, analysis, planning, execution, knowledge) reference blueprint for the implementation of self-adaptive applications. The monitoring platform that is provided by this project affords a possible approach in order to face the heterogeneity data that are received from different layers of Cloud, that is explicitly distinguish between *infrastructure-level metrics*, that are exposed only on IaaS clouds, from *application-level metrics*, that will rely on monitoring probes injected in the application code and therefore can be collected on any target cloud.

The approach which has been proposed for developing the monitoring platform of MODAClouds project is defining Data Collectors which gather data from resources and produce the RDF triples data as the output, and the Data Analyzer that at a high-level could be seen as accepting RDF stream data as an input and producing one or more streams of data in RDF format as the output.

In the present thesis, in the context of MODAClouds project, a monitoring framework is developed which provides the enabling mechanism to trigger runtime adaption and to fill the gap between design-time modules and run-time application behavior. The corresponding Data Analyzer is implemented by using C-SPARQL (continues-SPARQL) that is a query engine for RDF data.

Different approaches can be employed for developing the cloud monitoring systems. Aspect oriented programming (AOP) is one of the proper choices that can be utilized for accomplishing this task especially for PaaS layer. The capability of keeping separate the cross-cutting features like response time, and status code of a web page, is the advantage which makes AOP a promising option for monitoring purposes . Accordingly AOP has been employed in the present work for developing the monitoring platform.

For IaaS layer resources, we considered the data of two cloud providers in specific, Azure and Amazon EC2. In Amazon EC2 the data are directly gathered from the CloudWatch, while in Azure as we could not gather them directly they have been retrieved from storage.

In addition, meta-model for specifying the QoS, and meta-model and language for monitoring rules at design time, are defined. Moreover, a methodology for the automatic/semi-automatic translation of QoS constraints into monitoring rules for translating them into queries is proposed.

Original Contributions

The main objectives of this work are as follows:

- Designing Data Collectors in order to gathering the monitoring data produced by the various data sources, and producing RDF triple as the output for the Data Analyzer in order to get inside the system, analyze the received data and apply further modifications
- Defining a meta-model that specifies the constraints of the Quality of the Service at design time,
- Defining a meta-model and a language for the specification of the monitoring rules at design time, and mapping the QoS constraints into the related monitoring rule. Also new monitoring rules were provided by extending or composing the other monitoring rules,
- Identifying a methodology for the automatic/semi-automatic translation of QoS constraints into monitoring rules for translating monitoring rules into probes and monitoring queries that will be executed by the analysis component

Outline of the Thesis

This thesis is organized as follows:

- Chapter 2 discusses the main concept of the cloud computing and cloud monitoring and explains the state of the art concepts and techniques relative to our work
- In Chapter 3, first we introduce the MODAClouds project and then the main technologies that are used in this thesis are reviewed. In addition, there the case study of this thesis, MiC application, and its Palladio meta-models are provided.
- In Chapter 4 the monitoring architecture of MODAClouds, and related core ontology is reviewed and is described. Then knowledge base information generated during the design time and run time is explained. Finally dedicated to the mapping of the monitoring rules to C-SPARQL queries.

  •Chapter 5 the monitoring approach that is developed in this thesis, is evaluated. The main objectives are quantitatively evaluation of the overhead that is introduce by the monitoring platform which are done on MiC application.

2 CHAPTER 2

This chapter presents a general overview on Cloud Computing and Cloud Monitoring and explains the state of the art concepts and techniques relative to our work.

After a short introduction on basic concepts in Section 2.1, we provide and analyze a definition of Cloud, illustrate the main characteristics and show different structural models.

In Section 2.2, we provide a general overview on Cloud Monitoring then we investigate the current monitoring platforms acting at different layers of the cloud stack. A classification of the state of the art literature which considers general monitoring, infrastructure-level monitoring and application level monitoring is also proposed.

2.1 CLOUD COMPUTING BASIC CONCEPTS

In a world characterized by progress, fast changes and advances, demand for computing power has been increasing over the last half century. Handling workloads of great diversity and enormous scale is necessary in all the most significant fields of today's society, due to the penetration of Information and

Communications Technology (ICT) in our daily interactions with the world both at personal and community levels, encompassing business, commerce, education, manufacturing and communication services. With the rapid development of processing and storage technologies, and with the success of the Internet, computing resources have become cheaper, more powerful and more universally available than ever before. In such a setting, dynamic systems are required to provide services and applications that are more competitive, more scalable and more responsive with respect to the classical systems. This technological trend has enabled the realization of a new computing paradigm called Cloud Computing, in which resources (e.g., CPU and storage) are provided as general utilities that can be leased and released by users through the Internet in an on-demand fashion.

In a Cloud Computing environment, the traditional role of service provider is divided into two: the infrastructure providers who manage Cloud platforms and lease resources according to a usage-based pricing model, and service providers, who rent resources from one or many infrastructure providers to serve the end users. The emergence of Cloud Computing has made a tremendous impact on the Information Technology (IT) industry over the past few years, where large companies such as Google [9], Amazon [2] and Microsoft [12] strive to provide more powerful, reliable and cost-efficient Cloud platforms, and business enterprises seek to reshape their business models to gain benefit from this new paradigm. Indeed, Cloud Computing provides several compelling features that make it attractive to business owners.

The new mechanism is increasingly adopted in many areas, such as e-commerce, retail industry and academy for its various advantages:

• *No up-front investments*: Cloud Computing uses a pay-as-you-go pricing model. A service provider does not need to invest in the infrastructure to start gaining benefit from Cloud Computing. It simply rents resources from the Cloud according to its own needs and pay for the usage. As a consequence, cloud model is cost-effective because customers pay for their actual usage without up-front costs.

• *Lowering operating costs*: Resources in a Cloud environment can be rapidly allocated and deallocated on demand. Hence, a service provider no longer needs to provision capacities according to the peak load. This ensures huge savings since resources can be released to save on operating costs when service demand is low. In this way, costs are claimed to be reduced and in a public Cloud delivery model capital expenditure is converted to operational expenditure.

• *High scalability and elasticity*: Infrastructure providers pool large amount of resources from data centers and make them easily accessible. Scalability is possible via dynamic ("on-demand") provisioning of resources on a fine-grained, self-service basis near real-time, without users having to engineer for peak loads (surge computing). Indeed, a service provider can easily expand its service to large scales in order to handle rapid increase in service demands (e.g., flash-crowd effect).

• *Easy access*: Services hosted in the Cloud are generally web-based. Therefore, they are easily accessible through a variety of devices with Internet connections. These devices not only include desktop and laptop computers, but also cell phones and smart devices. Agility improves with users' ability to re-provision technological infrastructure resources.

• *Reducing business risks and maintenance expenses*: By outsourcing the service infrastructure to the Clouds, a service provider shifts its business risks (such as hardware failures) to infrastructure providers, who often have better expertise than many customers and are better equipped for managing these risks. In this way Cloud Computing guarantees business continuity and disaster recovery. In addition, maintenance of Cloud Computing applications is easier, because they do not need to be installed on each user's computer and can be accessed from different places. Consequently, a service can cut down the hardware maintenance and the staff costs.

However, despite the considerable development and spread of Cloud Computing, it also brings many challenges and new problems in terms of quality of service (QoS), Service Level Agreements (SLA), security, compatibility, interoperability, costs and performance estimation and so on. These issues have been analyzed and studied in the last few years but still a lot of investigation needs to be carefully addressed.

Before presenting the state of the art and discussing the main research challenges, in the next sections we explain what Cloud Computing is, highlighting its key concepts and architectural principles.

2.1.1 Cloud Computing Definition

The origin of the term "Cloud Computing" is obscure as it has never been defined in a unique way and precise circumstance. It appears to derive from the practice of drawing stylized Clouds to denote networks in diagrams of computing and communications systems since the half of the XX century.

The word "Cloud" is used as a metaphor for the Internet, based on the standardized use of a Cloud-like shape to denote a network on telephony schematics and later to depict the Internet in computer network diagrams as an abstraction of the underlying infrastructure it represents.

The main idea behind Cloud Computing is not a new one, unlike other technical terms; it is not a new technology, but rather a new operations model that brings together a set of existing technologies to run business in a different way. Indeed, most of the elements used by Cloud Computing, such as virtualization and utility-based pricing, are not new. Instead, Cloud Computing leverages these existing technologies to meet the technological and economic requirements of today's demand for information technology.

If we consider that with the first available large-scale mainframe in academia and corporations, accessible via thin clients / terminal computers, it became important to find ways to get the greatest return on the investment in them, allowing multiple users to share both the physical access to the computer from multiple terminals as well as to share the CPU time, and eliminating periods of inactivity (time-sharing), we can affirm that the underlying concept of Cloud Computing dates back to the 1950s.

In 1961, John McCarthy was the first to suggest publicly that computer time-sharing technology might result in a future in which computing power and even specific applications could be provided and sold through the public utility business model (like water or electricity). This idea was very popular during the late 1960s, but faded by the mid-1970s, since hardware and telecommunications were not sophisticated and prepared enough for this progressive scheme.

The term "Cloud" has also been used in various contexts such as describing large ATM (Asynchronous Transfer Mode) networks in the 1990s. Telecommunications companies began to offer VPN (Virtual Private Network) services instead of dedicated point-to-point data circuits, with comparable quality of service but at a much lower cost. The Cloud symbol was used to represent the demarcation line between provider's and user's responsibility. This boundary was soon extended to cover servers as well as the network infrastructure.

However, since 2000, the idea has resurfaced in new forms. It was after Google's CEO Eric Schmidt used the word to describe the business model of providing services across the Internet in 2006, that the expression really started to gain popularity. Since then, the term Cloud Computing has been used mainly as a marketing term in a variety of contexts to represent many different ideas. The ubiquitous availability of high-capacity networks, low-cost computers and storage devices as well as the widespread adoption of hardware virtualization, service-oriented architecture, autonomic, and utility computing had led to a tremendous growth in Cloud Computing in various fields of application. This is the reason why Cloud Computing term does not have a standard definition. The lack of general and uniform concept generated not only market hypes, but also a fair amount of skepticism and confusion. For this reason, recently there has been work on standardizing the definition of Cloud Computing. As an example, in [87] the author compared over 20 different definitions from a variety of sources to confirm the following standard definition:

Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs.

2.1.2 Characteristics

The above definition highlights the basic properties of Cloud Computing:

- *Ubiquity*: The user can totally ignore the location of the hardware infrastructure hosting the required service and make use of the service everywhere and every time he needs through his client application.
- *Convenience*: The consumer can use a service exploiting remote physical resources, without necessarily buying/acquiring them. As he is just charged for the resources provided according to a pay-per-use mechanism, utility-based pricing lowers service operating cost.
- *On-demand activation*: A service consumes resources only when it is explicitly activated by the user, otherwise it is considered inactive and the resources needed for its execution can be used for other purposes.

Moreover, the NIST definition also specifies five essential characteristics of Cloud Computing, as stated in [38]:

- *On-demand self-service*: Resources can be allocated or "allocated on demand", without requiring human interaction with the service's providers. The automated self-organized resource management feature yields high agility that enables service providers to respond quickly to rapid changes.

- *Broad network access*: Capabilities are available over the Internet and are accessed through mechanisms that promote use by simple (thin) or complex (thick) client platforms (e.g., any device with Internet connectivity such as mobile phones, laptops, and smart devices). Moreover, to achieve high network performance and localization, many of today's Clouds consist of data center distributed in many locations around the globe. A service provider can easily leverage geo-diversity to achieve maximum service utility.
- *Resource pooling*: Different physical and virtual resources are dynamically assigned and reassigned according to consumer demand and needs; they are pooled by providers to serve multiple resource users and using a multi-tenant model. The customer's control is independent from the exact location of the provided resources but may know a location at a higher level of abstraction (e.g., country, state, or data-center). Such dynamic resource assignment capability provides much flexibility to infrastructure providers for managing their own resource usage and operating costs. Examples of resources include storage, CPUs, memory, network bandwidth, and virtual machines.
- *Rapid elasticity*: Resources can be rapidly and elastically provisioned, in some cases automatically, to quickly scale out and rapidly released to quickly scale in; the consumer often perceives unlimited availability of resources that can be purchased in any quantity at any time. Compared to the traditional model that provisions resources according to peak demand, dynamic resource provisioning allows service providers to acquire resources based on the current demand, which can considerably lower the operating cost.
- *Measured service*: Resources usage is always automatically controlled and optimized by leveraging a metering capability at a level of abstraction appropriate to the type of service (e.g., storage, bandwidth, CPU activity time for processing services and so on). This monitoring mechanism provides transparency for both the provider and consumer of the agility that enables service providers to respond quickly to rapid changes in service demand such as the flash crowd effect.

2.1.3 Structure models

This section aims to give a global description of the architectural, business and various operation models of Cloud Computing.

A layered architecture

In a Cloud environment, services owned by multiple providers are collocated in a single data center. The performance and management issues of these services are shared among service providers and the infrastructure provider. The layered architecture of Cloud Computing provides a natural division of responsibilities: the owner of each layer only needs to focus on the specific objectives associated with that layer. However, multi-tenancy also introduces difficulties in understanding and managing the interactions among various stakeholders.

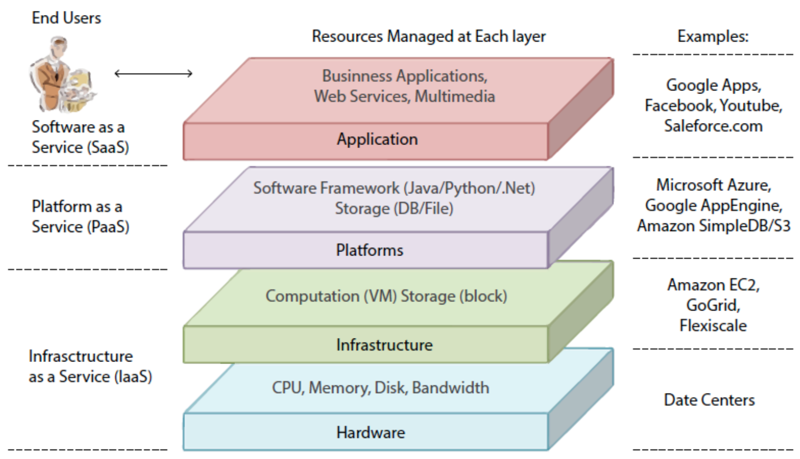

End Users
Resources Managed at Each layer
Examples:
Businness Applications, Web Services, Multimedia
Application
Software as a Service (SaaS)
Google Apps, Facebook, Youtube, Saleforce.com
Software Framework (Java/Python/.Net) Storage (DB/File)
Platforms
Platform as a Service (PaaS)
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3
Computation (VM) Storage (block)
Infrastructure
Amazon EC2, GoGrid, Flexiscale
Infrasctructure as a Service (IaaS)
CPU, Memory, Disk, Bandwidth
Hardware
Date Centers

Figura 2-1

Figure 1: Cloud Computing architecture [103]

Generally speaking, we can identify four layers that compose the architecture of a Cloud Computing environment: The hardware/data center layer, the infrastructure layer, the platform layer and the application layer, as shown in Figure 2.1 This classification allows to understand what each of the layers is composed of, what the intended function of that layer is, and how these layers interact with each other. By simplifying the Cloud Computing concept into layers, it is easier to define the roles within the overall structure and explain where the business fits into the model. A detailed description of each layer follows:

1. *The hardware layer:* This layer is responsible for managing the physical resources of the Cloud, including physical servers, routers, switches, power suppliers, and cooling systems. In practice, the hardware layer is typically implemented in data centers. A data center usually contains thousands of servers that are organized in racks and interconnected through switches, routers or other fabrics. Typical issues at hardware layer include hardware configuration, fault-tolerance, traffic management, power and cooling resource management. Note that the physical hardware is being sliced into virtual machines (VMs), each having their own small (usually Linux or UNIX based) operating system installed.
2. *The infrastructure layer*: Also known as the virtualization layer, the infrastructure layer creates pools of storage and computing resources by partitioning the physical resources using virtualization technologies such as Xen [22], KVM [11] and VMware [21]. These pools of resources are the key to providing elasticity, scalability and flexibility with respect to server architecture. Indeed, virtual machines can be brought online and assigned to a resource pool on-the-fly when the demand on that pool increases, while they can then be destroyed when no longer needed. The ability to provision and delete virtual machines on the y allows a vendor to provide Infrastructure as a Service (IaaS). As a consequence, instead of purchasing servers or even hosted services, IaaS customers can create and remove virtual machines and network them together at will. Thanks to virtualization technologies, the infrastructure layer offers VMs as a service to end users that have complete control of their environments.
3. *The platform layer*: Built on top of the infrastructure layer, the platform layer consists of operating systems and application frameworks, and abstracts the IaaS layer by removing the individual management of virtual machine instances. The purpose of this layer is to minimize the burden of deploying applications directly into VM containers. In fact, at this layer customers do not manage their virtual machines; they merely create their own programs and applications, which are hosted by the platform services they are paying for, within an existing API or programming language. This frees the developers from concerns about environment configuration and infrastructure scaling, but offers limited control.
4. *The application layer*: At the highest level of the hierarchy, the application layer consists of the actual Cloud applications that offer web-based software as a service

(SaaS), such as email or CRM (Customer Relationship Management). In this layer, users are truly restricted to only what the application is and can do; they get only pre-defined functionality and they cannot go much further than that. Indeed, applications are designed for ease of use and GTD (getting things done). Billing can be based on utility or at monthly fee. Either way, it is a simple way to get the application functionality you need without incurring the cost of developing that application. Different from traditional applications, Cloud applications can leverage the automatic-scaling feature to achieve better performance, availability and lower operating cost. We note that the architecture of Cloud Computing is modular, much more than traditional service hosting environments. Each layer is loosely coupled with the layer above and below, allowing each layer to evolve separately. The architectural modularity allows Cloud Computing to support a wide range of application requirements while reducing management and maintenance overhead.

Cloud service models

Cloud Computing adopts a service-driven operating business model, indicating a strong emphasis on service management. In other words, hardware and platform-level resources are provided as services on an on-demand-basis, according to the SLAs negotiated with its customers. Conceptually, every layer of the architecture described in the previous section can be implemented as a service to the layer above. Conversely, every layer can be perceived as a customer of the layer below. However, in practice, Cloud Computing providers offer services that can be grouped into three fundamental categories: infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS), as in Figure 2.

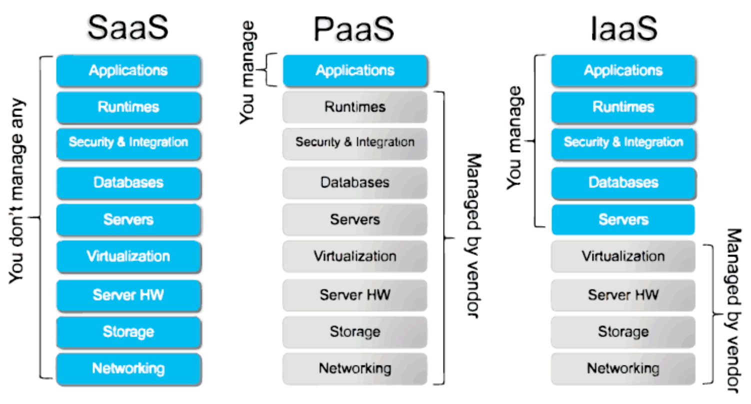


Figure 2: Cloud service models.

- *Infrastructure as a Service*: IaaS refers to on-demand provisioning computers, storage and other infrastructural physical resources, usually in terms of VMs. The Cloud owner who offers IaaS is called a IaaS provider. The consumer is able to deploy and run arbitrary software, which can include operating systems and applications. In this model, the consumer does not manage or control the underlying Cloud infrastructure but he is responsible for patching and maintaining the operating systems, deployed application software, storage, and he possibly has limited control of select networking components (e.g., host _firewalls). Cloud providers typically bill IaaS services on a utility computing basis, that is, cost reflects the amount of resources allocated and consumed. Examples of IaaS providers include Amazon EC2 [1An], Windows Azure Virtual Machines [15An], Google Compute Engine [8An], GoGrid [5An], and Flexiscale [4An].
- *Platform as a Service*: PaaS refers to providing platform layer resources, typically including operating system support, database, web server and software development frameworks. These provided capabilities are consumer-created or acquired applications, created using programming languages and tools supported by the provider. The consumer has control over the deployed applications and possibly application hosting environment configurations but cannot manage the underlying Cloud infrastructure. Application developers can develop and run their software solutions on a Cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers. With some PaaS offers, the underlying computer and storage resources scale automatically to match application demand such that Cloud user does not have to allocate resources manually. Examples of PaaS providers include Google App Engine [6An], Microsoft Windows Azure Compute [14An], and Force.com [19An].
- *Software as a Service*: SaaS refers to providing on-demand applications over the Internet. Cloud providers install and operate application software running on a Cloud infrastructure while Cloud users access this software from various client devices through a thin interface such as a web browser (e.g., web-based email). The consumer does not manage or control the underlying Cloud infrastructure and platform on which the application is running, with the possible exception of limited user specific application configuration settings. This eliminates the need to install and run the application on the Cloud user's own computers simplifying maintenance and support. The pricing model for SaaS applications is typically a monthly or yearly at fee per user, so price is scalable and adjustable if users are added or removed at any point. Examples of SaaS providers include Salesforce.com [19An], Rackspace [18An] and SAP Business ByDesign [20An], Google Apps [7An], Microsoft Office 365 [13An], and Onlive [17An].

We observe that PaaS and IaaS providers are often called the infrastructure providers of Cloud providers since, in the current practice, they are often part of the same organization (e.g. Google and Salesforce). However, according to the layered architecture of Cloud Computing, it is entirely possible that a PaaS provider runs its Cloud on top of an IaaS provider's Cloud.

IaaS instance options

In addition to providing the flexibility to easily choose the number, the size and the configuration of the compute instances the customers need for their applications, a IaaS provides customers different purchasing models that give them the flexibility to optimize their costs. For example Amazon EC2, IaaS market leader and reference model of this work, offers three kinds of instances: i) *On-demand* instances which allow the customer to pay a fixed hourly rate with no commitment; ii) *reserved* instances where the customer pay a low, one-time fee and in turn receive a significant discount on the hourly charge; iii) *on-spot* instances which enable the customer to bid whatever price he wants for individual instance, providing for even greater savings if his application have flexible start and end times. An accurate description of the three different EC2 instance options follows:

- *On-demand instances*: On-demand instances require no long-term commitments or upfront payments. Customers can increase or decrease compute capacity depending on the demands of their own applications and only pay the needed rate for the instances they use. A IaaS always strives to have enough on-demand capacity available to meet customers' needs, but during periods of very high demand, it is possible that it might not be able to launch specific on-demand instance types in specific availability zones (i.e., a specific Amazon data center) for short periods of time. On-demand instances are recommended for users that want the low cost and flexibility without any up-front payment or long-term commitment, or applications with short term, spiky, or unpredictable workloads that cannot be interrupted.
- *Reserved instances*: Functionally, reserved and on-demand instances perform identically but those reserved let the customer make a low, one-time, upfront payment for an instance, reserve it for a one or three year term, and pay a significantly lower hourly rate for that instance. Customers are assured that their own reserved instance will always be available for the operating system (e.g. Linux/UNIX or Windows) and availability zone in which they purchased it. For applications that have steady state needs, reserved instances can provide high savings compared to using on-demand instances. Reserved instances are usually recommended for applications with predictable usage, applications that require

reserved capacity, including disaster recovery and users able to make upfront payments to reduce their total computing costs even further.

- *On-spot instances*: Spot instances provide the ability for customers to purchase compute capacity with no upfront commitment and at hourly rates usually lower than the on-demand rate. Spot instances allow you to specify the maximum hourly price that you are willing to pay to run a particular instance type. IaaS sets a spot price for each instance type in each availability zone, which is the price all customers will pay to run a spot instance for that given period. The spot price fluctuates based on supply and demand for instances, but customers will never pay more than the maximum price they have specified. If the spot price moves higher than a customer's maximum price, the customer's instance will be shut down by the IaaS. Other than those differences, spot instances perform exactly the same as on-demand or reserved instances. For the majority of cases, spot instances are recommended for applications that have flexible start and end times, applications that are only feasible at very low compute prices and users with urgent computing needs for large amounts of additional capacity. Due to the nature of on-spot instances, competitions for their acquisition raise between customers.

Cloud deployment models

There are many issues to consider when moving an enterprise application to the Cloud environment. For example, some service providers are mostly interested in lowering operation cost, while others may prefer high reliability and security. Accordingly, there are different types of Clouds, each with its own benefits and drawbacks:

- *Public Clouds*: A Cloud in which service providers offer resources as services available to the general public or a large industry group and owned by a private organization selling Cloud services (like Amazon AWS, Microsoft and Google); these services are free or offered on a pay-per-use model. Public Clouds offer several key benefits to service providers, including no initial capital investment on infrastructure and shifting of risk to infrastructure providers. However, they lack fine-grained control over data, network and security settings, which restricts their effectiveness in many business scenarios.
- *Private Clouds*: Also known as internal Clouds, private Clouds are designed for exclusive use by a single organization. A private Cloud may be hosted internally or externally and managed by the organization or by a third-party represented by external providers. A private Cloud offers the highest degree of control over performance, reliability and security. However, they are often criticized for being similar to traditional proprietary server farms and do not provide benefits such as no up-front capital costs. Moreover, undertaking a private Cloud project requires a significant level and degree of engagement

to virtualize the business environment: every one of the steps in the project raises security issues that must be addressed in order to avoid serious vulnerabilities.

- *Community Clouds*: The Cloud infrastructure is shared by several organizations and supports a specific community that has common concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be managed by the organizations or by a third party and may exist on premise or o_ premise. The costs are spread over fewer users than a public Cloud (but more than a private Cloud), so only some of the cost savings potential of Cloud Computing are realized.
- *Hybrid Clouds*: A hybrid Cloud is an alternative solution to addressing the limitations of both public and private Clouds. It is a combination of two or more Cloud models (public, private or community), that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., Cloud bursting for load-balancing between Clouds). In this way companies and individuals are able to obtain degrees of fault tolerance combined with locally immediate usability without dependency on internet connectivity. Hence, hybrid Cloud architecture is flexible and scalable. Compared to public Clouds, they provide tighter control and security over application data, while still facilitating on-demand service expansion and contraction. On the down side, designing a hybrid Cloud requires carefully determining the best split between public and private Cloud components. For most service providers, selecting the right Cloud model depends on the business scenario. For example, computing-intensive scientific applications are best deployed on public Clouds for cost-effectiveness. Arguably, certain types of Clouds will be more popular than others. In particular, it was predicted that hybrid Clouds will be the dominating deployment model for most organizations.

It is fundamental to note that Cloud Computing is still an evolving paradigm. Its definitions, structure, use cases, underlying technologies, issues, risks, and benefits will be refined in a spirited debate by the public and private sectors. The definitions, attributes, and characteristics given in the previous discussion will evolve and change over time. Finally we remark that the Cloud Computing industry represents a large ecosystem of many models, vendors, and market niches. Our description attempts to encompass all of the various Cloud approaches.

2.2 CLOUD MONITORING

In the context of Cloud management, monitoring Cloud infrastructures and applications is a critical task. It allows us to i) get insights into the system and the health of the running applications, ii) to gather information and scalability and, as a result, coming up with adaptation decisions based on monitoring data. In general, monitoring approaches differ from various points of view, in particular:

- The monitoring actor (**who)** can be the application/service provider, cloud provider, or third party. Who performs monitoring has an impact on the aspects possible to monitor. For instance, an application/service provider has usually a full control over the execution of its application components/services and can easily probe their internals. In contrast, it could not easily access to infrastructure level information that the cloud provider may hide.
- **What objects and properties are monitored**. The monitored objects can be an application, cloud resources or some specific services such as queues. As for the monitored properties different approaches can be distinguished one from the other for the following two aspects:
    - **Types of monitored properties.** Monitored properties can be functional (proper functionality) or non-functional (quality aspects) such as execution cost, response time, and throughput.
    - **Punctual versus History-based monitoring**. The monitoring is punctual if it concentrates on values collected at particular instants of the execution. The term history-based monitoring refers to the case when the analysis considers the history of the system in a certain time window in order to discover the presence (absence) of sequences of values or events.
- **When monitoring happens**. This concerns the timing of the monitoring process with respect to the execution of the monitored system.
- **How the monitoring system is built**. This refers to the monitoring mechanism, the expressiveness of the language and abstraction level, the capability of diagnosing and deviation handling and the runtime support.

- **The language expressiveness**. The type of monitored properties and the capability of predicating on single values and histories lead to the expressiveness of the monitoring language. After deciding what we want to monitor, we need a way to render monitoring directives. Usually services are monitored through special purpose constraints that must be checked during execution such as compliance with promised SLAs. History-based constraints require a temporal logic to relate the values belonging to a sequence, while monitoring the QoS properties imposes a language allowing for suitable representation.
- **Abstraction level**. Monitoring properties can be expressed at various abstraction levels. There is a distinction between the level at which monitoring works and the level at which the user is required to define such properties. This distinction helps characterize what the user specifies with respect to what the execution environment must cope with. Abstraction level refers to the first aspect and does not consider what the runtime support executes. This aspect is taken into account when considering the degree of automation intrinsic within each approach.
- **Architecture of the monitoring environment**. Monitoring constraints must be specified and then evaluated while the system executes. The support could be in the form of modeling environment, meaning what the approach offers to the user to specify the monitoring constraints (e.g., Palladio component modeling) and execution environment meaning what the approach offers/requires to check directives at runtime. Usually, specification environments propose proprietary solutions, while execution environments can be based on standard technology, proprietary solutions, or suitable customizations of existing engines. The execution environment may also include the mechanisms deployed for generating the monitoring information such as instrumentation, reflection, and the interception of events/information exchanged in the execution environment.
- **Filtering**. Pure monitoring is in charge of detecting possible discrepancies between what stated by monitoring constraints and what is actually collected from the execution. Filtering and reasoning abilities enable analysis of complex properties by combining raw data collected by the monitoring infrastructure.
- **Monitoring output**. The monitoring environment can offer its output through specific and proprietary GUI or it can offer APIs enabling the retrieval of monitoring data. In general, the richness of information offered as an outcome of the monitoring activity is of paramount importance.

- **Derivation of monitoring constraints**. Monitoring directives can be either programmed explicitly or be derived (semi) automatically from other artifacts, e.g., design specification containing QoS information.
- **Reactive/proactive monitoring**. Reactive monitoring takes actions to solve problems in response to one or more incidents, after a problem has occurred. It is designed to analyze the direct and root causes of the problems and then take corrective actions to fix them. Optionally, it could collect data for comparison with past and future events and allow related risk assessment. For example, reactive monitoring comprises reduction, correlation, sequencing, notification and reporting of event, automated actions and responses, and the implementation of special-purpose policies to constrain problems. Proactive monitoring implies the definition of monitoring actions trying to identify and solve problems before occurrences such as the verification of SLAs, capacity planning and treatment of statistics to measure of the system behaves.

2.2.1 General Monitoring Approaches

In this section some approaches generally applicable to web service monitoring are presented, together with some cloud-specific ones that are dealing with all layers of the Cloud.

COMPAS [Mos02**]**. The COMponent Performance Assurance Solutions (COMPAS) is a performance monitoring approach for J2EE systems in which components are EJBs. The framework is divided into three parts: monitoring, modelling, and prediction. Java Management Extensions (JMX) is used in the monitoring part. An EJB application is augmented with one proxy for each component EJB. Such proxies mirror in the monitoring system each component in the original application. Timestamps for EJB life-cycle events are sent through these proxies to a central dispatcher. During runtime, a feedback loop connecting the monitoring and modelling modules, allows the monitoring to be refined by the modelling process in order to increase the efficiency of monitoring/modelling.

TestEJB [Mey04**]**. This approach deals with QoS specification of components by introducing a novel measurement architecture. It is is the performance monitoring of J2EE systems implementing an application-independent profiling technique. The framework focuses on response time by monitoring execution times of EJBs and also traces of users calls. The approach relies on the Java Virtual Machine Profiler Interface (JVMPI) and records events for constructing and deconstructing EJBs. This method allows tracing back memory consumption to individual components and introduces a significant overhead in the range of seconds and should only be used in a testing environment.

PAD [Par08]. Monitoring is not limited to just obtaining some raw data, rather it concerns also analyzing the data in order to detect functionality and performance issues that then trigger ameliorative adaptation decisions. The objective of Performance Anti-pattern Detection (PAD), which is based on the COMPAS framework, is the automatic detection of performance anti-patterns in EJB component-based systems. The framework includes three modules as performance monitoring, reconstruction of a design model and anti-pattern detection. PAD is portable across different midleware implementations as a result of using standard J2EE mechanisms. For each EJB module proxies are used regarding collect timestamps and call sequences. Based on rules implemented with the JESS rule engine, anti pattern detection on the reconstructed design model could be achieved. PAD collcts data at the correct level of abstraction and provide sufficient runtime context for the collected data.

An elastic Multi-layer monitoring approach [Kon12]. A peer-to peer scalable distributed system for monitoring is proposed in [Kon12], enabling deployment of long-living monitoring queries (query framework) across the cloud stack to collect metrics and trigger policies to automate management (policy framework). The monitoring architecture of the approach is composed of three layers as data, processing and distribution, interfacing on different levels with the cloud stack.
The data layer provides extensible adaptors to cope with resource heterogeneity. The processing layer describes complex queries (in a SQL-like syntax) over data and also defines policy rules to be triggered if needed. The distribution layer performs an automated deployment of processing operators in the correct places relying on services such as SmartFrog [Gol09], SLIM [Kir10] or Puppet [Tur07].

RMCM [Sha10]. Runtime Model Based Monitoring Approach for Cloud (RMCM) [Sha10] represents a running cloud through an intuitive model.The objective is to provide an opearable profile of a running cloud and to apply it for implementing a flexible monitoring framework. RMCM is presented from the point of view of three roles in the cloud: cloud operators, service developers, and end users. The entities of this model are: *Interaction behavior, Application, Middleware* and *Infrustructure.* The main focus of infrastructure monitoring is on the resource utilization. Applications are monitored from the design and performance point of views. The monitoring is based on server-agent architecture. A monitoring agent is deployed on each virtual machine in charge of collecting runtime information of all the entities on the same VM. These entities are equipped with various monitoring mechanisms to collect runtime information from entities of each level. Collected information is used to instantiate corresponding RMCM to be checked based on some pre-defined rules. Administrators can view and query this monitoring information from the DB. They also can modify the monitoring configuration for each agent.

mlCCL [Bar12**]**. The Multi-layer Collection and Constraint Language (mlCCL) [Bar12] is an event-based multi-level service monitoring approach which defines runtime data to be collected and how to collect, aggregate and analyze it in a multi-layered system such as Cloud. mlCCL Data is described as Service Data Object (SDOs) that may have two kinds of data collections: *messages* and *indicators*. *Messages* are used to obtain the request or response messages exchanged during service invocations. In the case of a new service invocation for which a message collection is defined, the mlCCL tool produces a new SDO and outputs it to an event bus so that the designer can make further use of it. In addition to collected messages and the location of them, an SDO contains a timestamp indicating *when* the message was sent or received by the service runtime (NTP - Network Time Protocol) timestamps are used for the purpose of clock synchronization when sampling sources on different computers with a precision in the order of 10 ms and an instanceID, that is a unique ID identifying the specific service call. *Indicators* are not triggered by any particular service call. They collect periodic information about a service. An indicator can be a Key Performance Indicator (KPI) such as average response time or throughput, or a Resource Indicator (RI) such as the amount of available memory or idle CPU in a virtual machine. Upon calculating a new indicator value by mlCCL runtime, it is wrapped in an SDO and output to an event bus for further use.

2.2.2 Infrastructure-Level Monitoring

Infrastructure-level monitoring involves collecting metrics related to CPU, memory, disk and network from a IaaS platform, either via monitoring probes deployed inside VMs or using services provided by the cloud platform itself (e.g., Amazon Cloudwatch). In this section we review current standard monitoring tools and cloud-specific tools that may be appropriate for this purpose.

Ganglia []. Ganglia is a scalable distributed monitoring system for high-performance computing system such as clusters and Grids. Ganglia is able to run on Linux and Windows.

Nagios []. Nagios offers the ability to monitor applications, services, operating systems, network protocols, system metrics and infrastructure components with a single tool. Nagios is able to respond to issues at the first sign of a problem and automatically fix problems when they are detected. Nagios is able to run on Linux and Windows.

MonALISA []. It is an assembler of autonomous multi-threaded, self-describing agent-based subsystems which are registered as dynamic services. It has the ability to collect local host information, and it's mainly Linux based.

JASMINA []. It is a set of monitoring tools that allows supervising a distributed infrastructure. It has the ability of running on any platform.

Zabbix []. It is an open source monitoring software for enterprise environment. It is available for Linux, Unix and Windows.

2.2.2.1.1 Cloud-specific monitoring

Amazon CloudWatch [2]**.** CloudWatch enables monitoring for Amazon cloud resources and services. CloudWatch provides monitoring metrics about CPU, disk, network, system status check and custom metrics such as memory and error rates. CloudWatch supports alarm for notification of predefined metric value. Graphs and statistics are provided for selected monitoring metrics. CloudWatch is neither open source nor free.

Azure Monitoring [3]**.** The key performance metrics for azure cloud services can be monitored in the Windows Azure Management Portal. It is possible to set the level of monitoring to minimal and verbose for each service role, and to customize the monitoring displays. Verbose monitoring data is stored in a storage account, which can be accessed outside the portal. Monitoring displays in the Management Portal are highly configurable. The user can choose the metrics that wanted to be monitored in the metrics list on the Monitor page, and the user can choose which metrics to plot in metrics charts on the Monitor page and the dashboard. The minimal metrics are limited to CPU Percentage, Data In, Data Out, Disk Read Throughput, and Disk Write Throughput.

Cloudify []. Cloudify is an open source PaaS for business-critical applications enabling on-boarding and scaling to any cloud. Cloudify supports monitoring system using a web management console or using the Cloudify shell. Cloudify in default can monitoring CPU and memory. It also supports monitoring probes and plugins to conduct monitoring, such as JMX. Alerts can be set inside the web management console.

Rackspace Cloud Monitoring []. Rackspace Cloud Monitoring is a service provided by Rackspace to monitor applications on cloud (not limited to Rackspace cloud). Rackspace Cloud Monitoring supports various monitoring metrics, including CPU, disk, memory, network, processes and custom metrics.

CopperEgg Reveal []. CopperEgg is a cloud computing systems management company and its products provide monitoring for websites, web applications, servers, and systems deployed in cloud. RevealCloud is a tool for server monitoring; RevealUpTime is a tool for website and web application monitoring; includes also custom monitoring metrics. CopperEgg products support almost all the major public cloud providers and platform. CopperEgg products are not free.

New Relic []. New Relic is an application performance management company. Its products provide user, application and server monitoring functionalities. Real user monitoring is able to provide real time performance metrics such as page load time, page views, Apdex score, identify poor performance pattern and alert and notification. Application monitoring is able to provide several metrics such as throughput, response time and Apdex score and transaction tracking and reporting. It also supports alerts and capacity analysis. Server monitoring provides performance metrics such as CPU, memory, network and IO. New Relic products are not free.

AppDynamics. AppDynamics is an application performance management company. Its products focus on providing performance management in cloud. AppDynamics provide real time monitoring and end user monitoring. AppDynamics is able to achieve troubleshooting by identifying bottlenecks, detecting transaction anomalies and code diagnostics. AppDynamics provides detailed report and visualised dashboard for statistics and comparison with each release. AppDynamics products are not free.

Beside the features of cloud-specific monitoring applications, there is no possibility of monitoring internal components, like the response time of the website pages that are hosted by a cloud provider.

2.2.3 Application-Level Monitoring

This section is devoted to application level Cloud monitoring, which is of significant importance, especially in the context of Cloud application SLA management. Due to virtualization as the basis for resource sharing, multiple virtual machines (VMs) can be run on a single physical machine or even multiple applications can be run on a single VM. As a result, per-application monitoring in such a shared environment is essential to keep applications healthy and guarantee QoS. Therefore, it is not enough just to monitor a physical machine or a VM to measure the application resource consumption, detecting SLA violations and managing resources efficiently.

RTCE [Hol10]. Run-time Correlation Engine (RTCE) provides a scalable log correlation, analysis, and symptom matching architecture that can perform real time correlation for large volumes of log data. Analysing the huge mass of data which is produced by software component, especially in heterogenous environment is a challeging issue. RTCE framework encompasses four functionalities: automatic data collection, data normalization into a common format, runtime correlation and analysis of the data to give a coherent view of system behaviour at run-time and a symptom matching mechanism identifying errors in the correlated data [Hol09].

Data from each application is read by Monitoring Agent (MA) and are routed in the form of events to the Evenvt Correlation Engine (ECE) via TCP/IP connection. After the data being procesed by ECE, they are presented on the web server, and users get them from different view. One of the limitation of RTCE is scalability in cloud computing environments.

A multi-layer approach for cloud application monitoring [Gon11]. A three-dimentional approach for cloud application monitoring is proposed in [Gon11], encompassing Local Application Survelliance (LAS), the Intra Platform Survelliance (IPS) and the Global Application Survelliance (GAS) dimensions. LAS monitors the application instance to check for rules violation; Its output is sent to the assigned IPS. The filtered results are then sent to the GAS components for more analysis.

Cloud Application Monitoring [Rak11]. The building of custom monitoring systems for Cloud applications is facilitated using the mOSAIC API. The mOSAIC approach as a whole contains four modules as the API, the framework (i.e., platform), the provisioning system, and the semantic engine. The API and the framework aim at the development of portable and provider independent applications. The provisioning system works at IaaS level and resource management. The functionality of the provisioning system is a part of the Cloud agency [Ven11]. The framework is a collection of predefined Cloud components in order to build complex applications. The framework constitutes a PaaS enabling the execution of complex services with predefined interfaces. The mOSAIC SLA management components are also part of the framework. The API offers the implementation of a programming model in a given language (currently Java, and Python in the future) to build applications. The API provides new concepts (e.g., the Cloudlet or the Connector) in order to focus on Cloud resources and communications instead of the resource access or communication details.

The mOSAIC monitoring API offers a set of connectors representing an abstraction of resource monitoring and a set of drivers implementing different ways of acquiring monitoring data (from different techniques); therefore, it supports monitoring by (i) offering a way to collect data directly from any of the components of a mOSAIC application, (ii) offering a way to collect data for any proposed monitoring techniques (accessing Cloud-provider, resource-related, and mosaic monitoring tools (called M/W - monitoring/warning - system), and (iii) supporting the mOSAIC Cloud application in order to access data regardless to the technology of the acquired resources and the way they are monitored. The aim of the set of mOSAIC monitoring tools, offered by the mOSAIC framework, is offering the ability to building up a dedicated monitoring system.

M4Cloud [Mas11]. It is a model-driven approach that classifies and monitors application-level metrics in shared environments such as the Cloud. The basis for the implementation of the monitoring phase is the Cloud Metric Classification (CMC). CMC identifies the following four models: application based (e.g., generic/specific), measurement based (e.g., direct/calculable), implementation based (e.g., shared/individual) and nature based (e.g., quantity/quality) models. The application based model supports the distinction of the metrics on the basis of the application they belong to.

The measurement based model is applied to define the formulas from which metrics can be calculated; the Implementation Based Model defines for each metric the corresponding measurement mechanisms, coherently with the formulas defined at the previous step; finally, the nature based model defines the nature of the metrics and their definition within SLAs. More info on the models could be found in the original article in [Mas11].

CMC is part of the M4Cloud framework. In this framework, the FoSII infrastructure [Bra10] is used as a Cloud Management System (CMS). Monitored data is analyzed and stored within a knowledge database and then is used for planning actions. Moreover, monitored data is also acquired and analyzed after the execution of such actions, for the purpose of efficiency evaluation.

Cloud4SOA [*].The Cloud4SOA monitoring offers a unified platform-independent mechanism, to monitor the health and performance of business-critical applications hosted on multiple Clouds environments in order to ensure that their performance consistently meets expectations defined by the SLA. In order to consider the heterogeneity of different PaaS offering Cloud4SOA provides a monitoring functionality based on unified platform independent metrics. The Cloud4SOA monitoring functionality allows to leverage on a range of standardized and unified metrics of different nature (resource / Infrastructure level, container level, application level, etc.) that, based on the disparate underlying cloud providers, allow the runtime monitoring of distributed applications so as to enforce the end-to-end QoS, regardless of where they are deployed across different PaaS. In the scope of Cloud4SOA several metrics have been defined from the cloud resource as well as the business application perspective, but not all of them have been enforced at runtime since they only provide useful information about the status of the application.

REMO [Men08]. Cost effectiveness and scalability are among the main criteria in developing monitoring infrastructure for large-scale distributed applications. REMO addresses the challenge of constructing monitoring overlays from the cost and scalability point of views jointly considering inter-task cost-sharing opportunities and node-level resource constraints. Processing overhead is modeled in this approach in a per message basis. A forest of optimized monitoring trees is deployed in the approach through iterations of two phases exploring cost sharing opportunities between tasks and refining the tree with resource sensitive construction schemes. In every iteration, a partition augmentation procedure is run generating a list of most promising augmentations for improving the current distribution of workload among trees, considering also the cost estimation for the purpose of limiting the list. Then, these augmentations are further refined through a resource-aware evaluation procedure and monitoring trees are built accordingly (through the resource-aware tree construction algorithm).

An adaptive algorithm is also considered for the purpose of balancing the cost and benefits of the overlay, which is useful especially for large-scale systems with dynamic monitoring tasks.

Planning the monitoring topology and collection frequency are important factors in keeping a balance between monitoring scalability and cost effectiveness. The drawback of proposed approaches up to date is that they either build monitoring topologies for each individual monitoring task (e.g., TAG [Mad02], SDIMS [Yal04], PIER [Hue05], join aggregations [Cor05], REED [Aba05], operator placement [Sri05]) or use a static one for all monitoring tasks [Sri05], which none of them is optimal. For instance, it could happen that two monitoring tasks collect data over the same nodes. Hence, in such a case it is more efficient to consider just one monitoring tree for data transmission, as nodes can merge updates for both tasks and reduce per-message processing overhead. Therefore, it is of significant importance to consider multi-monitoring-task level topology optimization for the purpose of monitoring scalability. Load management is another important factor to be considered in monitoring data collection, especially for data-intensive environments, meaning that the monitoring topology should be able to somehow control the amount of resources spent in order to collect and deliver the data. In the case of ignoring this fact, it may lead to overloading and consequently losing of data. Remo approach addresses all these issues considering node-level resources in building a monitoring topology and optimizing the topology for the purpose of scalability and ensuring that no node is assigned with monitoring workloads more than the amount that their available resources could support.

Three main advantages of this approach are as follows. At first, it identifies three critical requirements of large-scale application monitoring including sharing message processing cost among attributes, meeting node-level resource constraints and efficient adaptation based on monitoring task changes. Then after, a monitoring framework optimizing the monitoring topologies and addressing the above-mentioned requirements is proposed. Finally, techniques for runtime efficiency and support are developed as well.

In Table 1 a summary of the characteristic of the monitoring frameworks, reviewed in this section is reported, according to the classification dimensions introduced in Section 2.2.

Monitoring approaches	Who	What	When	How
COMPAS	Client/server provider	J2EE systems with EJB components	Periodical	Proactive
TestEJB	Client/server provider	J2EE systems with EJB components	Parallel with the application execution	Proactive
PAD	Service provider	Component-level	Parallel with the application execution	Proactive
Elastic multi-layer	Service provider	Multi-layer	Continuous at pre-defined time interval time interval	Proactive
RMCM	Service provider	Multi-layer	Continuous	Proactive/Reactive
mlCCL	Service provider	Multi-layer	Periodical	Proactive/Reactive
RTCE	Service provider	Comment-level	Periodical	Proactive
A multi-layer approach for cloud application monitoring	Service provider	Application-level	Continuous	Proactive
Cloud application monitoring	Service provider	Application-level	Event-based and at the predifined time interval	Proactive/Reactive
M4Cloud	Service provider	Application-level	On demand and periodically	Proactive/Reactive

REMO	Service provider	Application-evel	Collecting the metrics is not the scope of the work	Not part of the work

Table 1: Monitoring Frameworks

3 CHAPTER 3

The main methodologies that are used in this thesis are reviewed in the following chapter.

In 3.1 there is an overview of Aspect Oriented Programming and the review of current monitoring approaches based on that are provided.

Section 3.2, firstly exhibits a summary on Semantic Web then in the following there are more details on RDF streams, and C-SPARQL engine which is the core of the monitoring solution implemented by this thesis are provided. Also there is a revision over current stream monitoring approaches.

A general overview of the Palladio tool supports component-based application development is provided in Section 3.3.

Some explanations on our case study, MiC application and its Palladio meta-models, are given in Section 3.4.

In Section 3.5 we introduce MODAClouds project that is the main context of this thesis.

3.1 ASPECT ORIENTED PROGRAMMING

Aspect-Oriented Programming (AOP) was proposed to handle separation of concerns [Kiczales et al. 1997; Tarr et al. 1999], improving the modularity, and maintainability of an application, orthogonal concerns like logging, persistence, synchronization, failure handling.

Aspect-Oriented Software Development (AOSD) presents a new kind of module named *aspect*. In each aspect there are *pointcuts* and *advices*. A point during the execution of a method or the handling of an exception is called *join point*. A pointcut is an expression representing a set of join points. An advice describes the behavior of an aspect at a particular *join point*. When execution of a program reaches a join point, the control is transferred to related advice.

AOP supports both static and dynamic point cuts. Static pointcuts are events in the program execution that can be determined during compilation. Static point cuts do not support runtime adaption directly. Dynamic point cuts, instead, designate an execution point that cannot be decided at compile time; that means programmers are not obliged to determine the point cuts during compile time. The decision to execute an advice and the possibility to enable/disable could be done during the runtime. For example in AspectJ *if* and the *cflow* point cuts belong to this category. The *if* pointcut activates an advice whenever a condition is satisfied, and *cflow* pointcut activates an advice along the current control flow. Dynamic pointcuts are a fundamental mechanism used by dynamic AOP.

Weaving is the process of binding advices with the rest of the code. There are three different weaving. In compile-time weaving, sometimes referred to as *static* weaving, aspects are merged with the code-base during the compilation process. Load-time weaving enhances the code when the class is loaded in the virtual machine. In run-time weaving advices are woven during the execution of the application.

Dynamic AOP, is especially important for self-adaptive systems, refers to activating, configuring and removing aspects dynamically. Programmers can plug/unplug aspects during the program execution.

Dynamic support for AOP and weaving strategy are, in principle, orthogonal. An AOP implementation can hook all the possible join points statically and postpone at runtime their activation depending on the configuration of the active advices.

Different features of AOP make it suitable for using in different applications. Many researchers investigated AOP to implement *self-adaptive* systems, as they perceive many adaption concerns are crosscutting. At the same time, AOP has been used in the *service-oriented* application and *Web services* adaptability.

There are different AOP frameworks; among them we can name AspectJ, Spring AOP, and JBoss AOP. They are all supported by eclipse IDE. AspectJ language extension requires the use of an extended compiler and related tools, and has the lack of libraries. In JBoss advanced IDE features are not yet supported and its libraries are integrated with JBoss and JEMS. Spring AOP has the lack of IDE support for working with aspects, and it is integrated with Spring framework that makes it portable and adaptable for existing Spring users. A summary of the characteristics of these three frameworks is reported in Table 2.

AOP frameworks	Control flow	conditional	Weaving	IDE
AspectJ	cflow, cflow below	If	Compiled and load time	Eclipse, jdeveloper, jbuilder, netbeans
JBoss AOP	Via specified call stack	Via Dynamic cflow	Run time interception and proxies	Eclipse
Spring AOP	cflow	Custom pointcut	Run time interception and proxies	eclipse

Table 2: AOP Frameworks

The ability of AOP in modifying execution of an application by inserting or substituting the behaviors, and enable/disabling aspects on several execution points from one side and crosscutting aspect of monitoring at the other side, makes AOP as an effective way in the field of monitoring.

AOP is one of the possibilities that we propose for monitoring of Cloud Computing services, especially for PaaS layer, relying on the capabilities of AOP that is keeping separate the cross-cutting features like response time, and status code of a web page. In this case we are not forced to inject any additional modification to the current codes, and even we could add aspects to the current code without the detailed knowledge of the code itself. Also the feature of enable and disable of aspects helps us to do monitor whenever it is necessary.

3.1.1 Monitoring via Aspect Oriented Programming

For adding self-adaptive behavior or monitoring capabilities to an application, programmers do not need to know the structure of the code, and do any modifications, as the aspects are designed to be separated from the existing code and AOP enforce obliviousness. AOP developer only needs to specify the execution points in which advices must be triggered. In the following we review some of the monitoring approaches that are based on AOP.

In [Aspect EX 1], a fine-grain monitoring framework architecture has been proposed that called AOP-Monitoring framework. This framework is composed by two main modules: *Sensors* and the *Monitor Manager*. Sensors are assigned to collect data from the system, that are aspects. In this approach for every data or event that needs to be monitored, a sensor was considered. When a sensor collects enough data, this data is sent to the Monitor Manager for analyzing and making decision for the system. Monitor Manager is the responsible of collecting data from all the active sensors, and define policy to follow. The monitor manager uses data mining to predict failure or use the sensor to forecast methods. If method prediction needs more data, monitor manager can activate some more sensors or even deactivate others to obtain needed information. The monitor manager can also change the behavior of sensors in the case of redefine their elapse time between collecting data that made the system more flexible and adaptable and reduce the overhead when the behavior of the system seems correct. In Figure 3, the related architecture is shown in more details:

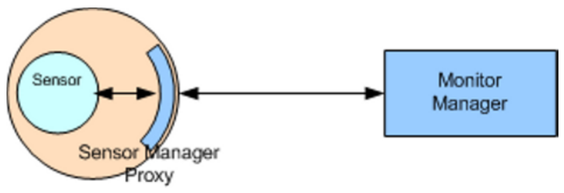


Figure 3: Detailed Architecture of a Sensor

As this picture shows, sensor architecture is contained real sensor and Sensor Manager Proxy. Sensor Manager Proxy can active or deactivate sensor, and change the time between monitor task, and communicate with Monitor Manager in order to send the received data that it has received from sensors and receive new orders from the Monitor Manager.

In the first version of the framework, two sensors have been developed, which they monitored the memory consumption and CPU status. For evaluating the overhead of using this framework, a simple web application on Tomcat Server 5.5.20 was run, and the two sensors were injected in every method call in the Tomcat Server core. The client workload for the experiment was generated with Httperf [Httperf]. This analysis showed that the throughput overhead introduced in the maximum point is around 6% (the number of requests were reduced from 422 req/sec to 395,7 req/sec) and the response time was increased around 16%.

In [IBM], a flexible and modular approach to a performance monitoring has been proposed that is a combination of AspectJ with JMX. They used the base idea and source code of the Glassbox [GlassBox]. In this work, the authors show how to gain statistics such as total counts, total response time for requests, and allow to drill down into information gathered from database.

AOP4CSM [AOP4CSM], is a Cloud service monitoring approach based on Aspect-Oriented programming. The main differences between this approach and related works in web service monitoring and Cloud services are as follow: AOP4CSM is a non-invasive approach that is the capability of AOP in terms of isolating aspects from source code; It does not modify the source code of the applications, and it is not based on special environment like Java-based application. This approach does not require any special hardware and/or configuration and could be installed on all Service-Oriented environments. AOP4CSM measures five QoS metrics: execution time, response time, communication time, throughput and availability. Regarding to invoke these parameters, specific join points are defined at important instants of time. To evaluate the performance of AOP4CSM, several experiments have been conducted in a private cloud on IaaS layer. These experiments are divided in measuring the computational overhead of AOP4CSM and its usefulness in the context of fault tolerance. For the first one, the response time of a cloud service was measured in two cases, with and without AOP4CSM. The average value of the AOP4CSM overhead is about 34 milliseconds. For the second case, the authors integrated that into fault tolerance framework and a cloud service based on a real life medical workflow, and their experiments showed that the failure rate has decreased about eleven percent.

SpringSource Hyperic [Hyp13] is a monitoring tool for managing various aspects of web applications. In particular, it provides support for real-time monitoring and, by default, visibility into *availability*, *performance*, *utilization*, and *throughput*. Hyperic is structured as a set of installable plugins, each one providing various metrics for a single application type. Hyperic supports the most popular application servers (e.g., JBoss, Apache Tomcat, IBM WebSphere, Microsoft .NET, etc.), mail servers (e.g., sendmail, postfix, Microsoft Exchange, etc.) and messaging middleware (e.g., RabbitMQ, ActiveMQ, Hadoop, etc.).

Finally, among the monitoring frameworks based on AOP, it is worth noticing Kieker [KIE13]. Kieker provides *monitoring probes* that can be added by adding annotations to the code, and supports Java-based systems although recently adapters for other type of systems have been added (i.e., .NET and COM). Kieker already comes with a set of predefined monitoring probes for the major metrics (i.e., response times, user sessions, CPU utilization, memory usage, etc.). Kieker moreover supports the Palladio component model [BEC09], it is able to derive Palladio models from the instrumented code and thus it takes advantage of the various analysis tools already available for this platform.

Its overhead has been shown to be minimal and it has been successfully used in production environments.

AOP-Monitoring framework is a flexible and adaptable fine grained monitoring framework. The ability to activate or deactivate sensors, sensor behavioral change at run time, and changing the monitoring level make this framework to have a better coverage of errors and failures than the traditional solutions. In the IBM research, an AOP-based system for performance monitoring was proposed that was designed to monitor multiple web applications simultaneously and provided correlated statistical results. SpringSource is another AOP based monitoring framework of web applications that is designed for most popular applications and mail servers. Among the frameworks that have been reviewed, AOP4CSM is a cloud service monitoring approach. This framework is a non-invasive monitoring approach that could be used in different domains like self-healing, fault tolerance, QoS management, load balancing and SLA management. In the domain of fault tolerance it was implemented on IaaS layer (Amazon EC2).

XXX TODO: Add a comparison Kieker with other and my approach XXX

3.2 Semantic Web

The Semantic Web is the extension of the World Wide Web that enables people to share content beyond the boundaries of applications and websites [semantic web 1]. It is a mesh of information linked up in such a way as to be easily readable by machines, on a global scale. It can be understood as an efficient way of representing data on the World Wide Web, or as a globally linked database. As shown in Figure 4, the Semantic Web is realized through the combination of certain key technologies [semantic web 1]. The technologies from the bottom of the stack up to the level of OWL have already been standardized by the W3C and are widely applied in the development of Semantic Web applications. The technologies are: Universal Resource Identifiers (URIs) provide means for uniquely identifying Semantic Web resources. The Semantic Web should be able to represent text documents in different human languages, and Unicode serves this purpose. XML provides an elemental syntax for content structure within documents. Resource Description Framework (RDF) is a framework for creating statements in the form of triples: subject – predicate – object. RDF Schema (RDFS) provides a basic schema language for RDF. For example, using RDFS it is possible to create hierarchies of classes and properties. Web Ontology Language (OWL) is used to formally define an ontology – "a formal, explicit specification of a shared conceptualization" [semantic web 2]. OWL extends RDFS by adding more advanced constructs to describe resources on the Semantic Web. By means of OWL and other ontology specification languages it is possible to explicitly define knowledge (i.e. concepts, relations, properties, instances, etc.) and basic rules in order to reason about this knowledge. OWL allows stating additional constraints, such as cardinality, restrictions of values, or characteristics of properties such as transitivity. It is based on Description Logics and thus brings reasoning power to the Semantic Web. Semantic Web Rule Language (SWRL) extends OWL with even more expressivity, as it allows defining rules so that whenever the conditions specified in the body of a rule hold, then the conditions specified in the head must also hold. SPARQL is an RDF query language - it can be used to query any RDF-based data, including statements involving RDFS and OWL.

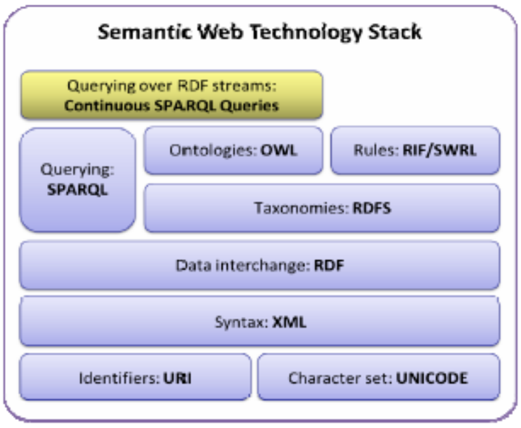


Figure 4: Semantic Web Technology Stack

3.2.1 C-SPARQL

In 2008, Della Valle et al. in [DEL09] called the Semantic Web community for techniques able to reason upon rapidly changing information. When reasoning on massive data streams, such as those characterizing the monitoring of cloud base applications, well known artificial intelligence techniques have the right level of expressivity, but their throughput is not high enough to keep pace with the stream (e.g., belief revision[GAE03]). The only technological solutions with the right throughput are Data Stream Management Systems (DSMS) [GAR07] and Complex Event Processing [LUC08], but, on the other hand, they are not expressive enough. A new type of inference engines is thus needed to reason on streams. [DEL09] named them *stream reasoners*.

In the following years, a number of stream reasoning approaches have been developed [BAR10,ANI11,TDO11,CAL10]. They share three main concepts:

1. they logically model the information flow as an RDF stream, i.e. a sequence of RDF triples annotated with one or more non-decreasing timestamps,
2. they process the RDF streams "on the fly", often by rewriting queries to the raw data streams, and
3. they exploit the temporal order of the streaming data to optimize the computation.

Within the monitoring systems of MODAClouds, that will be developed within this thesis, we propose to use Continuous SPARQL (C-SPARQL) [BAR10] – an extension of SPARQL that continuously processes RDF streams observed through windows (as done in DSMS). The current version of the C-SPARQL engine is an extension of SPARQL 1.1 and it includes support for all its constructs. The syntax and semantics of C-SPARQL were described in [BAR10b]. The C-SPARQL execution engine and its optimization techniques were illustrated in [BAR10c]. The optimization needed for high-throughput RDFS++ reasoning are described in [BAR10d]. Hereafter we provide a minimum guide to RDF streams and C-SPARQL syntax.

3.2.2 RDF Stream Data Type

C-SPARQL adds RDF streams to the data types supported by SPARQL1.1. An RDF stream is defined as an ordered sequence of pairs, where each pair is made of an RDF triple and its timestamp τ:

$$\ldots (\langle subj_i, pred_i, obj_i \rangle , \tau_i) (\langle subj_{i+1}, pred_{i+1}, obj_{i+1} \rangle , \tau_{i+1}) \ldots$$

Timestamps can be considered as annotations of RDF triples; they are monotonically non-decreasing in the stream ($\tau_i \leq \tau_{i+1}$). They are not strictly increasing because timestamps are not required to be unique. Any (unbounded, though finite) number of consecutive triples can have the same timestamp, meaning that they "occur" at the same time, although sequenced in the stream according to some positional order.

Example. In our running example, taken from monitoring system scenario, data streams are associated with http-status. In the streams every triple corresponds to the page-url that return a specific http-status. The predicate of the triple (t:returnedBy) is fixed, while the subject (?httpstatus) and object (?requestedurl) parts of the triple are variable. Thus, a physical source for this stream has items consisting of pairs of values. This arrangement is coherent with RDF repositories whose predicates are taken from a small vocabulary constituting a sort of schema, but C-SPARQL makes no assumption on variable bindings of its stream triples. An example of stream of status code of five requested urls.

Triple	Timestamp
c:402 t:returnedBy " http://myservlet.cloudapp.net/mic/"	t100
c:403 t:returnedBy " http://myservlet.cloudapp.net/mic/Register.jsp"	t101
c:404 t:returnedBy "http://myservlet.cloudapp.net/mic/Questions"	t102
c:403 t:returnedBy " http://myservlet.cloudapp.net/mic/SaveAnswer"	t103
c:401 t:returnedBy " http://myservlet.cloudapp.net/mic/Login"	t104

3.2.3 Data Sources and Time Windows

The introduction of data streams in C-SPARQL requires the ability to identify such data sources and to specify selection criteria over them. As for identification, we assume that each data stream is associated with a distinct IRI, that is a locator of the actual data source of the stream. More specifically, the IRI represents an IP address and a port for accessing streaming data. As for selection, given that streams are intrinsically infinite, we introduce the notion of windows upon streams, whose types and characteristics are inspired by those defined for relational streaming data.

Identification and selection are expressed in C-SPARQL by means of the FROM STREAM clause. The syntax is as follows:

```
    FromStrClause → 'FROM' ['NAMED'] 'STREAM' StreamIRI '[ RANGE'
Window ']'
    Window    → LogicalWindow | PhysicalWindow
    LogicalWindow → Number TimeUnit WindowOverlap
    TimeUnit    → 'ms' | 's' | 'm' | 'h' | 'd'
    WindowOverlap → 'STEP' Number TimeUnit | 'TUMBLING'
    PhysicalWindow → 'TRIPLES' Number
```

A window extracts the last data elements from the stream, which are the only part of the stream to be considered by one execution of the query. The extraction can be physical (a given number of triples) or logical (all triples occurring within a given time interval, whose number is variable over time).

Logical windows are sliding if they are progressively advanced by a given STEP (i.e., a time interval that is shorter than the window's time interval). They are non-overlapping (or TUMBLING) if they are advanced in each iteration by a time interval equal to their length. With tumbling windows every triple of the stream is included exactly into one window, whereas with sliding windows some triples can be included into several windows.

The optional NAMED keyword works exactly like when applied to the standard SPARQL FROM clause for tracking the provenance of triples. It binds the IRI of a stream to a variable which is later accessible through the GRAPH clause.

Example. A monitoring system query counts the number of pages returning http-statuses; the query considers the last 10 minutes, while the sliding window is modified every minute.

```
PREFIX mc: <http://www.modaclouds.eu/monitoring#>
SELECT DISTINCT ?httpstatus (COUNT(?requestedurl) AS ?pagecount)
FROM STREAM <http://stream.org/monitoredhttpstatus.trdf>[RANGE 10 MIN STEP
1 MIN]
WHERE { ?httpstatus mc:returnedBy ?requestedurl . }
GROUP BY ?httpstatus
```

The query is executed as follows: first, all pairs of status and url are extracted from the current window over the stream, then the total number of url for each status is counted into the new variable pagecount and every pair is extended into a triple, and finally the triple is projected as distinct pairs of status and number of url. The window considers all the stream triples in the last 10 minutes, and is advanced every minute. This means that at every new minute new triples enter into the window and old triples exit from the window. Note that the result of the aggregation does not change during the slide interval, therefore also the query result does not change during the slide interval; it changes instead at every slide change.

3.2.4 C-SPARQL Queries

All queries over RDF data streams are denoted as continuous queries, as they continuously produce output in the form of tables of variable bindings tables or RDF graphs. Each C-SPARQL query is registered through the following statement:

Registration → ‘REGISTER QUERY’ QueryName [‘COMPUTED EVERY’ Number TimeUnit] ‘AS’ Query

Only queries in the CONSTRUCT and DESCRIBE form can be registered as generators of RDF streams, as they produce RDF triples, associated with a timestamp as an effect of the query execution. The optional COMPUTED EVERY clause indicates the frequency at which the query should be computed. If no frequency is specified, the query is computed at a frequency that is automatically determined by the system.

Example. Assume that a (classic, static) RDF repository stores (a) the page hosted in a http-server, (b) the URL of each page, and (c) the virtual machine containing http-server. We now show a query that combines static knowledge (from the repository) and dynamic knowledge (from the streaming data) in order to periodically count how many http-statuses have returned from each VM in the last 30 minutes. In this example the window is sliding with a step of five minutes. From now on, the c: and t: prefixes are omitted for brevity.

```
REGISTER QUERY httpstatuspervm COMPUTED EVERY 5 MIN AS
SELECT DISTINCT ?vm (COUNT(?httpstatus) AS ?httpstatuscount)
FROM STREAM <http://stream.org/monitoredhttpstatus.trdf> [RANGE 30 MIN STEP
5 MIN]
FROM <http://www.modacloud.eu/monitoringsystemmap.rdf>
WHERE { ?httpStatus mc:returnedBy ?url .
?page mc:hasurl ?url .
?page mc:IsIn ?httpserver .
?vm mc:contains ?httpserver .
}
```

The query is executed as follows. As in the previous query, all pairs of bindings of http-status and url are extracted from the current window over the stream, and joined to a graph pattern used to extract from the RDF repository the pair of bindings of URL with their pages, the pages with their http-server and, finally, the http-server with their VM. Then, the number of http-statuses returned by the page in each VM is counted into the new variable passages. Finally, pairs of distinct VM and http-status count are projected.

3.2.5 Stream Registration

C-SPARQL allows the production of RDF streams, registered through the following statement:

Registration → 'REGISTER STREAM' QueryName ['COMPUTED EVERY' Number TimeUnit] 'AS' Query

Only CONSTRUCT and DESCRIBE queries can be registered as RDF streams, as they produce RDF triples that are associated with a timestamp as effect of the query execution. Every query execution produces from a minimum of one triple to a maximum of an entire RDF graph, depending on the query construction pattern. In the former case, a different timestamp is assigned to every triple. In the latter case, the same timestamp is assigned to all the triples of the constructed graph. Still, the system-generated timestamps are in monotonic non-decreasing order.

Example. The following example allows the construction of new RDF data streams, by registering CONSTRUCT queries. Consider the previous query in which the projected pairs of districts and passages are used to construct a triple. The corresponding C-SPARQL query is:

```
        REGISTER STREAM httpstatuspervm COMPUTED EVERY 5 MIN AS
        CONSTRUCT {?vm mc:returned ?httpstatuscount}
        FROM STREAM <http://stream.org/citytollgates.trdf> [RANGE 30 MIN STEP 5 MIN]
WHERE
        {
          SELECT ?vm (COUNT(?httpstatus) AS ?httpstatuscount)
          WHERE {
               ?httpStatus mc:returnedBy ?url
               ?httpStatus a mc:clientErrors
               ?page mc:hasurl ?url
               ?page mc:IsIn ?httpserver
               ?vm mc:contains ?httpserver
              }
        }
```

This query uses the same logical conditions as the previous one, but using the inference, extracts only the http-statuses of the client-error class (and its subclasses, that could be found in the ontology, e.g. 400, 401, 402, 403, 404 etc.) and constructs the output in the format of a stream of RDF triples. Every query execution may produce from a minimum of one triple to a maximum of an entire graph. In the former case, a different timestamp is assigned to every triple. In the latter case, the same timestamp is assigned to all the triples of the graph. In both cases, timestamps are system-generated in monotonic order. Results of two evaluations of the previous query are presented in the table below.

triple	**Timestamp**
c:303 t:returnedBy "http://www.modaclouds.eu/url1"	t400
c:404 t:returnedBy "http://www.modaclouds.eu/url2"	t400
c:403 t:returnedBy "http://www.modaclouds.eu/url3"	t401
c:404 t:returnedBy "http://www.modaclouds.eu/url4"	t401
c:301 t:returnedBy "http://www.modaclouds.eu/url5"	t401

The first evaluation occurs at t400. Suppose that only data from two sources (i.e., c:303 and c:404) are present in the window. Then, the evaluation generates two triples with the same timestamp (i.e., t400).

The second evaluation occurs at t401. Suppose that part of the data elaborated by the previous query is still in the window and that new data related to c: 403 entered in the window. Then, the evaluation produces 3 triples; all of them have the same new timestamp (i.e., t401).

3.2.6 Stream monitoring approaches

In the context of semantic web, we presented some relevant monitoring approaches that are based on the idea of using semantic query language and engines.

Linke Stream Middleware (LSM) [Stream Related work 1] is a platform in the sensor fields which collects data from sensors, time dependent or time streams, and analyzes them. Its layer architecture contains: i) Data Acquisition, which collect data from different wrappers that are plugable during run-time. The format of input data are different from one wrapper to the another one, but the Mediate Wrappers use data transformation rules to map data in the RDF format; ii) Linked Data Layer, in this layer the data is composed by adding global identifiers to the data items and an ontology that capture the data model, (here Semantic Sensor Network ontology has been used). iii) Data Access layer, queries over the Linked Data Layer are executed in the layer. For this purpose SPARQL query language is used for sensor meta data and CQELS engine is used stream data; iv) Application Layer, the ability of the query processing of previous layer powered this layer, which simplify the development of the applications with a SPARQL end point. The output results from the queries are sent to a chosen channel.

In [Stream Related work 2] the authors have presented a framework to introduce self-management in Platform-as-a-Service environments. Their framework is based on the concept of viewing cloud platform as a sensor network. They proposed to employ techniques from the Semantic Sensor Web (SSW) [22SSWA]. The SSW technology, namely RDF data streams and SPARQL query engine, has been used for the challenge of processing multiple heterogeneous data streams. They also followed the IBM MAPE-K (Monitor, Analyze, Plan, and Knowledge) reference model to create adapt loops. The high-level architecture of their framework, as Figure X shows, contains 3 main elements: triplication engine, continues SPARQL query engine and OWL/SWRL reasoning engine. The triplication engine is responsible for consuming and “homogenizing” the data generated by deployed applications, platform components, external services, etc. The engine takes as input streams of raw data, and generates streams of RDF triples. Continuous SPARQL query engine takes as input the flowing RDF data streams generated by the triplication engine and evaluates pre-registered continuous SPARQL queries against them, to support situation assessment. The OWL ontologies and SWRL rules provide expressivity for adaption policies. When a critical situation is discovered, an adaption plan has to be generated, that is not just association of event and condition, but more complex reasoning. Authors have proposed a scenario where a number of applications are deployed on cloud platform, where the response time of the notification service is monitored. The related data sent to the C-SPARQL engine in RDF format. Based on the defined C-SPARQL queries in the case of detecting critical situation, the possible reactions will be performed.

Both of the stream monitoring approaches that have been discussed are based on SSN ontology, while in this thesis a specific ontology is defined for monitoring cloud resources. In LSM the CQELS engine was used to analyze the stream while in the second approach, and in this thesis, C-SPARQL engine is used. In [], the authors compared C-SPARQL engine versus CQELS engine, and showed that C-SPARQL has a larger set of functionalities and is more flexible.

3.3 PALLADIO

In this section, we will review the Palladio tool that is a software tool supporting component-based application development. Palladio is the starting point that will be extended during this thesis in order to allow the definition of monitoring rules with a white-box approach.

The aim of Palladio is allowing performance metrics analysis of component-based applications depending on the underlying hardware infrastructure. The framework comprises the Palladio Component Model (PCM), a component-based development, a software architecture simulator (called SimuCom) and an Eclipse tool (Palladio-Bench). Additionally, other analytical solvers and simulation tools are supported (i.e. LQNS, LQSIM).

The Palladio Component Model is composed of several meta-models describing various aspects of a component-based application and defined by different kinds of users. In particular, four roles are defined (Figure 5) [1]:

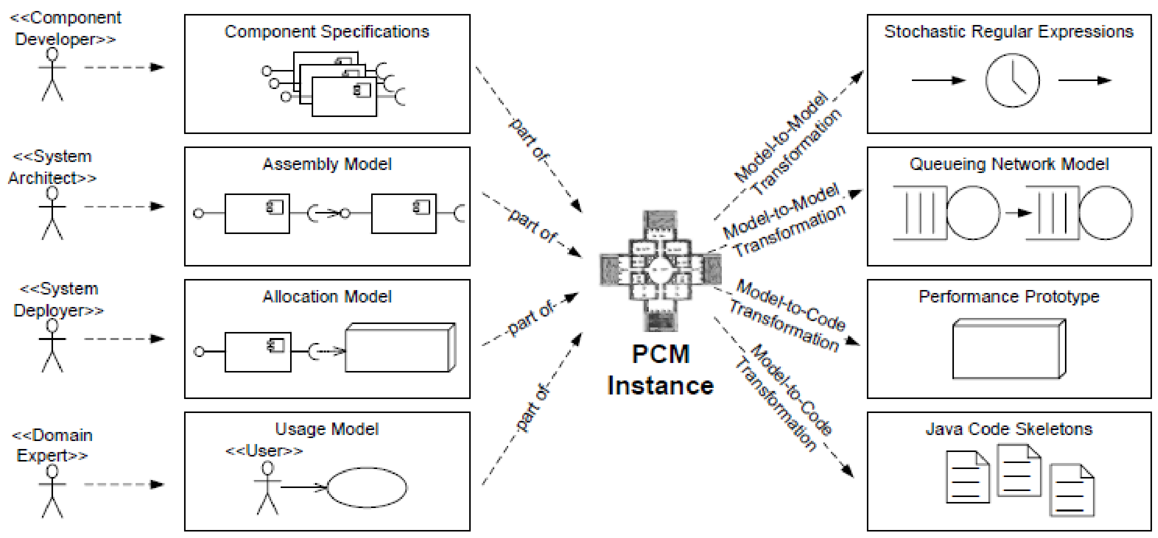


Figure 5: Palladio - Developer Roles in the Process Model

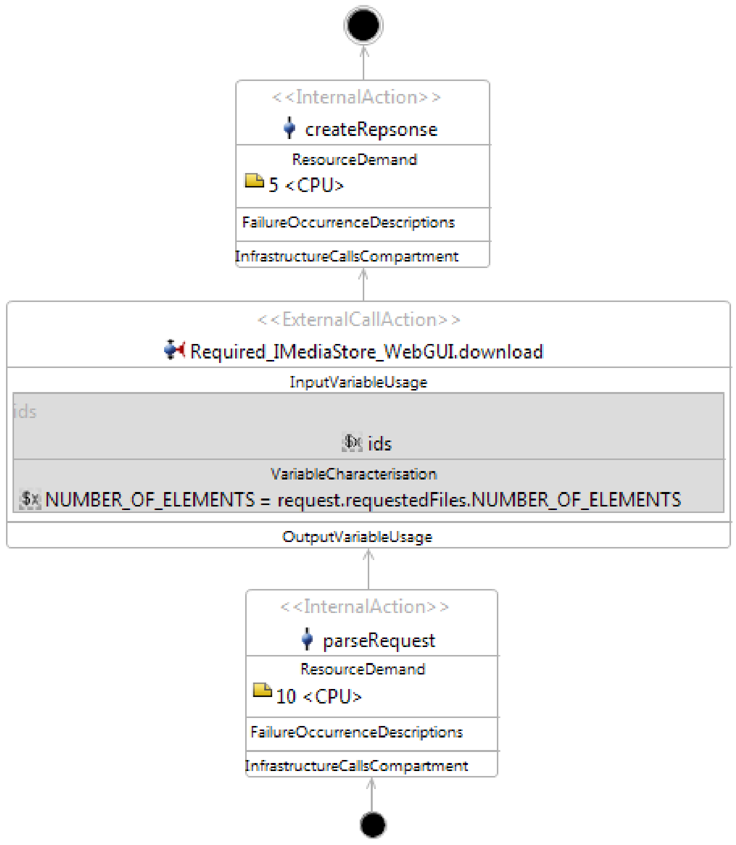
- Component Developer: She/he defines the Repository Model implementing all the available components.
- System Architect: She/he builds up the general software architecture (System/Assembly Model) using components picked up from the Repository Model.
- System Deployer: She/he defines the Resource Model representing the features of the hardware resources hosting the application. Then the Resource Model is used to define the Allocation Model that specifies which components will run on which hardware resources.
- Domain Expert: She/he defines the Usage Model representing how users interact with the application.

The development process

Palladio Framework is available in a self-contained Eclipse Juno distribution called Palladio Bench3. The aim of this section is to express the process to develop the meta-models representing the software system. The first step consists of defining a set of abstract components and interfaces. This task is performed by the Component Developer. Abstract components are defined in terms of required and/or provided interfaces that are saved into the Repository Model and can be used to build up system architectures. Components and interfaces are later connected through Requires and/or Provides relations.

Interfaces are used to define the list of methods provided by the components and are linked to them by Provides relations. Furthermore, when methods are used by other components they are linked to the corresponding interfaces by Requires relations. Components can also contain information about the dependency by some parameters which are related to input variables or objects.

The Component Developer defines also the so-called Resource Demand Service Effect Specification (RDSEFF or SEFF) for the methods/actions provided by each component, so for example in Figure 6 a WebGUI component contains the TTPDownload SEFF, which is shown in Figure 6.



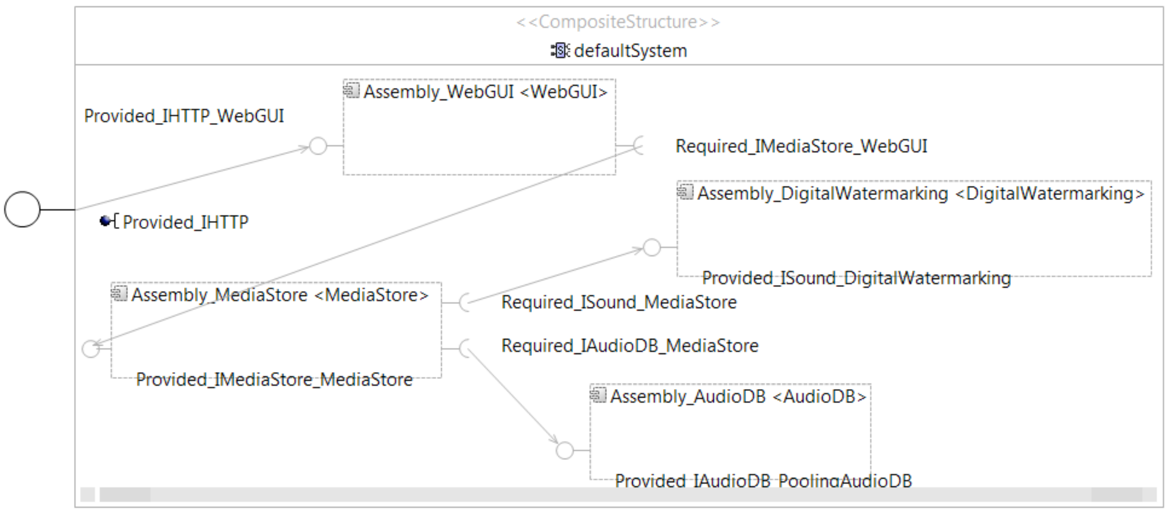
<<InternalAction>>
createRepsonse
ResourceDemand
5 <CPU>
FailureOccurrenceDescriptions
InfrastructureCallsCompartment
<<ExternalCallAction>>
Required_IMediaStore_WebGUI.download
InputVariableUsage
ids
ids
VariableCharacterisation
NUMBER_OF_ELEMENTS = request.requestedFiles.NUMBER_OF_ELEMENTS
OutputVariableUsage
<<InternalAction>>
parseRequest
ResourceDemand
10 <CPU>
FailureOccurrenceDescriptions
InfrastructureCallsCompartment

Figure 6: Palladio example, HTTPDownload SEFF

A RDSEFF is quite similar to an activity diagram/graph composed of a flow of actions linking a starting point (represented as a bullet) to an end point (represented as a circled bullet). Actions can be internal (Internal Action) or external (External Call Action) depending on whether they are defined within the component or they are calls to operations defined by external components. In the first case, the Component Developer can specify the units of low-level resources (such as CPU and Storage) used by the action, while in the second case he/she can specify some parameters useful to determine the resource usage on the external components side. Furthermore, resources utilization can be specified using deterministic or stochastic expressions. Stochastic expressions allow defining discrete or continuing distribution functions. Finally, within the RDSEFF it is also possible to specify probabilistic branches, each of which in turn is defined by an activity graph.

The main task of the System Architect is to builds up the System Model by picking the needed components from the repository and defining the connections between them. Each component can be reused several times within the system. Moreover, components can be included in the Repository Diagram, but their use is not mandatory in the system. This is due to the fact that the System Architect may want to create several versions of the system differing each other for some components in order to compare the performance of the architectures. The system model is characterized by System Interfaces, exposed to users (provided interfaces) or to other external systems (provided and/or required interfaces). Figure 7 shows the graphical representation of the System Model of the Media Store example (one of the Palladio's built-in examples). The system provides the IHTTP interface which is directly derived from the IHTTP interface provided by the WebGUI component. So, the system is composed by the following components: WebGUI, MediaStore, AudioDB and DigitalWatermarking. They are combined together through their interfaces in order to realize a composite service which is provided by the system through the IHTTP interface. In this way, the system becomes a black-box for the users, which access it only through the IHTTP interface.

The System Deployer, instead, defines a set of suitable hardware resources specifying their characteristics in the Resource Model and provides the Allocation Model. The later model is in charge of defining the resource each component has to be allocated on.



<<CompositeStructure>>
defaultSystem
Assembly_WebGUI <WebGUI>
Provided_IHTTP_WebGUI
Required_IMediaStore_WebGUI
Assembly_DigitalWatermarking <DigitalWatermarking>
Provided_IHTTP
Provided_ISound_DigitalWatermarking
Assembly_MediaStore <MediaStore>
Required_ISound_MediaStore
Required_IAudioDB_MediaStore
Provided_IMediaStore_MediaStore
Assembly_AudioDB <AudioDB>
Provided_IAudioDB_PoolingAudioDB

Figure 7: Palladio - Media Store Example, System Diagram

The Palladio framework does not provide any mechanism to define quality of service constraints or monitoring rules for the application under development. These issues will be the focus of the next chapter.

3.4 Meeting in the Clouds

MiC is a social networking web application [CPIM]. It allows a user to register and to choose her/his topics of interest providing a grade in the range 1-5. At the end of the registration process, MiC identifies the most similar users in the social network according to the registered user's preferences, in particular, similarity is computed through the *pearson coefficient (Pearson, 2010)*. After registration, the user can enter into the MiC portal and can interact with his "Best Contacts" writing and reading on the selected topics. More precisely, the application is composed of a front-end developed as JSP and servlets and a back-end developed as a CPIM (*Cloud Provider Independent Model*) CloudTask. The front-end and back-end are decoupled by a task queue and shared user profiles through the SQL Service. The same service also stores messages, and best contacts that are accessed by the front-end. The Blob Service is used to store pictures while the NoSQL Service stores user interests and preferences. Both are accessed by the front-end. Finally, the Memcache Service is used to temporarily store the last retrieved user profiles and best contacts messages with the aim of improving the response time of the whole application.

MiC application has been deployed on Google App Engine (GAE), Azure, and Amazon EC2. In Chapter 5, the overhead of the monitoring framework that is developed in this work be assessed by performing analysis on Azure and Amazon EC2.

As discussed before in Section 3.3, Palladio tool provides meta-models that are representing the software system. The information that is provided by these meta-models are employed for defining the Knowledge Base which is next used as the supporting information for the C-SPARQL engine. The mentioned knowledge base is provided by extending the core ontology for a specific application and extracting design time information that will be added to the permanent RDF data.

In the next parts of the present section, the Palladio models of the MiC application and the procedure of definition of the extended ontology will be defined; while section 4.7 will be devoted to the explanation of the permanent information.

The Palladio meta-models are also used as the extension for the specification of the Quality of Service (QoS) constraints developed in this thesis. The fact which will be discussed more in details in the next section.

The information that is provided by Palladio meta-models of MiC application contains components and resources of the MiC and the way the components are allocated to the resources.

Figure 11 shows the graphical representation of the Repository Model of MiC, displaying eight interfaces and components listed below:

- Components: Frontend, Backend, Blob, NoSQL, SQL, Memcache, TaskQueue, Mailing
- Interface: MiC IF, Backend IF, Blob, NoSQL, SQL, Memcache, TaskQueue, Mailing

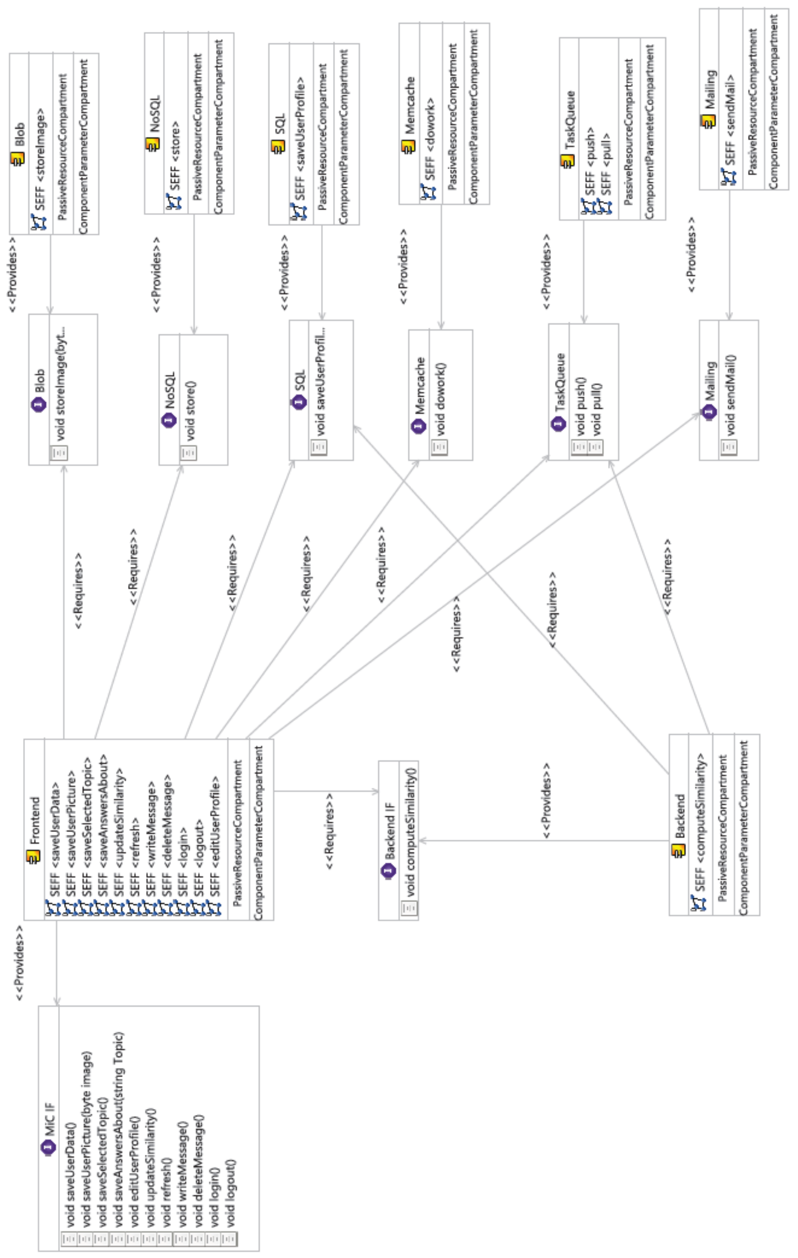


Figure 8: RipositoryDiagram of Mic

Components and interfaces are connected through <<Requires>> and/or <<Provides>> relations, as it is illustrated in the Figure . Interfaces are used to define the methods that are provided by the components and are linked via <<Provides>> relations. Therefore, the Frontend component as instance provides the methods (which correspond to the main JSP pages of the application) *saveUserdata, saveUserPicture, saveSelectedTopic, saveAnswersAbout, editUserProfile, updateSimilarity, refresh, writeMessage, deleteMessage, login and logout* through MiC interface, and requires the methods within *Backend, Blob, NoSQL, SQL, Memcache, TaskQueue, and Mailing* components.

Figure 12 shows the graphical representation of the *System Model* of the MiC application. The system provides MiC interface which is directly derived from the MiC interface which is provided by Frontend component.

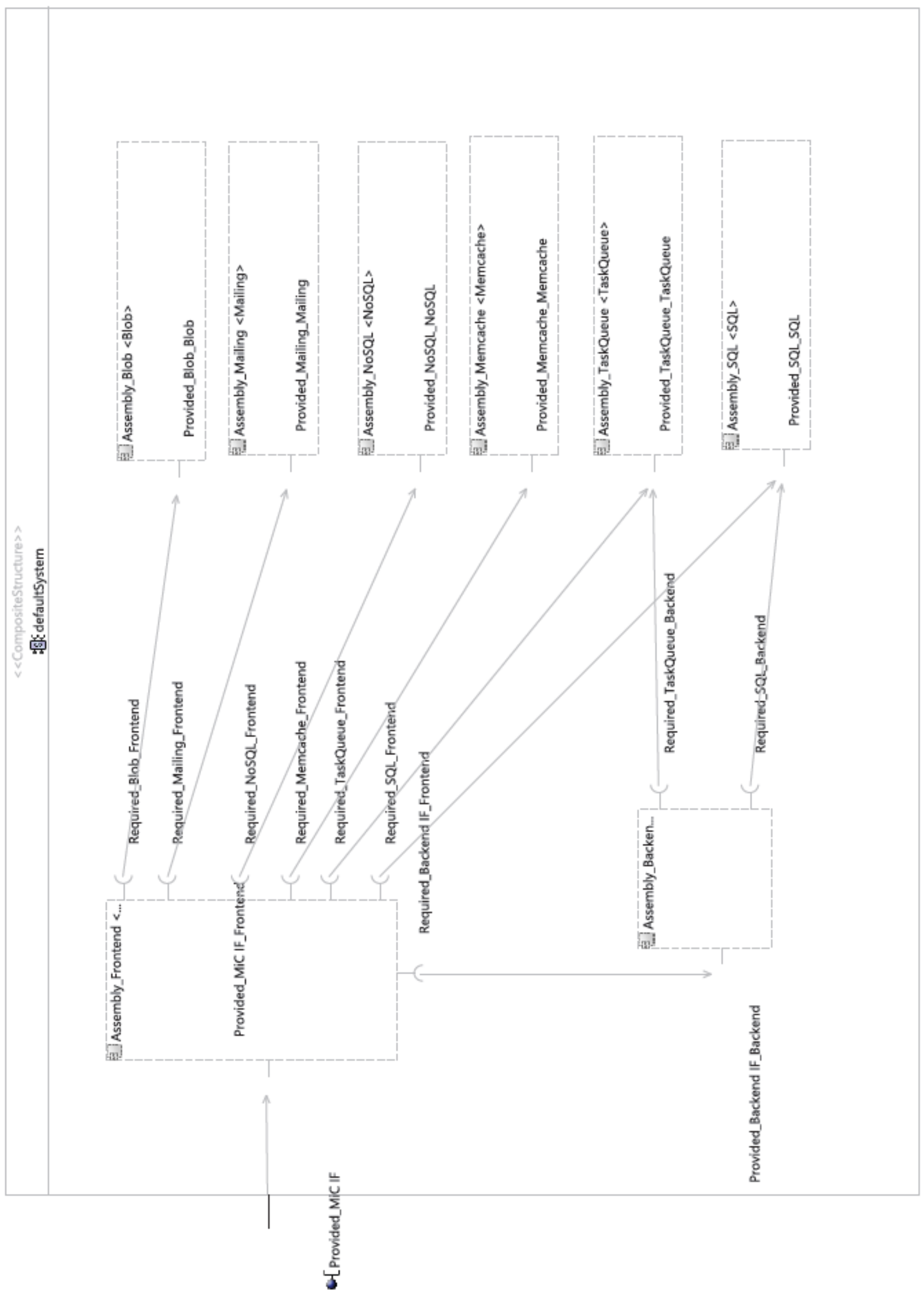


Figure 9: System model of MiC

All the required hardware resources of MiC are defined in Resource Environment diagram.

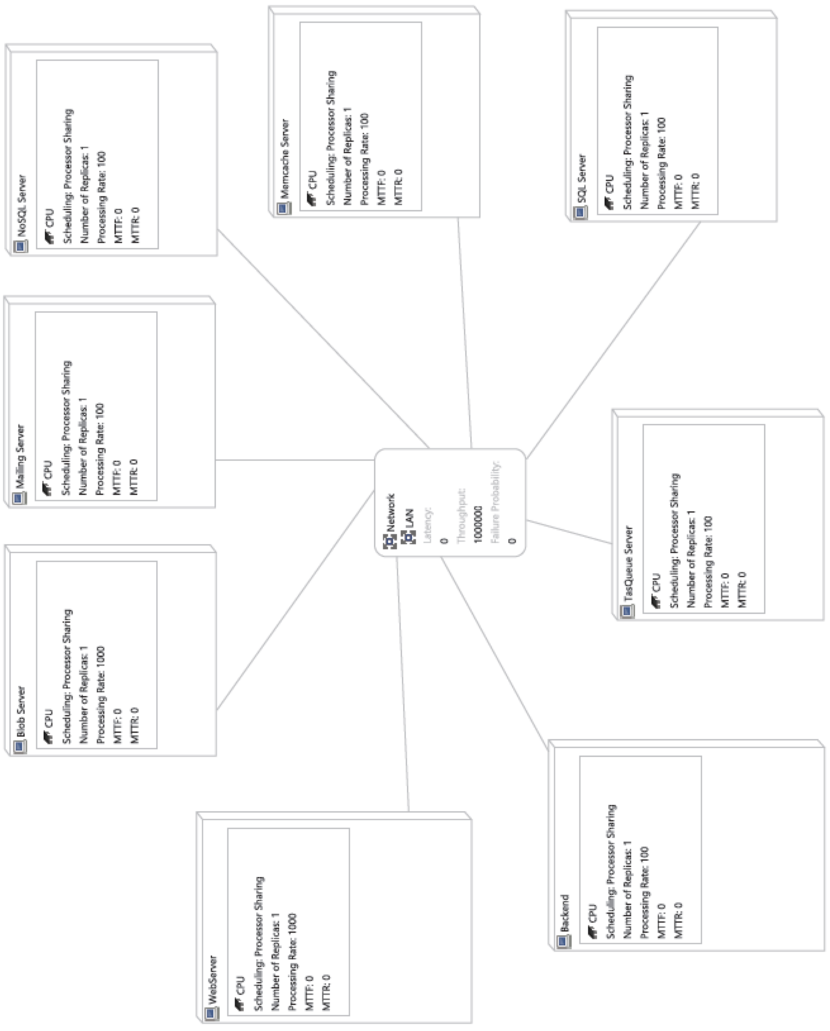
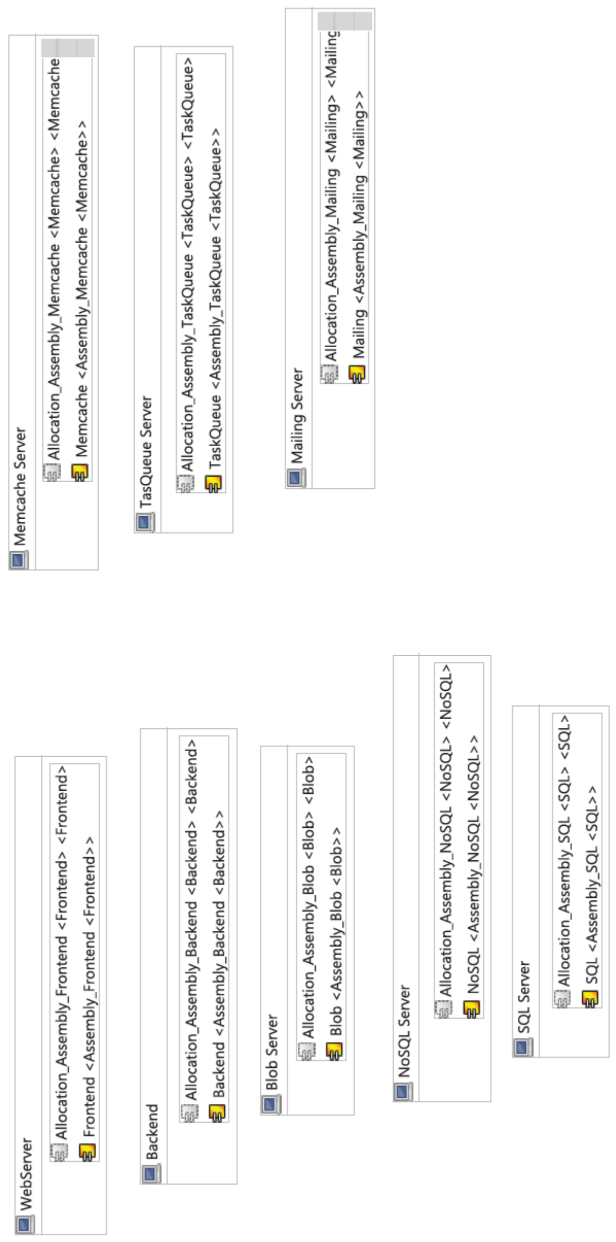


Figure 10: ResourceEnviornment of MiC

Finally the allocation diagram, Figure 14, describes how the components are allocated to the hardware resources. In our case, each of the components is allocated in a separate resource; while the servers and components are called *resource container*, *allocation context* respectively.



Memcache Server
Allocation_Assembly_Memcache <Memcache> <Memcache
Memcache <Assembly_Memcache <Memcache>>
TasQueue Server
Allocation_Assembly_TaskQueue <TaskQueue> <TaskQueue>
TaskQueue <Assembly_TaskQueue <TaskQueue>>
Mailing Server
Allocation_Assembly_Mailing <Mailing> <Mailing
Mailing <Assembly_Mailing <Mailing>>
WebServer
Allocation_Assembly_Frontend <Frontend> <Frontend>
Frontend <Assembly_Frontend <Frontend>>
Backend
Allocation_Assembly_Backend <Backend> <Backend>
Backend <Assembly_Backend <Backend>>
Blob Server
Allocation_Assembly_Blob <Blob> <Blob>
Blob <Assembly_Blob <Blob>>
NoSQL Server
Allocation_Assembly_NoSQL <NoSQL> <NoSQL>
NoSQL <Assembly_NoSQL <NoSQL>>
SQL Server
Allocation_Assembly_SQL <SQL> <SQL>
SQL <Assembly_SQL <SQL>>

Figure 11: Allocation Diagram of MiC

Extended Ontology for MiC application

For each application, related ontology will be designed by extending the core ontology, based on the information supplied by Palladio Diagrams.
Resource containers represent either platform or infrastructure and can be defined through the resource environment meta-model. Repository Model, describes all the application components of the case study.

In our case study, MiC application, the following applications are included:

- Applications: Frontend, Backend, Blob, NoSQL, SQL, Memcache, TaskQueue, Mailing

Each component requires a different class which extends the Software class in the ontology. As the resource container shows, for MiC application eight platform components are considered:

- *Platform: WebServer, Blob Server, Mailing Server, NoSQL Server, Memcache Server, SQL Server, TaskQueue Server, Backend*

3.5 MODACLOUDS

Finally in this section, we provide an overview of the MODAClouds FP7 IP project which provides the context of this thesis development.

Current Cloud's offer is becoming day by day wider providing a vibrant technical environment, where SMEs can create innovative solutions and evolve their services. Cloud promises cheap and flexible services to end-users at a much larger scale than before. However, Cloud business models and technologies are still in their initial hype and characterized by critical early stage issues, which pose specific challenges and require advanced software engineering methods.

The main goal of MODAClouds, Model-Driven Approach for design and execution of applications on multi clouds, is to provide methods, a decision support system, an open source IDE and run-time environment for the high-level design, early prototyping, semi-automatic code generation, and automatic deployment of applications on multi-Clouds with guaranteed QoS. Model-driven development combined with novel model-driven risk analysis and quality prediction will enable developers to specify Cloud-provider independent models enriched with quality parameters, implement these, perform quality prediction, monitor applications at run-time and optimize them based on the feedback, thus filling the gap between design and run-time. Additionally, MODAClouds provides techniques for data mapping and synchronization among multiple Clouds.

MODAClouds innovations are:

- simplify Cloud provider selection favoring the emergence of European Clouds,
- avoid vendor lock-in problems supporting the development of Cloud enabled Future Internet applications,

- Provide quality assurance during the application life-cycle and support migration from Cloud to Cloud when needed.

The monitoring framework that will be developed by this thesis will provide the enabling mechanism to trigger runtime adaption and to full the gap between design-time modules and run-time application behavior.

4CHAPTER 4

The QoS (Quality of Service) management of applications that can execute on multiple Clouds requires the ability to collect metrics on the application state and the processed workload, and use these metrics to decide at runtime adaptations to ensure high quality of service.
The main objective of the present thesis is to develop a monitoring framework for the MODAClouds project.
Section 4.1 provides an overview of the MODAClouds project run-time environment.
In Section 4.2 the monitoring architecture of MODAClouds is reviewed; while in section 4.3 the MODAClouds core ontology is described. Section 4.4 is focused on how the Palladio meta-model is extended for the specification of the of the QoS constraints; the constraints which are subsequently employed in Section 4.5 for the specification of the monitoring rules.
In Section 4.6, the approach for the definition of knowledge base information generated during the design time and run time is explained. Section 4.7 describes the mapping of the monitoring rules to C-SPARQL queries. Finally, section 4.8 is dedicated to the Explanation of the Data Collectors that are designed in the present work.

4.1 MODACLOUDS RUN-TIME ENVIORNMENT

MODAClouds includes a *Runtime Environment* that will implement the MAPE-K loop (monitor, analysis, planning, execution, knowledge) reference blueprint for the implementation of self-adaptive applications. The *Monitoring Platform* will be the system, inside the *Runtime Environment*, playing the main monitoring and analysis roles of the MAPE-K loop. These two capabilities may be summarized as follows:

- *Monitoring***:** The monitoring component is responsible for managing the different probes that provide information regarding the performance of the system. In MODAClouds, probes can capture the current consumption of critical node resources (such CPU and memory) but also other performance metrics (e.g., number of processed requests in a time window and the request process latency). The monitoring granularity is specified by rules. Probes can also raise notifications when the system configuration changes.
- *Analysis***:** The analysis component is responsible for processing the information captured by the monitoring component and to generate high level events or high-level aggregate information. For instance, it may combine the values of CPU and memory utilization to signal an overload condition in the platform.

Figure 8 illustrates the *Monitoring Platform* role in the *Runtime Environment* envisioned in the MODAClouds project. The *Monitoring Platform* will receive a set of monitoring rules from the design-time environment (*MODAClouds IDE*) that will specify *what* to measure and will allow to customize *how* to measure it (e.g., monitoring resolution). The *Monitoring Platform* will be then responsible for obtaining information from the running application from its cloud environment and analyze this data in order to achieve two main objectives: 1) generating triggers and data streams for the *Self-Adaptation Platform* of the *Runtime Environment*; 2) Sending relevant monitoring data to the design-time IDE as a feedback, in order to provide the decision support and insights on application behavior at runtime for the application developer and the application *QoS Engineer*. The last scenario also includes the case where the application has been deployed on a sandbox, i.e. a controlled test environment, with the aim of calibrating the design-time tools and models based on the initial runtime information.

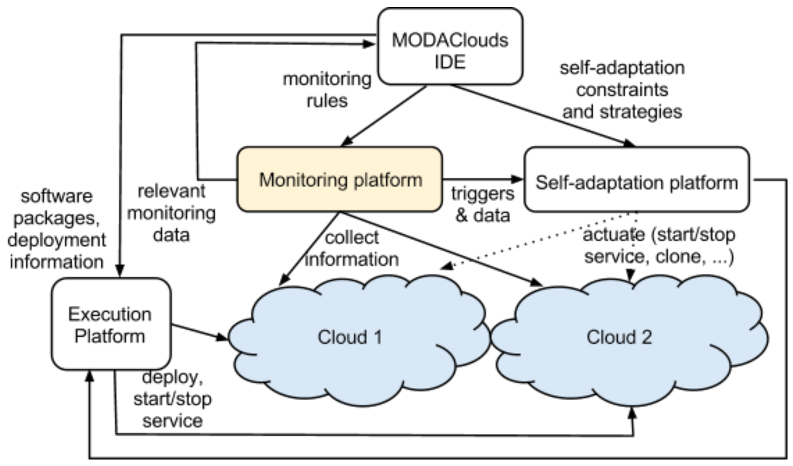


Figure 12: Conceptual architecture - MODAClouds runtime environment

The main objectives of the present thesis are listed below:

- Defining a meta-model for specifying the constraints of Quality of Service (QoS) at design time.
- Defining a meta-model and a language for the specification of the monitoring rules at design time.
- Identifying a methodology for the automatic/semi-automatic translation of QoS constraints into monitoring rules for translating monitoring rules into probes and monitoring queries that will be executed by the analysis component.

Monitoring rules are derived semi-automatically from specifications of QoS constraints. It should be noted that, due to the fact that the monitoring activities introduce overhead in the running application and incur also in costs, they should only be enabled while needed. Accordingly, required mechanisms in order to enable/disable monitoring rules, either manually or by the other monitoring rules, will be developed.

The analysis component will be implemented by a C-SPARQL engine which requires knowledge base to run the queries; where, the knowledge base, as shown in Figure 9, contains ontology and permanent RDF data. The C-SPARQL engine utilizes knowledge base information defined during design-time or run-time in order to analyze the RDF streams that are received from different Data Collectors. The procedure of defining the knowledge base information and the approach through which the C-SPARQL engine employs them, will be explained in details in Section 4.6.

In the present work, MODAClouds ontology, as shown in Figure 10, is employed as the core ontology and will be extended for each application based on the related design time specification. In the present thesis, MiC application which is a social networking web application and has been deployed on Google App Engine (GAE), Azure, and Amazon EC2, is considered as the case study and is explained more in Section 4.3 in details.

4.2 MODACLOUDS MONITORING ARCHITECTURE

The monitoring system architecture is illustrated in Figure 9 and its components are described in the following subsections [MODAClouds deliverable D6.2].

MODAClouds IDE and Monitoring Rules

Monitoring rules are defined at design time either manually by the Cloud application developer or administrator or automatically by the *MODACloudsIDE* that derives them from the definition of the non-functional requirements (for instance constraints on response time, availability, etc.) specified for the cloud-based application.
In the *Monitoring Platform* active monitoring rules continuously process the input monitoring data and evaluate the corresponding conditions. Each monitoring rule defines the monitoring resource and related metrics that will be monitored. Monitoring rules may be active or inactive. The activation of non-active rules can be triggered by the violation of some active rule, according to the activation dependencies specified by the developer at design time. Their activation, cause the other data collectors to be activated and start gathering data from the resource.

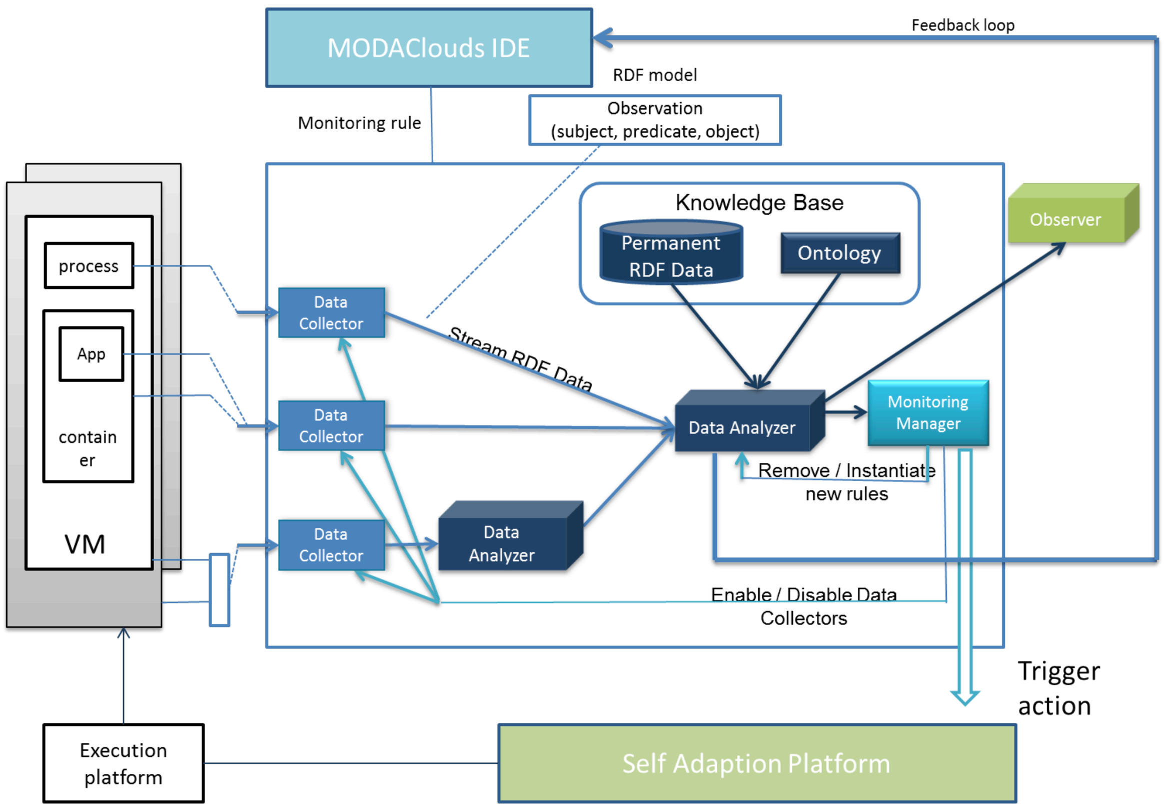


Figure 13: Monitoring Architecture

Data Collectors

Data Collectors (DC) have two main objectives:

- Gathering the monitoring data produced by the various data sources;
- Associating semantic information to the monitored data. This is possible thanks to the structured nature of the data format that will be used to transport the data (RDF).

Data collectors will deal with the heterogeneity of different data sources by producing data under the form of an RDF triple. For some data sources, the data collector may rely on other solutions providing a unified interface over some cloud aspects. For example, libraries such as jclouds are able to interface with multiple cloud providers using the same API. Cloud-proprietary APIs and services (e.g., Amazon CloudWatch API) may be directly invoked as an alternative. Within the MODAClouds project the decision of the best approach will be taken experimentally, by direct comparison of alternatives to understand the best trade-off between overhead and implementation complexity. However, in either case the implementation details will be hidden by the fact that each data collector will communicate with the rest of the system only via RDF. In this thesis we developed data collector for java applications through aspects. Furthermore we developed specific data collector for Amazon EC2 and MS Azure by invoking directly the cloud providers API.

To be as general as possible, the data monitored by the data sources will be retrieved following two strategies: *push* and *pull*. In the push strategy, data sources actively send (i.e., push) data to the data collectors, which passively wait for them. In order to perform a push data collection, data sources must know a data collector endpoint. In the pull strategy, data collectors periodically query (i.e., pull) the various data sources and retrieve data from them. While the push-based approach integrates more naturally with the stream-based architecture of the *Monitoring Platform*, it cannot be adopted in all cases as not all data sources support this strategy (consider for instance the case of a closed-source component offering a SNMP interface). In these cases, a pull-based approach will be adopted. Within MODAClouds, the actual used strategy will then be identified case by case and the needed logic will be embedded in the data collectors. The other parts of the *Monitoring Platform* will see the data streamed out by the data collector in the RDF format and will not be aware of how these data have been retrieved. In this thesis, RDF pushes while EC2 and Azure pull.

Data Analyzer

The *Monitoring Platform* will be populated by a set of data analyzers (DAs). A DA is a basic unit to process monitoring data. At a high-level of abstraction, a DA is seen as accepting in input a continuous stream of monitored data (in RDF format) and producing in output *one or more* streams of data (in RDF format). At a finer granularity, the main duties of a data analyzer include the generation of one or more output streams providing a higher level of abstraction with respect to the given input. For example data at different time granularities may be aggregated to produce a homogeneous view over the data at a longer time scale. Similarly, a DA may perform indirect estimation of unobservable or missing data and output this on new streams.

Since DAs will be heterogeneous and serve different purposes, we distinguish DAs into four categories:

- *Detection* DAs will be responsible to detect a violation in a QoS constraint (e.g. SLA) of a measured metric and raise a trigger to the Self-Adaptation Platform.
- *Correlation DAs* will establish a relationship between measurements collected on different components of the application, with the aim of generating measures that summarize component runtime execution correlations.
- *Estimation* DAs will estimate QoS metrics of the system within a time horizon specified by the MODAClouds IDE that are not directly observable by the data collectors, or that cannot be observed due to overhead concerns.
- *Forecasting DAs* will forecast, using statistical and stochastic methods, trends for some of the metrics needed by the *Self-Adaptation Platform* to manage the application QoS.

Within the above conceptual categorization, it is useful to remember that a single software component may serve in multiple roles. For example, detection and correlation may be done by the DAs. In this thesis we focus on detection DAs that will be implemented by using C-SPARQL, that is a query engine for RDF data. C-SPARQL provides an efficient, yet expressive tool to execute queries on data streams based on the system ontology. C-SPARQL queries correspond to our monitoring rules. As we explained before in Chapter 3, they work on time window, i.e., the most recent triples of a given stream, observed while the data is continuously flowing. The support for RDF format allows for a great deal of flexibility and interoperability.

The task of the data analyzers will be supported by the presence of a monitoring ontology that provides information about the various relationships among the gathered data. Ontology formally represents the knowledge of a specific domain. Ontologies define a set of common concepts and the relationships among them. From this knowledge it is then possible to apply automated reasoning techniques allowing for the extraction of further knowledge. In the context of the MODAClouds monitoring architecture, ontologies are a semantic reference to the components of the run-time environment supporting monitoring rules and their execution and are used by C-SPARQL engine. In particular, monitoring rules writers rely on the formal model represented by the ontology to take advantage of existing concepts and their relationships.

Knowledge base

The task of the data analyzers will be supported by the presence of the Knowledge Base that provides information about the running system. The knowledge base is composed of an *ontology* and *RDF permanent data*. The Ontology formally represents the knowledge of a specific domain. Ontologies define a set of common concepts and the relationships among them. From this knowledge it is then possible to apply automated reasoning techniques allowing for the extraction of further knowledge. In the context of the MODAClouds monitoring architecture, ontologies are a semantic reference to the components of the run-time environment supporting monitoring rules and their execution and are used by C-SPARQL engine. In particular, monitoring rules writers rely on the formal model represented by the ontology to take advantage of existing concepts and their relationships.

RDF permanent data is a database containing the actual information about the running system, and it is in practice one of the infinite possible instantiation of the ontology. The knowledge base requires to be initialized at design time and to be kept updated during the runtime so that DAs may rely on the exact representation of the system.

Observer

An Observer represents any software component that needs to receive information from the Monitoring Platform. Monitoring Platform maintains a list of observers and updates them according to their interests.

Feedback Loop

A monitoring feedback loop will carry information about the application execution to the *MODACloudsIDE* for further analysis and display to the *QoS engineer* or the *Cloud App* developer.

4.3 MODACLOUDS CORE ONTOLOGY

Figure 10 shows the main concepts that are relevant for the Monitoring platform and that constitute the core ontology.

The purpose of the monitoring activity is defined by a set of Monitoring Rules that checks a set of QoS Metrics parsing Monitoring Data. A Monitoring Datum is produced by one of the several Data Collectors attached to the running system. Each Data Collector is in charge of monitoring a Resource. Monitoring Rules may trigger other Monitoring Rules, for example violation of some QoS constraints at the front end of an application may enable monitoring on a back end component. Furthermore, monitoring rules may trigger data collectors that are currently disabled in order to have as an instance finer grain monitoring data. Violation is a Time Stamped Data Item that is generated by the producer which could either be the Data Analyzer or Data Collector, the mentioned data item can also be the one consumed by Consumer. Data Analyzer, which evaluates the Monitoring Rule, is considered as a Producer while writing on stream and as a Consumer while reading it.

During the design time, the core ontology is extended to represent the relevant components of the application which is being monitored.

As the ontology shows, a component can contain another component. Allocation diagram of Palladio shows that each resource contains an application. In our case, each of the components is allocated in a separate resource. This relation could be defined after all the components are specified during the run-time.

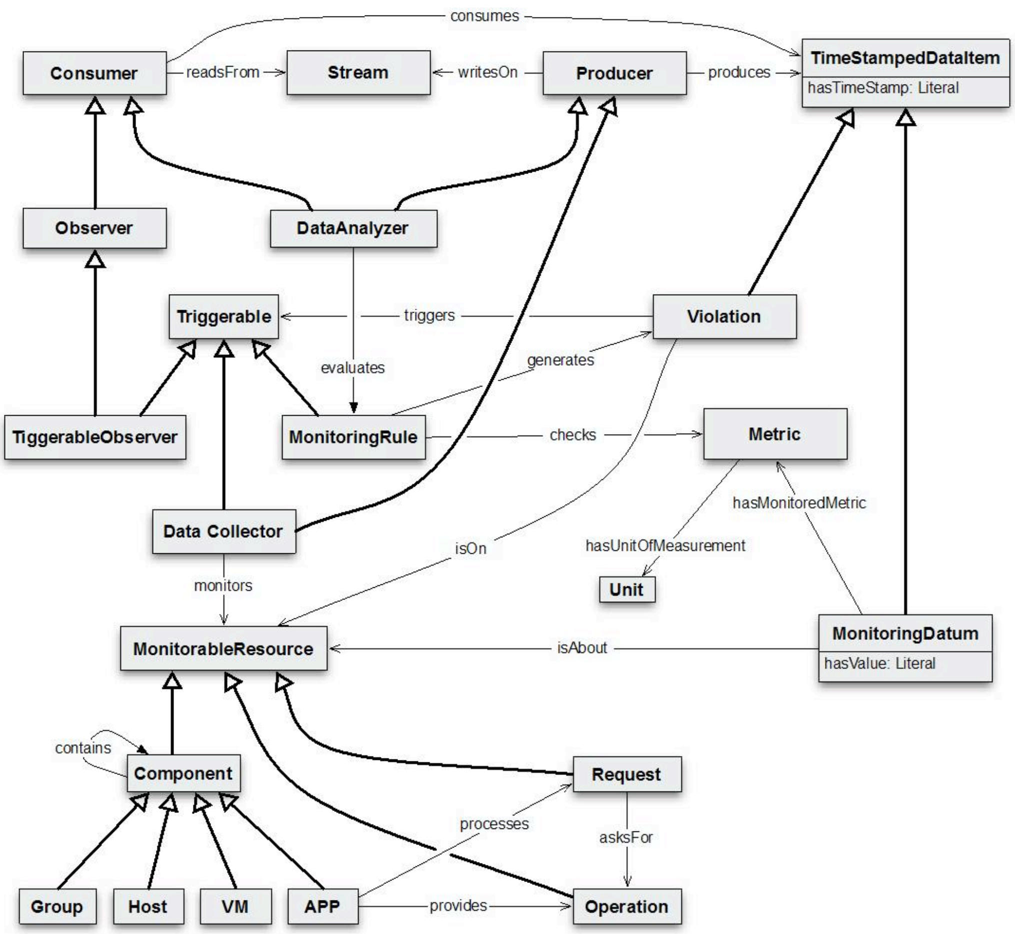


Figure 14: MODAClouds Ontology

4.4 QoS Constraints Definition

In this Section first we describe the definition of QoS, and Palladio meta-model extension for the specification of the Quality of Service (QoS) constraints, and some QoS constraints examples on the Palladio models of MiC application will be next provided.

QoS plays a pivotal role in the optimal delivery of web services, and as more applications are deployed on public clouds, the task of handling QoS becomes harder. As more applications share the same infrastructure, their demand for resources may create contention that reduces the QoS perceived by the user.

An alternative and disputable definition of QoS is requirements on a metric, a tangible quantity measured from a system, which reflects or predicts the subjectively experienced quality. As it was discusses in Section , the metrics that will be monitored are related to the monitorable resources belonging to different abstraction level.

Each QoS constraint is describe by five definitions:

Name: For each QoS a name will be defined.
Target QoS metric: A metric that is related to the target resource and its value is checked by a predicate.
Target QoS Resource: The resource that QoS will be applied on it.

- Predicate (can include standard stats function, e.g., mean, standard deviation, etc.): Checks if the aggregating function specified in the computation field satisfies a given condition.
- Hard/Soft constraint (and their associated priority): Shows whether it is always applicable or not

The QoS constraint applies on a target metric of the resource to check a predicate.

The procedure of specifying QoS constraints on Palladio models of MiC application as a case study is explained in the following part.

In particular, the predicates of QoS constraints at infrastructure and container level metrics will be associated to Palladio Resource environment model, while predicates on constraints at application level will be introduced in the system model.

As shown in the environment Palladio model, the constraints could be applied on all the resources of the MiC application. Figure 7 shows a constraint on the CPU utilization of the Web Server in which the predicate, as a hard constraint, states that the mean value of the CPU Utilization should be lower than 60%.

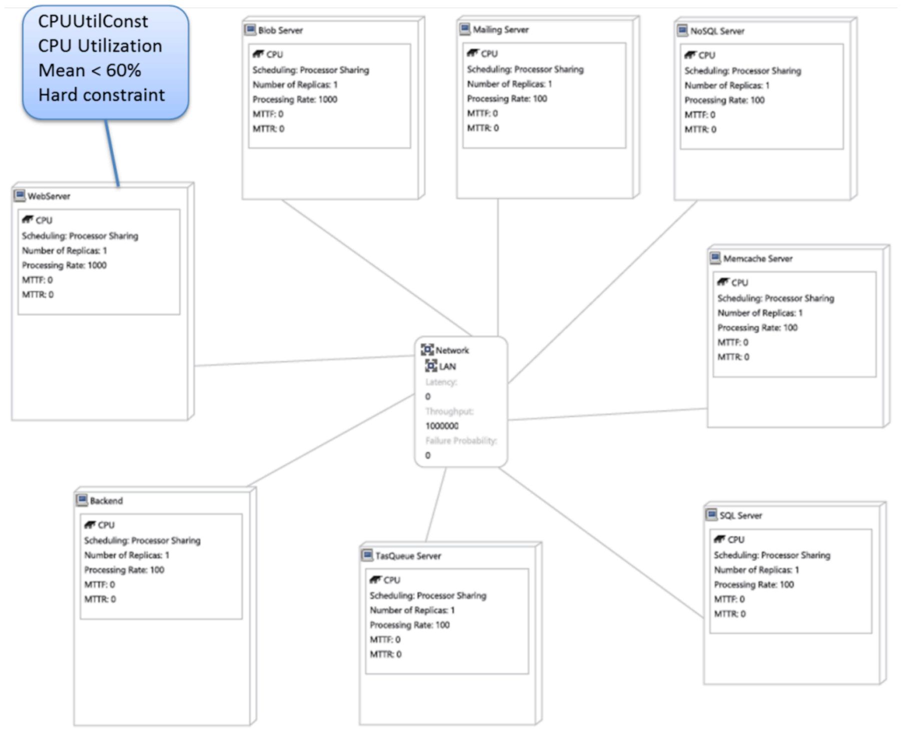


Figure 15: Definition of QoS constraints in the MiC Resource Environment model

Response time constraint, as another instance shown in Figure 16, could be applied on the components of system diagram and its corresponding method inside of the system.

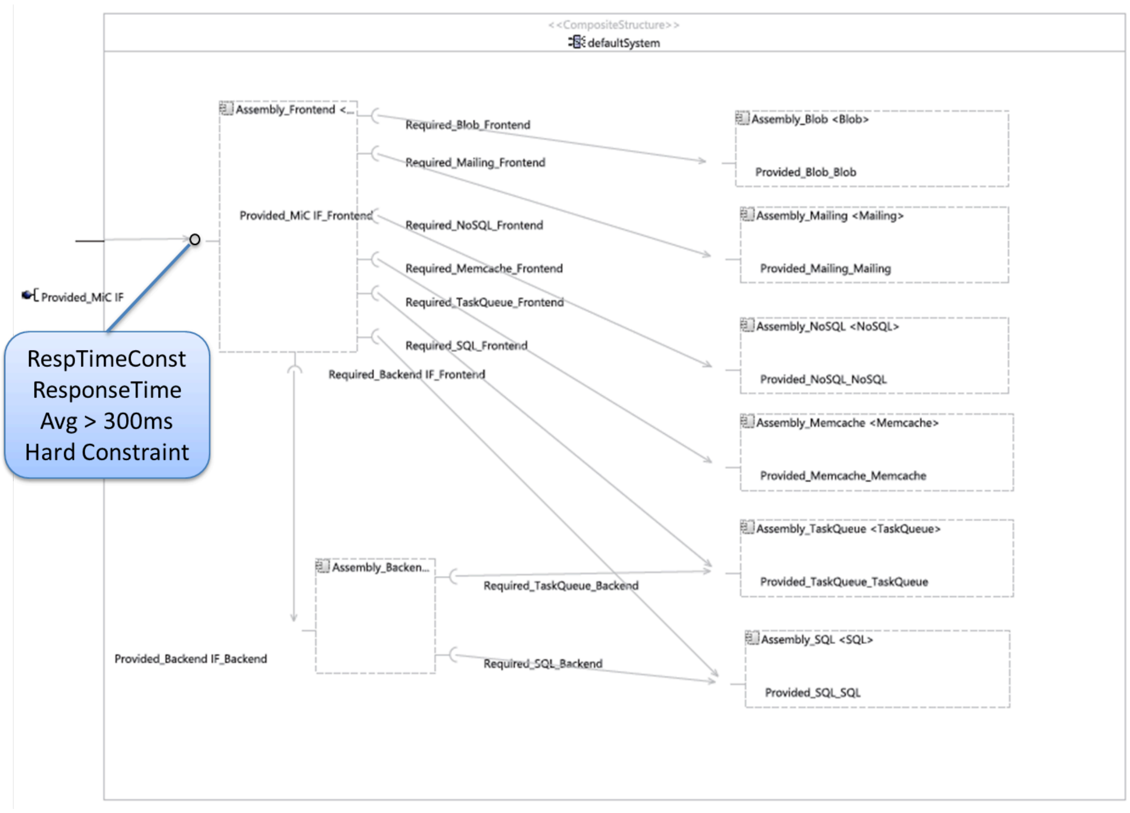


Figure 16: Definition of QoS constraints in the MiC System model

The QoS constraints and particularly the soft constraints with their priority will be used within MODAClouds for design time exploration, though this issue beyond the scope of the present thesis. In the following section, a class diagram for relation among QoS constraints and monitoring rules is provided. Afterwards, the monitoring rules together with their different types are described.

4.5 MONITORING RULES DEFINITION

The present section first discusses the relation between QoS constraints and monitoring rules. Afterwards, the monitoring rules and their definitions are described and some related examples are next provided. The section also elucidates how the monitoring rule is derived from the QoS and the way they can be specified by an application developer without introducing explicit constraints.

The following class diagram shows the relation between QoS constraints and the monitoring rules. As we discussed before, for each QoS constraints there is one monitoring rule, while a monitoring rule may be related to a QoS or not.
Each monitoring rule could be the extension of another rule, or it may be composed by other rules.

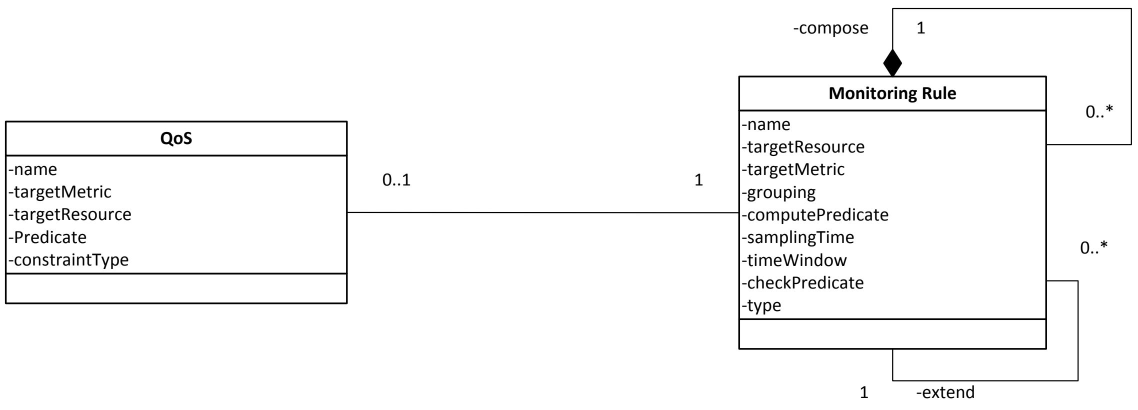


Figure 27: Class Diagram

The purpose of the monitoring activity is defined by a set of Monitoring Rules that checks a set of Metrics observed by Monitoring Resources in the form of Monitoring Datum.

Each monitoring rule contains the following definitions:

Name: For each monitoring rule a name will be defined which could be the same as QoS constraint name.

Target resource: The resource that will be monitored

Target QoS metric: A metric that is related to the target resource and its value is checked by a predicate.

Compute: The aggregation function applied on the values of the target metric.

Grouping: Monitoring datum can be taken at a specific resource (e.g., a component or a VM) or over a set of resources belonging to a group (e.g., the set of VMs hosting the presentation tier of an application).

Sampling Time: Specifying the interval for receiving data from the resource, that could be specified or not.

Sampling probability: A probability value that defines whether the collected data will be sent to the data analyzer or not.

Time window: A window extracts the last data elements that are received, which are the only part of the RDF stream to be considered during the execution of the C-SPARQL query.

Time Unit: shows the unit time of window**.**

Check Predicate: checks if the aggregating function specified in the computation field satisfies a given condition.

Storage type: If the rule is permanent, the output of the query will be logged.

```
Name → [‘Name’]
Target resource → [‘resource name’]
Target QoS metric → [‘metric name’]
Grouping → Monitorable Resource
Compute → ‘mean’ | ‘stddev’ | ‘percentile’ | ‘total
Sampling time → Number TimeUnit
Sampling probability → Number in [0,1]
Time window → Number TimeUnit
TimeUnit → ‘ms’ | ‘s’
Check predicate → [‘predicate expression’]
Storage type    → ‘stream’ | ‘permanent’
```

Considering the fact that a very fine-grained monitoring cause overhead, different sampling probability values are considered for the data collectors and rule definitions. There are three different types of monitoring rules:

- Simple Rules
- Composed Rules
- Extension Rules

A simple rule is directly derived from a QoS constraint. For all of the QoS constraints, a monitoring rule will be derived, just adding some default values for the additional fields that characterize the monitoring rule with respect to the QoS constraint.

In the following example, Figure 17, we are going to drive rules from CPU Utilization QoS.

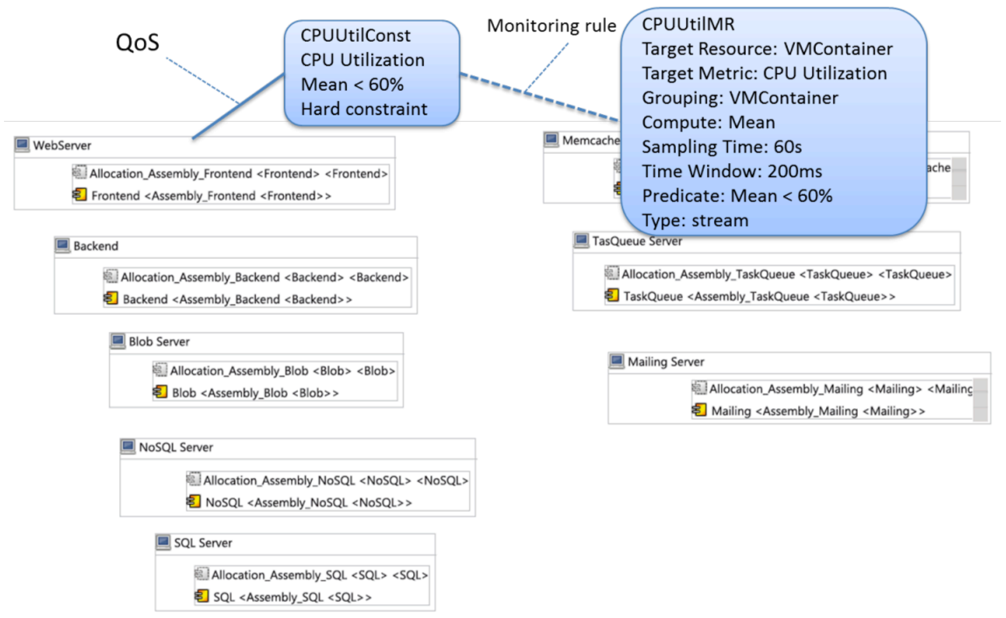

QoS
CPUUtilConst
CPU Utilization
Mean < 60%
Hard constraint
Monitoring rule
CPUUtilMR
Target Resource: VMContainer
Target Metric: CPU Utilization
Grouping: VMContainer
Compute: Mean
Sampling Time: 60s
Time Window: 200ms
Predicate: Mean < 60%
Type: stream
WebServer
Allocation_Assembly_Frontend <Frontend> <Frontend>
Frontend <Assembly_Frontend <Frontend>>
Backend
Allocation_Assembly_Backend <Backend> <Backend>
Backend <Assembly_Backend <Backend>>
Blob Server
Allocation_Assembly_Blob <Blob> <Blob>
Blob <Assembly_Blob <Blob>>
NoSQL Server
Allocation_Assembly_NoSQL <NoSQL> <NoSQL>
NoSQL <Assembly_NoSQL <NoSQL>>
SQL Server
Allocation_Assembly_SQL <SQL> <SQL>
SQL <Assembly_SQL <SQL>>
Memcache
TasQueue Server
Allocation_Assembly_TaskQueue <TaskQueue> <TaskQueue>
TaskQueue <Assembly_TaskQueue <TaskQueue>>
Mailing Server
Allocation_Assembly_Mailing <Mailing> <Mailing
Mailing <Assembly_Mailing <Mailing>>

Figure 17: CPUUtilization Monitoring Rule

In order to allow the definition of complex monitoring rules and foster the re-use of monitoring rules specification, we add some Object Oriented features to the MODAClouds monitoring framework.

In a *composed rule*, the check predicate also includes the predicates of the other monitoring rules and introduces an additional predicate for its target metric. Hence, in the composed rule both pre-existing rules and the additional monitoring rules will be evaluated. In addition, in the composed rule, if the time windows of the rules are not the same, the lower one will be considered

As an instance, the following rule checks the average response time of login operation:

CheckLoginRT

Target Resource: Login
Target Metric: ResponseTime
Grouping: Request
Compute: Mean
Sampling Time: 60s
Time Window: 600ms
Predicate: Mean < 300ms
Type: stream

A new composed rule called *SystemHealthy* checks the health of system via checking both the error code of the requests that ask for Frontend operations during the defined time window and the predicate of *CheckLoginRT*:

SystemHealthy

Target Resource: Frontend

Target Metric: overalError

Grouping: Request

Compute:

Sampling Time: -

Time Window: 600ms

Predicate: ErrorCode = OK && CheckLoginRT.predicate = true

Type: stream

In the new rule, *CheckLoginRT* rule beside the predicate of the new rule will be checked.

A rule could be the extension of a current rule. In this case, it can have its own definitions value or use the values of the super rule. If the values of super rule change, it will effect on the value of the extended rule. As an example, a new rule *CheckNoSQLRT* extends the *CheckLoginRT* rule as follow:

CheckNoSQLRT

Extends: CheckLoginRT

Target Resource: NoSQL
Compute: Mean, Percentile
Predicate: super.Predicate && percentile(95) < 500ms
Type: persistent

The main difference among the composed and extension rules is that, in the composed case more than one rule is checked, while in the extension rule only the predicate of the new rule is checked independently and only the values that differ from the super rule need to be specified.

Within the MODAClouds framework, the rules associated to QoS hard constraints will be always running. Vice versa, some rules may depend on the result of the other rules and may be activated only in case some events in the system are detected. Furthermore, since monitoring rules add overhead and cost in the system, they should be active when needed.

Rules dependencies can be specified through a dependency directed graph:

- Vertex : Rule
- Arc: connects source and target rules, the target rule is enable in the case of violation of the source rule predicate
- Arc can be annotated with activation/deactivation time, and an action that may notify an element of the MODAClouds run-time environment or change a property of the target rule

In the case of violation in check predicates of a rule, after a specified delay time, another rule could be triggered. Similarly, when the check predicate becomes true again, after a specified delay time, the rule will be deactivated.

The following rule shows as an example, if the predicate of *CheckLoginRT* be violated during the activation time, the *CheckExtServerRT* rule will be activated:

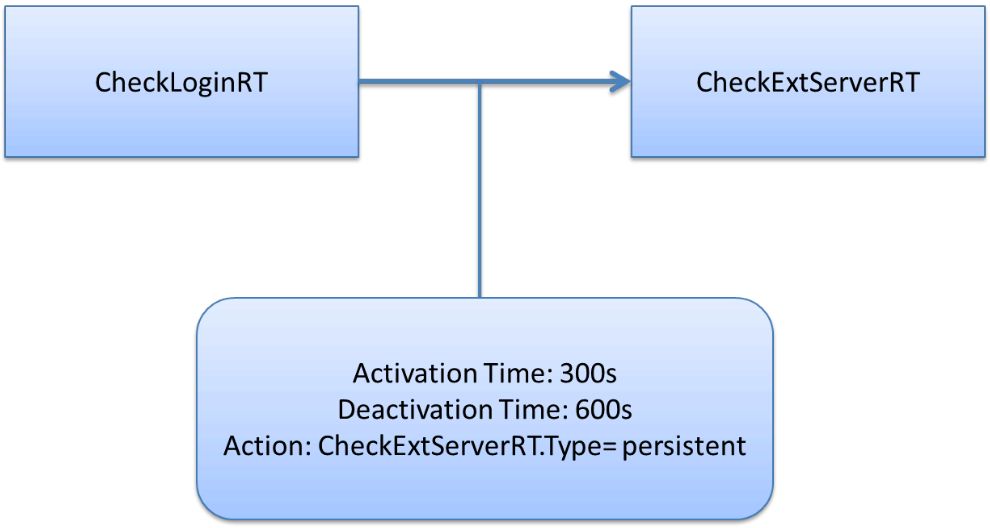


Figure 18: Activation/Deactivation of rule

In the following sections, we will show how the C-SPARQL queries will be generated from the current rules.

4.6 KNOWLEDGE BASE INFORMATION

As described before, in the present thesis we focus on detection DAs that is implemented by using C-SPARQL engine which based on the Knowledge Base executes queries on data streams collected by different DCs. The knowledge Base is composed of an *ontology* and *RDF permanent data*.

In Section 4.7.1 and 4.7.2 we will describe how the permanent knowledge base data are detected for an application during the design-time, and run-time. We also some provide some examples on MiC application.

4.6.1 Design-Time Information

For each application, design-time information is the one that could be retrieved from its specification and should be adaptable to the core ontology, and then the extracted information will be added into the permanent RDF data in order to be used in the process of mapping monitoring rules into the C-SPARQL queries.

One of the specified information that could be augmented into the permanent data is Metric that is defined by two components and specified during the design time. All the metrics that could be defined as the target metrics and will be checked by monitoring rules need to be specified clearly. These metrics are about monitoring resources either components or operations.

Parts of the relations among the concepts of the ontology are fixed in design time, and could be added into the permanent RDF Data.

By investigating the Palladio meta-models of the application that we explained before, all the methods of the components in repository meta-model can be considered as Operation. As the ontology shows, each operation is a Monitorable Resource and is monitored via Data Collector.

Some of the monitoring rules could be defined in design time for the application. Monitoring rules are evaluated by Data Analyzer during the run-time and check the Metrics of the monitoring resources.

In the following, we describe the design time information of our case study, and show how they are recorded in to the Permanent RDF Data.

As we discussed before in Chapter 3 for introducing data streams in C-SPARQL, it is needs to identify data sources. For this purpose IRI is used. Each data stream is associated with a distinct IRI that is a locator of the data source of the stream.

IRI of a data source composed of the prefix of the stream that has been registered in the C-PARQL engine in addition to the data source definition. All the sources that are added in to the permanent RDF Data during either design time or run time will have distinct IRI. We will explain them more in the following sections.

In this work, we register this stream http://micapp.cloudapp.net/mic/ for MiC application, and we consider "mc" as its prefix

In our case study we define the follow metrics, and record them into the knowledge base:

- Metric: *mc:Metric#CPUUtilization, mc:Metric#DiskWriteBytes, mc:Metric#DiskReadBytes, mc:Metric#DiskWriteOps, mc:Metric#DiskReadOps, mc:Metric#NetworkInBytes, mc:Metric#DiskOutBytes, mc:Metric#ResponseTime, mc:Metric#ErrorCode*

  To record each of the metric into the data base, a complete RDF needs to be considered, as an example:

  mc:Metric#CPUUtilization mc:type mc:Metric

From our reference example repository diagram, we consider each method as an operation; for example the operations that are defined in MiC application for the Frontend component are:

- Operation: *mc:Operation#saveUserdata, mc:Operation#saveUserPicture, mc:Operation#saveSelectedTopic, mc:Operation#saveAnswersAbout, mc:Operation#editUserProfile, mc:Operation#updateSimilarity, mc:Operation#refresh, mc:Operation#writeMessage, mc:Operation#deleteMessage, mc:Operation#login, mc:Operation#logout*

And they are saved in the data base in RDF format, like:

mc:Operation#login ***rdf:type*** *mc:Operation*

In this thesis, we consider a data collector for each the operation, and as it is a triggerable, it could be disabled or enabled by monitoring rules. Here we show some of the data collectors for our operations:

- DataCollectors: *mc:DataCollector#saveUserdata, mc:DataCollector#saveUserPicture, mc:DataCollector#saveSelectedTopic, mc:DataCollector#saveAnswersAbout, mc:DataCollector#editUserProfile, mc:DataCollector#updateSimilarity, mc:DataCollector#refresh, mc:DataCollector#writeMessage, mc:DataCollector#deleteMessage, mc:DataCollector#login, and mc:DataCollector#logout*

And all of them are recorded as RDF streams. For example, a data collector for the login is recorded as a type of a Data Collector:

mc:DataCollector#login ***rdf:type*** *mc:DataCollector*

4.6.2 Run-Time Information

After the deployment of the application, related run-time information that are adaptable to the core ontology, could be specified. Run-time information are either permanent or stream. The permanent information is the one that is fixed during the deployment, while the streaming information is not fixed and could not be recorded in the knowledge base.
In this section, we will first explain the permanent run-time information, and then in the rest we will discuss about the stream one.
For specifying the permanent run-time information, we follow the ontology and investigate the Palladio models of the application.
All the components defined by the Palladio repository model are considered as applications monitoring resources while being ran; while multiple components with the same task can be ran simultaneously. It is noteworthy that each of the mentioned running components is unified by a unique ID and will be added to the data base in RDF format.

Palladio resource environment specifies the resources of an application, each of which is considered as a virtual machine that is a monitorable resource component. Whilst, any resource should to be identified in order to be recorded as permanent data.
As the repository model of the MiC is shown, eight applications are considered:

- Applications: mc:Frontend, mc: Backend, mc:Blob, mc:NoSQL, mc:SQL, mc:Memcache, mc:TaskQueue, mc:Mailing

For each running application, a unique id is defined. For example a running Frontend can be recorded as the following RDF:

mc: Frontend#Frontend1 ***rdf:type*** mc:Frontend

It shows that Frontend1 is a running application, and its type is Frontend application.

For the MiC application, we have defined eight resources, as it depicted in the resource environment of the Palladio model. For each running resource, like the other permanent data that we have explained till now, it is needed to specify a unique id and then record it into the knowledge base. The eight VMs that are specified for MiC application are:

- VMs: mc:*WebServer*, mc:*Backend*, mc:*TaskQueue*, mc:*SQLServer*, mc:*MemcacheServer*, mc:*NoSQLServer*, mc:*MailingServer*, mc:*Blob Server*

And related RDF formats for a running webserver could be like:

mc:WebServer1#WebServer1 ***rdf:type*** *mc:WebServer*

This RDF explains that WebServer1 is a running virtual machine from the Webserver virtual machine type.

As the ontology shows, a component can contain another component. Allocation diagram of Palladio shows that each resource contains an application. In our case, each of the components is allocated in a separate resource. This relation could be defined after all the components are specified during the run-time.

For example a running virtual machine can contain a running application:

mc:WebServer1 ***mc:contains*** *mc:Frontend#Frontend1*

mc:Frontend#Frontend1 ***rdf:type*** *mc:Frontend*

mc:WebServer#WebServer1 ***rdf:type*** *mc:WebServer*

Another relation that is defined by the ontology is *provides,* which exists among a running application and different operations. As we have discussed in the previous section, operations are design-time information, and we have specified them before. Palladio allocation diagram of MiC also shows this relation. As an example, a running *Frontend* application, like *Frontend1,* provides the methods inside *MiC IF c*omponent, like *login* method:

mc:Frontend#Frontend1 ***mc:provide*** *mc:Operations#login*

mc:Operations#login ***rdf:type*** *mc:Operation*

mc: Frontend#Frontend1 ***rdf:type*** *mc:Frontend*

In the rest of this section, we describe the stream run-time information. The nature of this information is streaming that are not fix information neither during the design time nor in run-time, so they will not be added into the knowledge base.

From the ontology, *request* and all the related relations are stream information. As the ontology shows, for each request we have the following relations:

- A request *asksFor* an operation.
- A request *processedBy* an application.

For example a *request,* like *req1,* can ask for an operation. This relation among them could be defined in RDF format:

mc:Request#req1 ***rdf:type*** *mc:Request*

mc:Request#req1 ***mc:asksFor*** *mc:Operation#saveUserPicture*

mc:Operation#saveUserPicture ***rdf:type*** *mc:Operation*

From the ontology MonitoringDatum is an *RDFTriple and* is another streaming. It contains a reference to a metric with the relation of *hasMonitoringMetric*, and *isAbout* a *MonitorableResource*.

The relation among Data Collector and Monitorable Resource, *monitors,* is also stream information. Whether a data collector monitors a monitorable resource or not, will change during the run time by monitoring rules. For example in the case of generating a violation by an specific monitoring rule, it may trigger a data collector and enable it to monitor a monitorable resource.

In the following table we summarize the overall information that could be extracted from ontology and show which of them is defined during run time and which during the design time, and they are categorized based on permanent and stream.

Components/Relations	Permanent DesignTime	Permanent RunTime	Stream RunTime
Component		✓	
Operation	✓		
Request			✓
MonitoringDatum			✓
Metric	✓		

MonitoringRule	✓		✓
provides		✓	
contains		✓	
asksFor			✓
isAbout			✓
process			✓
hasValue			✓
hasMonitoredMetric			✓
checks			✓
monitors			✓
produces			✓
isOn			✓
triggers			✓
hasUnitOfMeasurement	✓		

In the next section we show how the monitoring rules are translated in to the C-SPARQL queries.

4.7 TRANSLATING MONITORING RULES TO C-SPARQL QUERIES

In this section we show the steps of translating monitoring rules in to C-SPARQL queries and then we will discuss how they will be generated automatically. Each C-SPARQL queries correspond to one or more monitoring rules.

As it has been discussed in Section 4.5 there are three types of monitoring rules, the first step in the translation is transformation of extended and composed rules into simple rules. Then in the process of translating, we try to map the rule definition into C-SPARQL query definitions.

The query name can be the same as Monitoring rule name, and to definin the Window of the query, time window, and sampling time of the rule are used.

The Sampling probability and storage type definitions are not used during the translation.

For the rest of the translation the Metrics features and their category will be utilized. By following the ontology, each metric has a monitoring datum which has a value and is about a monitorable resource. The monitorable resource can be the target resource of the rule definition, or a resource that is in a relation with target resource, as the ontology relation shows it.

For every target metric the following RDF stream needs to be added in to the query:

?monitoringDatum ***mc:hasValue*** *?value;*
mc:hasMonitoredMetric *mc:Metric#TargetMetric;*
mc:isAbout *?monitorableresource.*

For the rest of the query, the target metric, target resource and grouping definitions are considered, all together. For the metrics two main abstraction levels are defined, Infrastructure and Application. Infrastructure metrics correspond to the VM, Host, and Application monitorable resources, while Application level metrics are related to the Operation, and Request monitorable resources.

- Infrastructure metrics
  - If metric is Infrastructure and target resource is VM, Host, or Application then group by components

    ?monitorableresource mc:type ?VM
    ?VM mc:type mc:VM#targetResource
  - If metric is application level, the target resource is an application and the grouping by operation: System health

    ?monitorableresource mc:type ?request
    ?request mc:asksFor ?operation
    ?operatiomc:providedBy ?app
    ?app mc:type mc:app#targetResource
- Application Metric: If the metric is an application level, the target resource is either Operation, or Request
  - If the metric is application level, and target resource is operation:

    ?monitorableresource mc:type mc:operation#targetResource
    or
    ?monitorableresource mc:type ?request
    ?request mc:asksFor ?operation
    ?operation mc:type mc:operation#targetResource
  - If the metric is application level, and the target resource is request:

    ?monitorableresource mc:type ?request

Here you can see the determined part of the C-SPARQL query for the monitoring rules:

```
REGISTER STREAM/QUERY AS NAME
CONSTRUCT { Stream}
FROM STREAM StreamIRI [“RANGE” Window]
  WHERE{
  ?monitoringDatum mc:hasValue ?value;
                   mc:hasMonitoredMetric mc:Metric#TargetMetric;
                   mc:isAbout ?monitorableresource.
  .
  .
  }
  GROUPBY()
  HAVING(RulePredicate^^xsd:valuetype)
  FILTER(RulePredicate)
```

The GROUPBY, HAVING and FILTER definitions are related to the purpose of each query.

The rest of the query will be extended, based on the monitoring resource target.

For the composition rule, it just needs to translate the composited rule without considering the predicate of base rule, and then the stream result of the other rules will be added in to the translation of composition rule.

For extended rule, first we put the translation parts of the super rule, and then complete the rest of rule by translating the definition of the new rule.

Related example for both composition, and translation rules are provided in the last part of this section

In the following, we explain the process of translating different types of monitoring rules in to the C-SPARQL queries via some examples.

Simple rule Example

As an example, here we show the translation of the CPUUtilization rule:

REGISTER QUERY CPUUtilViolation AS

FROM STREAM <http://mic-app.com/stream> [RANGE 300s STEP 60s]

```
WHERE {
            ?datum  mc:hasValue ?CPU;
                    mc:hasMonitoredMetrics mc:Metric#CPUUtilization ;
                    mc:isAbout ?vmContainer .
            ?vmContainer mc:type mc:VirtualMachine .}
Group By (?vmContainer)
Having (AVG(?CPU) > “0.6”^^xsd:float)
```

Violation rule Example

Consider an existence monitoring rule that checks the response time of all requests that ask for *SaveUserData* operation (mc:MonitoringRule#CheckSaveUsrRT). The new rule will be created in the case that this rule is violated. In the following case we want to trigger the Data Collector to monitor SaveUserPicture Operation.

mc:MonitoringRule#CheckSaveUsrRT ***rdf:type*** *mc:MonitoringRule*

mc: MonitoringRule#CheckSaveUsrRT ***mc:checks*** *mc:Metric#ResponseTime*

mc:Metric#ResponseTime ***rdf:type*** *mc:Metric*

mc: MonitoringRule#CheckSaveUsrRT ***mc:checks*** *mc:Metric#ResponseCode*

mc:Metric#ResponseCode ***rdf:type*** *mc:Metric*

The new rule is:

```
REGISTER STREAM AS CheckSaveUsrRTViolation
CONSTRUCT {[] rdf:type mc:Violation;
              mc:isGeneratedBy mc:MonitoringRule#CheckSaveUsrRT;
              mc:isOn mc:Operation#SaveUserData. }
FROM STREAM <http://mic-app.com/stream> [RANGE 600s STEP 60s]
WHERE {
          ?datum1 mc:hasValue ?resTime;
                  mc:hasMonitoredMetrics mc:Metric#responseTime ;
                    mc:isAbout ?request .
          ?datume2 mc:hasValue “200”;
                   mc:hasMonitoredMetrics mc:Metric#responseCode ;
                   mc:isAbout ?request .
           ?request mc:asksFor mc:Operation#SaveUserData.
Filter (AVG(?resTime) > “300”^^xsd:float).
}
```

Extended rule Example

Here we want to show how an extended rule will be trasnlated from the base rule. Those definitions of the extended rule that has the value of super rule definition, it is just needed to use related query translation, then for the rest, the translation of rule will be done. As the following example shows.

From the example that has been expressed in section 4.5, *CheckNoSQLRT* rule is extended from the *CheckLoginRT* monitoring rule.

First we translate the base rule:

```
REGISTER STREAM AS CheckLoginRTViolation
CONSTRUCT { [] rdf:type mc:Violation;
                   mc:isGeneratedBy mc:MonitoringRule#CheckLoginRT;
                   mc:occursOn ?request
}
FROM STREAM <http://mic-app.com/stream> [RANGE 600s STEP 60s]
WHERE {
            ?datum mc:hasValue ?resTime;
                     mc:hasMonitoredMetrics mc:Metric#responseTime ;
                       mc:isAbout ?request .
            ?request mc:asksFor ?operation.
            ?operation rdf:type mc:Operation#Login . }
    GROUPBY(?request)
    Having(AVG(?resTime)  < “300”^^xsd:float)
```

And the extension rule is translated as follow:

```
REGISTER Query AS CheckNoSQLRT
FROM STREAM <http://mic-app.com/stream> [RANGE 600s]
WHERE {
            ?datum1 mc:hasValue ?respTime;
                    mc:hasMonitoredMetrics mc:Metric#ResponseTime.
                   mc:isAbout ?request .
            ?request mc:asksFor ?operation.
            ?operation rdf:type mc:Operation#Login.
            ?datum2 mc:hasValue ?respTime;
                    mc:hasMonitoredMetrics mc:Metric#ResponseTime ;
```

```
                mc:isAbout ?request .
          ?request mc:asksFor ?operation.
          ?operation rdf:type mc:Operation#NoSQL . }
GROUP BY (?request)
HAVING( AVG(?resTime)  < “300”^^xsd:float && PERC(0.95) < “500”^^xsd:floa )
```

Composition rule Example

For translating a composition rule into the related C-SPARQL query, it is needed to check the rule predicate that is defined in the predicate of the composition rule.

For this purpose, the stream that is generated via the base rule is used in the query of the composition rule.

As an example, the predicate of *SYSTEMHealthy* rule checkes the predicate of the *CheckLoginRTViolation*, that is generated an stream in the case of violation.

```
REGISTER Query AS SystemHealthy
FROM STREAM <http://mic-app.com/stream> [RANGE 600s STEP 60s]
FROM STREAM <http://mic-app.com/stream/CheckLoginRTViolation> [RANGE
600s STEP 60s]
WHERE {
        ?datum ms:hasvalue ?resCode;
               mc:hasMonitoredMetrics mc:Metric#responseCode ;
               mc:isAbout ?request .
        ?request mc:asksFor ?operation.
        ?operation rdf:type mc:Operation#Login .
                 mc:isprovidedBy ?application;
        ?application rdf:type mc:Application#Frontend
   }
   GROUPBY(?request)
   HAVING(?respCode = 200)
```

4.8 DATA COLLECTORS

In the present thesis, Data collectors are designed to deal with the heterogeneity of different data sources by producing data under the form of an RDF triple. As we explained before the data monitored by the data sources will be retrieved by pushing or pulling strategies. While in the push strategy, data sources actively send (i.e., push) data to the data collectors, in the pull strategy, data collectors periodically query (i.e., pull) the various data sources and retrieve data from them. In order to perform a push data collection, data sources must know a data collector endpoint. The push-based approach integrates more naturally with the stream-based architecture of the *Monitoring Platform*. In this thesis, RDF pushes while EC2 and Azure pull. Here we present three different data collectors that are designed in this thesis:

- **Aspects Data Collector**: For each operations in the component of Palladio repository model a point cut is defined. In each join point the execution time related to that operation will be computed and will be sent to the data analyzer. Each point cut, that acts as a data collector, could be activated while a rule defines for that and then it will start feeding the engine some are already activated, some will be deactivated. High level data collector in the case violation spesicfic dc could be activated. There is no sampling time for these rules, as the request for each operation does not have defined time. A point cut could be applied to all methods or a specific one. To create a point cut, our monitoring framework will generate it by receiving the related method name from the user side, and substituing that in the defult pointcut expression.
  - **Data Collector for CloudWatch**: In Amazon EC2, the data are provided periodically. For the metrics that are defined, the data are gathered periodically, and are sent to the data analyzer.
  - **Data Collector for Azure storage**: Windows Azure Diagnostic enables us to collect diagnostic data from an application running in Windows Azure. After the diagnostic data is collected it can be transferred to a Windows Azure storage account for persistance. The data of the metrics that are added in diagnostic, will be stored in to storage table. When the data collector is triggered it starts collecting data from storage table for the related metric. As the Azure storage is NoSQL, related query should applied to extract the data. After the data is collected, it will be sent to the data analyzer.

5 CHAPTER 5

In this chapter we will evaluate the monitoring approach developed in this thesis. The main objectives are quantitatively evaluate the overhead that is introduce by the monitoring platform which are done on MiC application, we described in Section , and evaluate the expressiveness of the proposed monitoring rules from the interview that has been done with a professional system engineer partner of the MODAClouds consortium and provider of a case study for the validation of the project.
In Section 5.1 the basic settings of the system for testing have been specified, and the experiment results of the application on the cloud were discussed. Section 5.2, describes some monitoring rules and experiment that are performed on our monitoring platform. Finally in Section 5.3, we will provide the interview with one of the MODAClouds.

5.1 EXPERIMENT RESULT ON THE CLOUD

5.1.1 Basic Test Settings

We deployed the MiC application in Azure. In particular, the application Frontend has been deployed on a small worker role. For the evaluation of the overhead we added the aspects to every single page. The backend was on a medium worker role. Both worker roles run a Tomcat web container and were deployed in the West Europe Microsoft data center. Adding aspects to every frontend servlet page corresponds to the worst case evaluation scenario, since before and after advice execution requires few millisecond as the generation of a dynamic web page, while the execution of the backend method for the

evaluation of the Pearson coefficient requires several seconds and hence in that case the monitoring overhead would benegligible.
In order to represent a multi-cloud and distribute monitoring environment, C-SPARQL engine was deployed in Amazon EC2 in the Virginia Amazon Region during the tests.
The monitoring rule defined for the tests was a simple rule which checked the response time and status code of all frontend operations for each individual during the time window of 300 samples.
We considered the user session registration as the test plan. Registration scenario contains:

- Save picture in to Blob storage
- Save User Profile in to SQL Server
- Select topics of interest from Data storage
- Answer questions from Data storage, and
- Save all users selection in to the Data storage

JMETER was used as workload injector and three clients were deployed medium host in the same data center providing the worker roles.The Apache JMeter™ desktop application is open source software, 100% pure Java application [Reference] designed to load test functional behavior and measure performance. In particular, master controller initiated the test by two slaves clients.

- XXX Here add Palladio diagrams annotated with your rule. XXX

5.1.2 Experiments and graphs

In order to estimate the overhead, the platform has been investigated under different conditions.

- Terms of comparison were MiC without any monitoring rule
- Different sampling probabilities were considered
- The reported results were executed the week, as the Azure performance were worst during the weekend

The test were performed:

- Overhead estimation for single user scenario
- Sampling probability effect
- Overhead estimation for multi users scenario (representative of real application)

In the first run a single user was tested for 10 hours with 0.1 sampling probabilities. As for a single user we have small number of request, the test has been done for a longer time period with respect to the following analyses. In this run every response time is exactly the demanding time of running single pages with and without aspect actually without any problem of concurrency. In this test we could evaluate the effect of the aspect. The related results are shown in Table 1.

	Number of Samples		Response Time		Overhead
	Native	Aspect	Native	Aspect	
Total	2150	2155	**414.76**	**443.04**	6.82%

Table 3: Overhead

From the table, the overall overhead of the system is around 7%, that is reasonable in production environment.

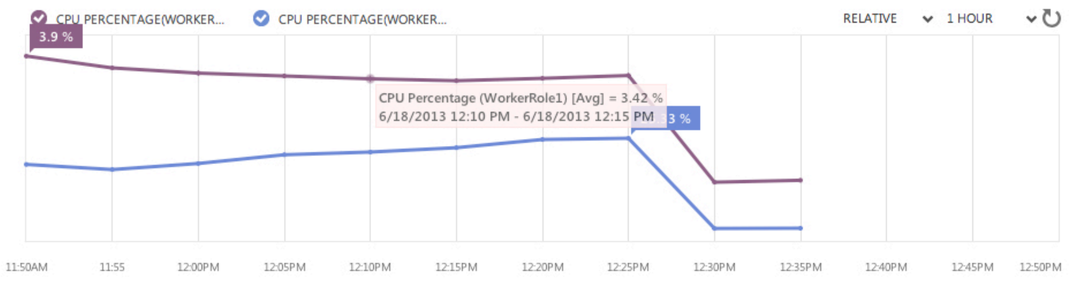
In the second run, we tested the system with small number of users, for ten, twenty and thirt number of users. The results that, shown in Table, present that aspect version of the application in all cases have errors. The investigation displayed that this problem is caused because of feeding the C-SPARQL engine for all the requests. The errors specify the importance of using sampling probability in a production environment. , so the rest of the tests reported in this chapter were done based on the sampling probability.

Number of Users	Error% (Total)		Throughput
	Native	Aspect	
10	0%	11.31%	1.5/sec
20	0%	19.78%	3.0/sec
30	0%	23.92%	1.4/sec

Table 4 : Test result without sampling

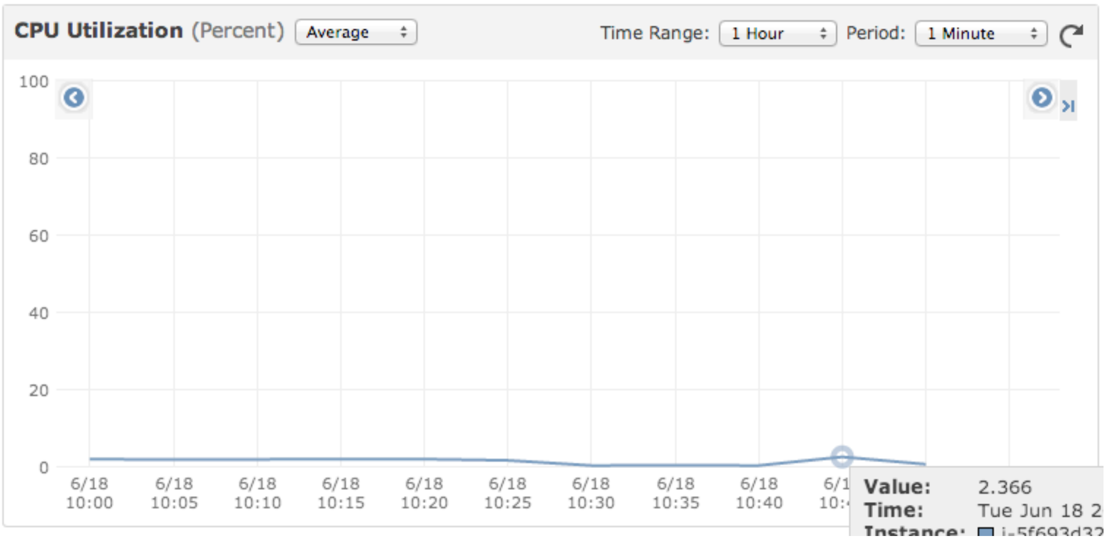
The CPU utilization of the Azure worker roles, and C-SPARQL engine were also registered during this test for aspect version of the application.

Figures 1, and 2 show the results of testing 10 users in the aspect version of the application. Worker role 1 is in the purple, while worker role 2 is the blue line. Figure shows the result of 20 users and Figures X-Y illustrate the results achieved for 30 users:



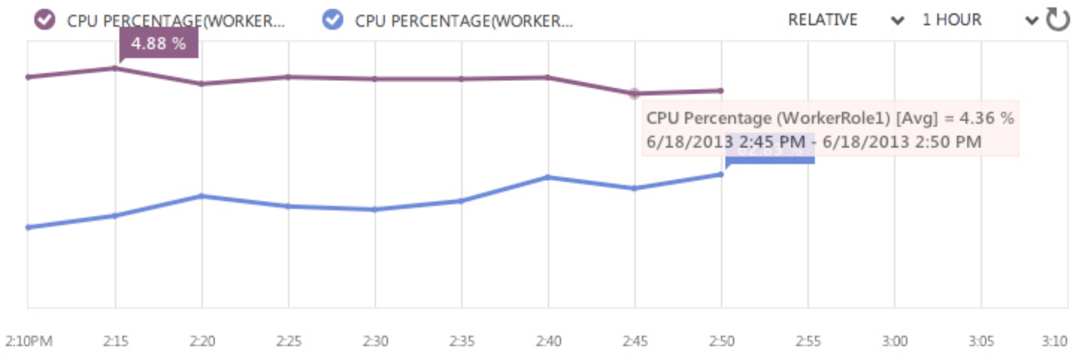
CPU Percentage (WorkerRole1) [Avg] = 3.42 %
6/18/2013 12:10 PM - 6/18/2013 12:15 PM

Figure 19: CPU Utilization for 10 users in Aspect version



Value: 2.366
Time: Tue Jun 18 2
Instance: i-5f693d32

Figure 20: CPU Utilization of C-SPARQL for 10 users in Aspect version



CPU Percentage (WorkerRole1) [Avg] = 4.36 %
6/18/2013 2:45 PM - 6/18/2013 2:50 PM

Figure 21: CPU Utilization for 20 users in Aspect version

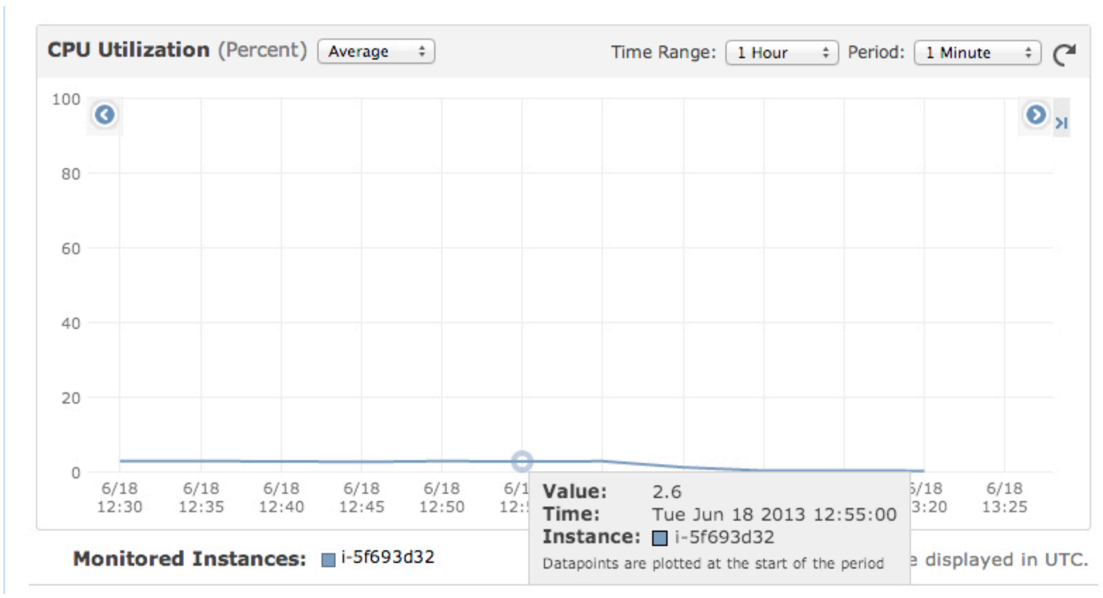
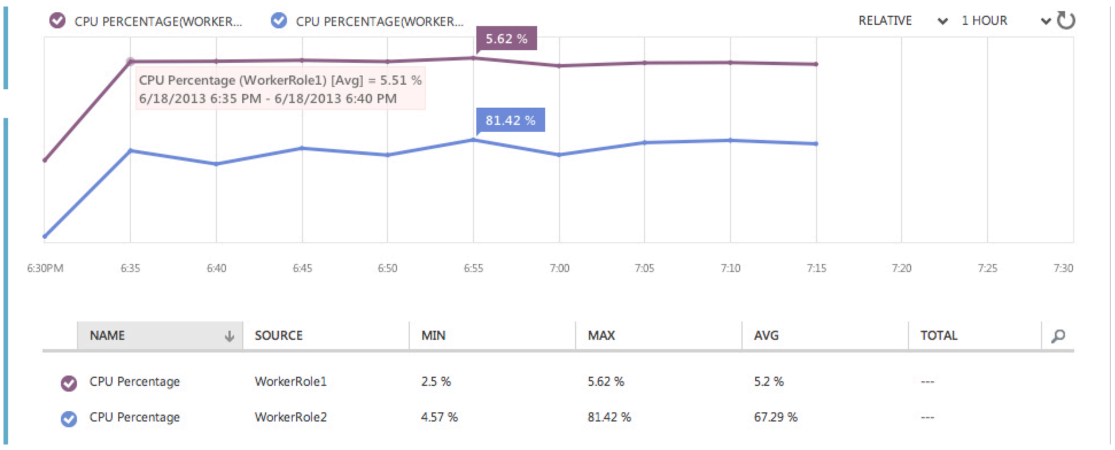
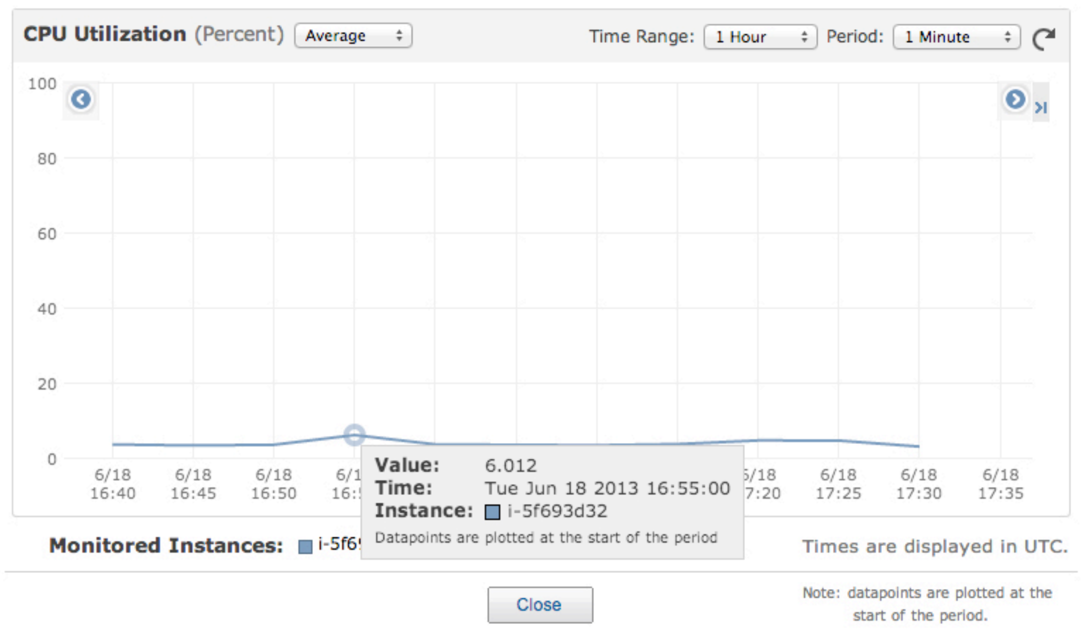


Figure 22: CPU Utilization of C-SPARQL for 20 users in Aspect version



NAME	SOURCE	MIN	MAX	AVG	TOTAL
CPU Percentage	WorkerRole1	2.5 %	5.62 %	5.2 %	---
CPU Percentage	WorkerRole2	4.57 %	81.42 %	67.29 %	---

Figure 23: CPU Utilization for 30 users in Aspect version



CPU Utilization (Percent) Average Time Range: 1 Hour Period: 1 Minute

Value: 6.012
Time: Tue Jun 18 2013 16:55:00
Instance: ▨ i-5f693d32
Datapoints are plotted at the start of the period

Monitored Instances: ▨ i-5f6

Times are displayed in UTC.

Note: datapoints are plotted at the start of the period.

Figure 24: CPU Utilization of C-SPARQL for 30 users in Aspect version

The following Table summarized the CPU utilization of all the VMs in three cases that has been investigated during this test:

Number of Users	Worker Role 1 %	Worker Role 2 %	C-SPARQL %
10	3.42	17.61	2.366
20	4.36	43.28	2.6
30	5.51	72.77	3.534

Table 5: CPU utilization of VMs in Aspect version

As the table shows, CPU utilization is low for front end role, while it is high for backend role, which is clearly the application bottleneck , and for C-SPARQL utilization it is very low.

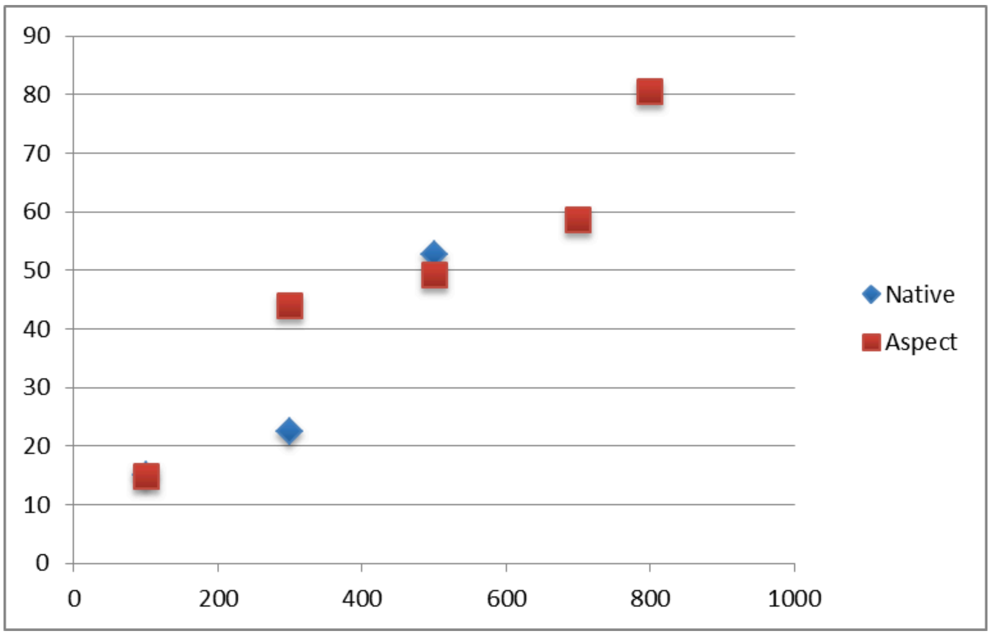
We did more extended evaluation by increasing the number of users until the worker roles became saturated. We tested from 100 users to 800 users. Table is shown the related results.

	X(request/sec)		Web role Utilization%		Response Time (msec)		Error rate%	C-SPARQL utilization	overhead
Number of Users	Native	Aspect	Native	Aspect	Native	Aspect			
100	15	15.1	10.83	9.62	145	95	0.56	1	

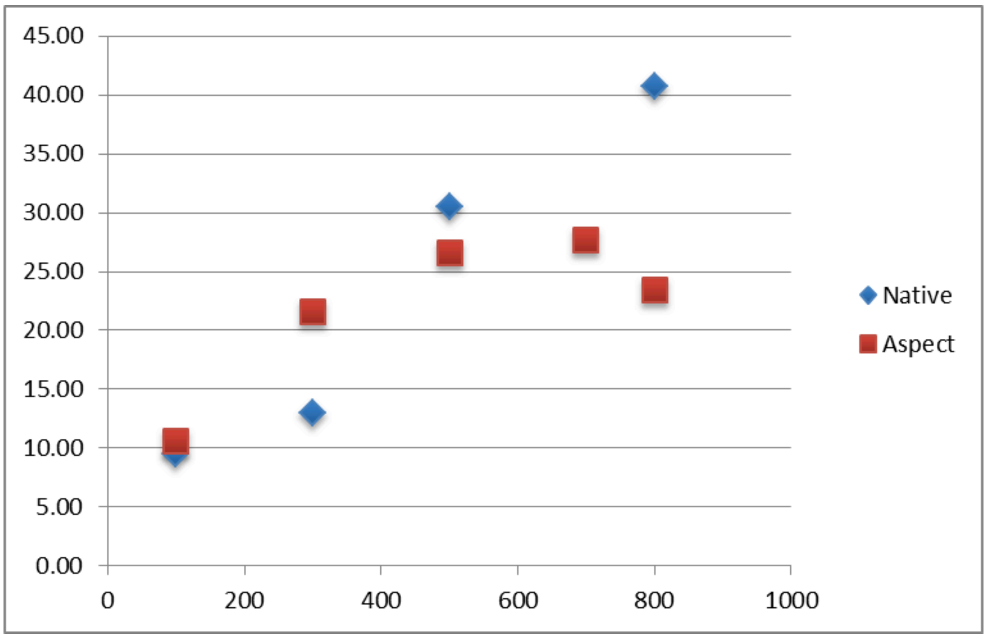
300	45.1	43.8	21.49	18.38	139	99	1.39	1.2	
500	72	55.8	29.58	22.60	153	123	1.72	1.5	
700	88.8	59.8	33.34	23.82	161	164	1.89	1.7	
800	79.7	54.8	30.36	35.42	214	1753	1.95	1.9	

In the following EXPLAIN

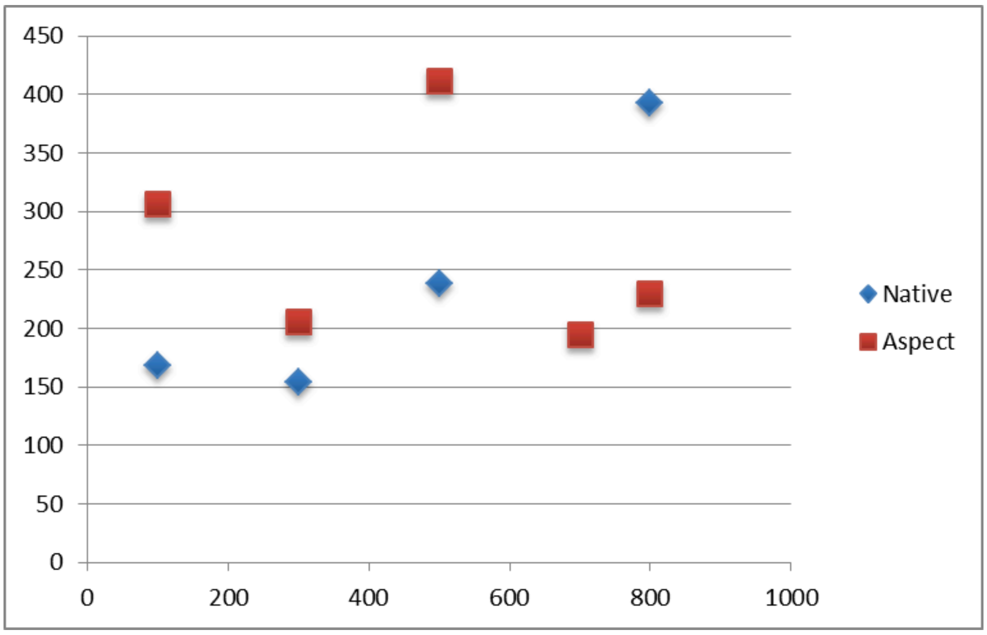
Throughput (X):



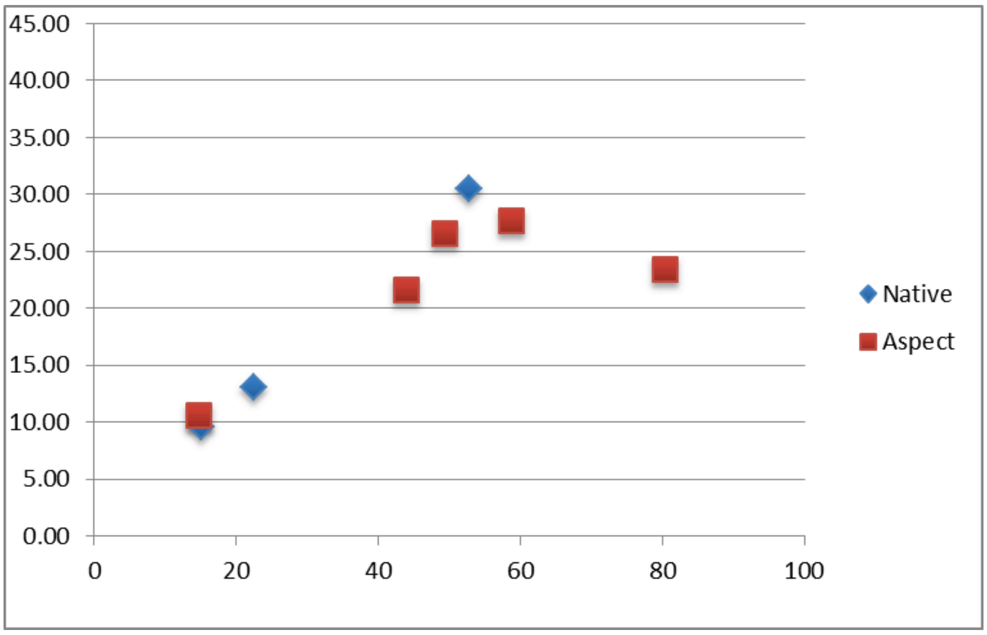
Utility:



Response time:



Utility versus Throughput:



In the 800 users even the role without the aspect was saturated. There is no significant overhead in response time in any of workload condition. The error code reduced, as we used the sampling, but the problem is that starting from 500 users we have some performance degradation in throughput and also increase of standard deviation of the response time. We investigated what was caused the problem. Spike in the utilization of the worker role of the aspect that shows maybe because of **garbage collector**, also the clean version without aspect has problem as soon as the role become saturated, and we have performance degradation, and the throughput is higher than the other one. The lesson learning is that this tool need to be managed carefully.

5.2 EVALUATION OF THE POWER OF MONITORING RULES EXPRESSIVENESS

In order to validate the power of monitoring rules that we have define in this thesis, and explained in the previous chapter, we did an interview with a system engineer from BOC, one of the partner of the MODAClouds project. The interview was performed through Skype and overall last 2 hours and half. In the following a report from the interview is provided in order to show the way our approach can be used in their application which will provide of the MODAClouds project case study.

The main objectives of the interview were:

- Whether the monitoring rule language that has been defined in this present work can be easily understood for who deals with monitoring challenges or not
- The monitoring language allows to define the monitoring rules that are used in real systems
- Identify benefits of the approaches of the monitoring rules presented in this work

Moreover, they mentioned that combination rule is one of the approaches that will be useful in monitoring of their application, as they use data from different data collectors, and put query on them.

Their provided application names ADOxx Application [REFRENCE]. ADOxx is a Meta modeling platform that enables the end user to instantiate products by defining meta-models and having scenarios specific functionality. The products that are derived from this platform are used by end customers either in a rich client scenario with relational data base behind it or alternatively as a web application with a web tier, a business tier, and data base. Web based access is actually the case study considered within MODAClouds project.

The main motivation for BOC to move this application to the Cloud is to move from BOC data center to cloud base infrastructure to gain flexibility especially in terms of critical issue of services locations, especially in Europe, and also for service proximity to guarantee low latency between web browser and web tier. The other goal is to get rid of hardware staff of the data center and to leverage cloud services and benefit from the flexibility of cloud services.
This application follows classical 3-tier architecture:

- Web tier: Java application usually in tomcat web container
- Presentation tier: REST API eccessed from java script base user interface which is running on end customer browser. It talks to the Business tier, and is SOA (SOAP)
- Business tier: is C++, which is targeted in windows and challenge for cloud plans. It provides different logics, provide script interfaces for logic.

The data base currently used is oracle Microsoft SQL server can be used as alternative.
The plan to move to cloud is first one to one migration to IaaS, after that take the benefit of cloud in terms of PaaS features.
Curently Nagios is used in order to monitoring the ADOxx application**.** It is an open source computer system monitor, network monitoring and infrastructure monitoring software application. It is deal with the concept of monitoring server in the case that when all the configurations for monitored infrastructure, services, etc. are ready, you can access monitoring system as either black box then you can query in some case of HTTP server get request from the server, or agent running in the server that allows you reach all kind of scripting that helps you get all the needed information.
In the commands one can define a threshold and how the services be monitored. The rules can be defined in Nagios, for example binding the actions. As an example if the response exceed certain threshold another instance of the application could be launched, and bind it to the load balancer to scale out the application. Nagios is mostly used for handling the problems. Currently there is no possibility to scale the application as it is in a standard data center without cloud features, and it's also didn't needed so far scale in such a high way, what is done some times is extend the environment that need more long time planning.
Nagios does not separate data collectors from data analyzers; they are combined at the same entity. Actually there are scripts that gather data and analyze the data to check whether the status is critical, for warning or and provide some performance data.

The monitoring approach developed in this thesis could be helpful as it collects data from different application resources, and the combination of data collectors approaches with BOC will provide better functionality.
As an example, in our approach aspects are used to monitor web applications that are for application tires and not business tires.
Administrative of Nagios receives data from bunch of plugins available for example CPU or Memory usage. Those are not cloud specific, and whenever one faces with something that no plugin existed for that, it should be designed. In this case especially in the cloud context as we define data collectors in our approach, they can use the beneficial of our feature.
Nagios monitoring could be integrated with C-SPARQL solution. In this case for any violation in the constraints, an alert will be sent to Nagios. As there is no way to push data to the Nagios, an intermediate like observer that is defined in our architecture is needed.
Both Nagios, and C-SPARQL engine can query on the recent data. This definition has been specified as time window in our work, and Nagios can make a profit of that in its queries.
Inside of the data analyzer there are rules that could distinguish, and in the case of happening an special situation the data will be pushed to the observer and from there it will be keep it up by nagious.
Bothe approaches could be used with each other in terms of using the advantage of our approach to collect data, analyze them in C-SPARQL engine, and then provide them as the input to the Nagios. This means that Nagios be used as the next layer after C-SPARQL.
As it was discussed during the interview, the monitoring rule language that has been defined in this thesis is easily understandable for who deals with challenges of monitoring.
Also they mentioned that combination rule is one of the approaches that could be useful in monitoring the ADOxx, as they use data from different data collectors, and put query on them.
Here is a monitoring rule example, that they provided based on our language on their case study application.

6 CONCLUSION