Relative QoS:
A New Concept for Cloud Service Quality

Gustavo Alves¹, Caio Silva¹, Everton Cavalcante¹,², Thais Batista¹, Frederico Lopes¹

¹Federal University of Rio Grande do Norte, Natal, Brazil
²IRISA-UMR CNRS/Université de Bretagne-Sud, Vannes, France

gustavo@ppgsc.ufrn.br, caio.crd@gmail.com, evertonrsc@ppgsc.ufrn.br, thais@ufnet.br, fred@imd.ufrn.br

Abstract—In a Cloud Computing scenario, cloud services are accessed via Internet and typically there is a communication latency between the service provider and its clients. As a consequence, QoS parameters noticed by the clients (such as service response time) have different values depending on their location. In this paper, we propose the relative QoS (RQoS) concept, which defines that the quality of a service is relative to its geographical location. We also introduce a monitoring mechanism that provides RQoS parameters of cloud services in the SaaS (Software as a Service) layer, so that clients can choose the services that better suit their needs. In order to validate this new concept and the monitoring mechanism, we present a proof of concept and some computational experiments to show that there are variations in the values of QoS depending on the geographic location where services are monitored. The performed experiments also aimed to evaluate the performance of the monitoring mechanism in terms of the assessment of RQoS parameters.

Keywords—Cloud Computing; Service Monitoring; SaaS; Relative QoS; RQoS; RQoS Monitor.

I. INTRODUCTION

Cloud Computing can be defined as a paradigm that enables accessing a set of shared and configurable computing resources (e.g., networks, servers, storage facilities, and applications) typically named cloud [1, 2]. Such computing resources are accessed via network in a simple, pervasive way and are offered as services. They can be also quickly provided and released with minimum management effort and interaction with the service provider. Moreover, these services can be provided on demand, so that clients pay only for the use of such services (pay-per-use model).

Cloud Computing providers establish agreements with their clients, the so-called Service-Level Agreements (SLAs). These agreements specify values of availability and other QoS (Quality of Service) parameters regarding the provided services. Given the dynamic nature of services in a cloud environment, it is essential to continuously monitor them, so that clients (users and/or applications) can be notified when the services are no longer available, or in cases of QoS degradation, or even to ensure that the QoS statements presented in the SLA are really being provided [3]. In other words, this monitoring procedure can be used to verify that the cloud provider is not breaking the agreed contract.

As cloud services are accessed via Internet, the physical machines in which these services are executed and the clients that consume them usually are at different geographic locations. As shown in [4], the geographical location of both client and service provider interferes in the latency communication between them. For this reason, QoS parameters noticed by clients (such as service response time) have different values depending on their location. Therefore, the monitoring mechanism should be at the same geographic location or even at the same computational node of the client.

Current work on service monitoring (e.g., cloud services, Web services, ubiquitous services, etc.) deals with QoS metadata as absolute values. Such QoS values can be defined either from a single monitor or from different sources, so that monitored data are aggregated into a single QoS value by using mathematical formulas regardless the location of services, clients, and the monitoring mechanism. However, in the Cloud Computing context, this strategy does not reflect real QoS values since cloud services are typically accessed via Internet and the network interferes in the communication between services and clients.

In order to address these issues, this work proposes the relative QoS (RQoS) concept, which defines that the service quality is relative to the geographical location of the monitored service. In order to show the applicability of this concept, we present RQoS Monitor, a mechanism that monitors and assesses RQoS parameters of cloud services. Such parameters are provided by the mechanism to service clients, which can choose the services that better suit their needs. The RQoS Monitor is divided into two different tools that work together: (i) App Monitor, which runs in the same computational node as the client, and (ii) Service Monitor, which performs the monitoring process itself and is able to identify if a service is really in failure and/or if there is a communication problem between the service and the client. Such a monitoring mechanism currently considers only services in the SaaS (Software as a Service) cloud layer, i.e., services that can be accessed as Web services.

Aiming at validating the RQoS concept and the RQoS Monitor mechanism, we present a proof of concept and some computational experiments to show that there are variations in the values of QoS depending on the geographic location of the monitored services. In addition, the performed experiments have also aimed to evaluate the performance of the monitoring mechanism in terms of measurement of RQoS parameters.
This paper is structured as follows. Section II introduces the RQoS concept. Section III presents the RQoS Monitor architecture. Section IV details how the service monitoring process works. Section V presents the proof of concept as well as the performed experiments to validate and evaluate the monitoring mechanism. Section VI discusses related work. Finally, Section VII contains final remarks.

II. RELATIVE QoS

Several works in the literature [5]–[7] deal with QoS metadata as **absolute** values, i.e., a unique QoS value is considered in service monitoring. Such QoS values can be defined either from a single monitor or from different sources, so that monitored data are aggregated into a single QoS value by using mathematical formulas. However, in the Cloud Computing context, services are accessed via Internet and the absolute QoS noticed by the clients can vary depending on their location. This is due to the fact that the geographical location of both client and service provider interferes in the communication latency between them [4] (in terms of network infrastructure, package losses, package deviation, routing configurations, etc.) and hence in the perceived QoS. In this perspective, the **relative QoS** (RQoS) concept introduced in this work defines that the QoS is **relative** to the geographical location or physical machine where the service is being monitored. In other words, for each service, the absolute and relative QoS values can be different if the client and the cloud service are in different locations.

The client that consumes the cloud services usually is the same that consumes the QoS values generated by the monitoring mechanism. Therefore, if the monitoring mechanism is at the same geographical location of the client, then the QoS noticed by the client corresponds to the same QoS generated by the monitor. As an example, consider the cloud services, monitors, and platforms shown in Fig. 1. **Service 1** (deployed in USA) is monitored by **Monitor X** and is consumed by **Client X**, both deployed in a virtual machine located in Brazil. The same service is also monitored by **Monitor Y** and consumed by **Client Y**, both deployed in a machine located in Japan. According to the RQoS concept, **Service 1** has two different QoS values, one relative to **Monitor X** and other relative to **Monitor Y**.

In this work, the monitoring mechanism can be deployed in a virtual machine, so that the assessed QoS is relative to the geographical location of such a machine. However, as different virtual machines are physically allocated at the same cloud platform, the QoS is relative to the cloud platform location, not to a specific virtual machine. In cases in which a given platform has physical machines in more than one geographical location, the different instances of the same platform are considered as different cloud platforms. For example, as the Amazon Web Services (AWS) platform [8] has resources distributed in Brazil and in different regions of USA (east, west), Amazon Brazil, Amazon Western USA, and Amazon Eastern USA are considered as three different cloud platforms, thus meaning that a given virtual machine can be instantiated in the Amazon Brazil platform, not instantiated in the AWS platform itself.

III. RQoS MONITOR ARCHITECTURE

This work introduces a monitoring mechanism called **RQoS Monitor** to assess RQoS metadata of cloud services. The **RQoS Monitor** performs a monitoring process in the client perspective, so that it is possible to identify if a service is really on failure and/or if there is a communication problem between a service and a client. The **RQoS Monitor** architecture is divided into two different tools that work together, namely the **App Monitor** and the **Service Monitor**, as presented in the next subsections. Both tools were implemented by using the Java programming language.

A. App Monitor

The **App Monitor** is a monitoring tool deployed with the clients (applications) that consume the cloud services. As previously mentioned, service monitoring needs to run at the same geographical location or computational node of the client, so that the QoS values generated by the monitoring tool are the same values noticed by the client. The **App Monitor** adopts a hybrid (passive and active) monitoring approach [9]. The **passive monitoring** is characterized by the interception of any artifact generated by the client (e.g., requests, network packages, etc.) in order to communicate with the cloud service, thus retrieving relevant information about the monitored service (such as availability and response time). This monitoring approach is suitable to the Cloud Computing context as increasing the number of requests or data processing implies in the consumption of more resources, and then the client will pay more due to the use of the resources. However, if the service is not frequently used, the monitored information tends to be out-of-date.

On the other hand, in the **active monitoring** approach, the monitor itself performs periodic requests to the services (in a time interval defined by the client) in order to retrieve information to be used in the monitoring process. In this case, although the monitor will not consider outdated information, the performed requests increase the use of the cloud service and hence the value paid by the client. Therefore, the **App Monitor** performs the passive monitoring while the client makes requests to the service. When the client takes a long time without performing any requests, the monitor automatically triggers the active approach in order to update information about the services.

**Fig. 2** depicts the architecture of the **App Monitor**, which was developed as a Web service and is composed of the **REST Facade** and the **Monitoring Module**. The **REST Facade**...
manages the communication between the App Monitor and the Service Monitor and it provides the RESTful interface summarized in Table I. The RESTful service was implemented using RESTEasy [10], a set of frameworks to facilitate the development of RESTful Web services. In turn, the Monitoring Module is responsible for monitoring services and temporarily storing monitored data. It is composed of two main components, namely TCP Monitor and Blackboard. The TCP Monitor is responsible for monitoring the network interface of the machine in which the App Monitor is deployed in order to capture network packages created when clients make requests to the cloud service. The JnetPcap Java Library [11] is used by the TCP Monitor to intercept packages in the network interface of the machine, so that the captured packages enable the monitor to verify if the services are available and capture the response time of a request. The TCP Monitor can monitor only requests regarding a single service, so that the Monitoring Module must create an instance of a TCP Monitor for each monitored service. In turn, the Blackboard component works as a shared repository in which data monitored by the different TCP Monitor instances are stored until the Service Monitor requests these data by calling the getInfo method of the App Monitor.

B. Service Monitor

The Service Monitor is the tool responsible for retrieving monitored data from each App Monitor instance. Data retrieved by the App Monitor enable the Service Monitor to assess the following RQoS parameters for each service: (i) MTBF (Mean Time Between Failures), which is the mean time in which the service is in failure; (ii) MTTR (Mean Time To Recovery), which is the mean time spent by the service to recover from a failure; (iii) Uptime, which is the percentage of time in which the service was available; and (iv) Response Time, which is the mean response time of the service.

IV. MONITORING CLOUD SERVICES

As described in Section III-A, the App Monitor is deployed in the same computational node of the cloud service client. In the passive monitoring approach, the App Monitor intercepts requests to services used by the clients in order to retrieve the request response time and to identify if the request was successfully performed. Fig. 3 presents an activity diagram that illustrates this passive monitoring process. The App Monitor (specifically the TCP Monitor) monitors the virtual machine network interface (Activity 1) in order to capture packages created when the client makes a service request. When the TCP Monitor detects that a service request was made, it captures the network package and stores the date and hour of the event (firstTime) in the Blackboard component (Activity 2).

In order to avoid considering packages that do not represent a service request, the TCP Monitor performs a two-step filtering. First, it verifies if the captured package is a TCP/HTTP package, which might indicate a service request. Next, it verifies if there is a reference to the service in the package header. If the package represents a service request, the TCP Monitor stores the number of the port opened for that request in the Blackboard component and it waits for a response (Activity 3). When the response is captured, the TCP Monitor stores the date and time (secondTime) in which the package containing the response was received at the machine (Activity 4). Afterwards, the TCP Monitor calculates the response time by subtracting these time instants (Activity 5). Finally, the following request information are stored in the Blackboard: (i) the service data, such as name and IP address; (ii) the response time of the request; (iii) the success flag, which indicates whether the request was successfully performed or not; and (iv) the timestamp, which refers to the date and time of storage. When the response to the request is an error (e.g.,

![Fig. 2. RQoS Monitor architecture.](image)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>refreshConfig</td>
<td>Updates the monitoring configuration and starts the monitoring process.</td>
</tr>
<tr>
<td>getInfo</td>
<td>Receives a set of services as input and returns monitored data of these services.</td>
</tr>
</tbody>
</table>

Table I RESTful Interface of the App Monitor.
due to service unavailability, service error, timeout, etc.), the TCP Monitor stores a negative value for the response time and assigns the false value for the success flag. It is important to mention that a new entry containing this request information is stored in the Blackboard whenever a service request is performed, thus creating a service request history.

When the client does not perform several requests to a service, the passive monitoring approach becomes unfeasible as it needs requests to perform the monitoring process and the information about the services will become out-of-date. In order to overcome this problem, the monitor starts the active approach, i.e., it makes requests to services that were not requested for more than $n$ seconds ($timeInStandby$). The Monitoring Module maintains a timer to each service: such a timer is started with $timeInStandby$ and restarted whenever a request to the service is made. For instance, when the timer reaches zero, a request to the service is made and the timer is restarted to $timeInStandby$. During this time, if the client makes a service request, then the timer is restarted. After making the request, the monitoring process proceeds similarly to the passive approach as the TCP Monitor listens only to the network interface and it does not recognize the difference between a request from a client and a request from the Monitoring Module.

In parallel to the process performed by the App Monitor, the Service Monitor makes periodic requests to the App Monitor instances deployed in virtual machines in different cloud platforms. Requests to the App Monitor are made in a period of time greater than the one defined by $timeInStandby$ since requests made in a shorter interval may not return any new monitored data.

Before performing requests to all deployed App Monitor instances, the Service Monitor (specifically the Monitoring Module) generates a set of services that will be provided as input to the getInfo method of each App Monitor in order to retrieve the assessed RQoS parameters (see Table 1). Algorithm 1 describes the procedure for generating such a set of services. Given a set of cloud platforms ($PS$) where virtual machines are running, a set of virtual machines is retrieved for each cloud platform (line 2). Next, a set of monitored services ($services$) is retrieved (lines 5 and 6).

Algorithm 1 creates the service list per virtual machine and stores the information recovered by App Monitor instances.

**Input:** $PS$ – List of cloud platforms where virtual machines run an App Monitor instance

**Output:** $MAP$ – set of $<platform, service list>$ tuples

```
1: for each platform $p \in PS$ do
2:     machines $\leftarrow$ getMachinesByPlatform($p$)
3:     services $\leftarrow$ $\emptyset$
4:     for each machine $m \in$ machines do
5:         appMonitor $\leftarrow$ getAppMonitorByMachine($m$)
6:         servicesAP $\leftarrow$ getMonitoredServices(appMonitor)
7:         services $\leftarrow$ services $\cup$ servicesAP
8:         info $\leftarrow$ performRequestToAppMonitor(appMonitor)
9:         platform $\leftarrow$ getPlatformByMachine($m$)
10:    timeStamp $\leftarrow$ getCurrentTime()
11:    storeRequestInformation(info, platform, timeStamp)
12: end for
13: MAP $\leftarrow$ MAP $\cup$ $<p, services>$
14: end for
```

Afterwards, a request to the App Monitor is made to retrieve (i) the request response time, (ii) the flag indicating if the request was successfully performed, and (iii) the date and time in which the request was performed (line 8). This information ($info$) is stored in the Metadata Repository component in addition to the name of the cloud platform and current time (line 10). As a result, this procedure returns a set of tuples containing the name of the platform and the set of services monitored by all App Monitor instances deployed in virtual machines in this platform (line 13).

After retrieving all information monitored by the App Monitor, the Monitoring Module can perform the RQoS assessment, as summarized in Algorithm 2. Given a set of cloud platforms ($PS$) and the set of $<platform, service list>$ tuples returned by Algorithm 1 ($MAP$), for each platform $p$ in the set of platforms ($PS$), a set of monitored services ($services$) is retrieved in line 2. Next, a set of service requests is retrieved for each service $s \in services$ (line 4). With these data, the Controller component call each specialized assessor (lines 5 to 8). Finally, the Controller stores the assessed RQoS parameters in the Metadata Repository component (line 10).

Each RQoS assessor of the Service Monitor has its own assessment rules, which make use of the monitored data from the App Monitor. For instance, the response time parameter is determined by the arithmetic mean of the response time for all $n$ requests (see Equation 1):

$$responseTime = \frac{1}{n} \sum_{i=1}^{n} responseTime(i)$$  \hspace{1cm} (1)$$

The MTBF parameter is calculated by using the formula presented in Equation 2, in which $maxDate$ is the date and hour of the last request, $minDate$ is the date and hour of the first request, and $totalFail$ is the amount of failed requests, i.e., when the success flag is equal to false.
Algorithm 2 Assessment of RQoS parameters

Input: $PS$ – List of cloud platforms where virtual machines run an App Monitor instance, $MAP$ – set of <platform, service list> tuples

1: for each platform $p \in PS$ do
2: 
3: services $\leftarrow$ getServicesByPlatform($p$, $MAP$)
4: for each service $s \in services$ do
5: 
6:     data $\leftarrow$ getRequestData($p$, $s$)
7:     mtbf $\leftarrow$ assessMtbf(data)
8:     mttr $\leftarrow$ assessMttr(data)
9:     responseTime $\leftarrow$ assessResponseTime(data)
10:     upTime $\leftarrow$ assessUpTime(data)
11:     timeStmp $\leftarrow$ getCurrentTime()
12:     storeRelativeQoS($s$, $p$, $mtbf$, $mttr$, responseTime, upTime, timeStmp)
13: end for
14: end for

$$MTBF = \begin{cases} \frac{\text{maxDate} - \text{minDate}}{\text{totalFail}} & \text{totalFail} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$ (2)

The $MTTR$ parameter is calculated by using the formula presented in the Equation 3, in which $totalFailTime$ is the total time in which the service has been in failure and $totalFail$ is the amount of failed requests.

$$MTTR = \begin{cases} \frac{totalFailTime}{totalFail} & \text{totalFail} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$ (3)

Finally, the $uptime$ parameter is calculated by using the formula presented in the Equation 4, in which $totalFailTime$ is the total time in which the service has been in failure and $totalTime$ is the considered time window, so that $totalTime - totalFailTime$ represents the time in which the service has been available.

$$uptime = \begin{cases} \frac{totalTime - totalFailTime}{totalTime} & \text{totalTime} \neq 0 \\ 1 & \text{otherwise} \end{cases}$$ (4)

Besides the assessment of RQoS parameters, the Service Monitor also identifies service failures. When verifying that the $n$ last requests to a service were not successfully performed (i.e., the $success$ flag is false), the Service Monitor triggers a notification message to the client. Such a verification considers $n$ failed requests in order to avoid false positives. This is due to the fact that not always a single failed request means that the service is unavailable, as network oscillations hampering the communication might have happened, or even the service could be under a quick maintenance.

V. EVALUATION

A. Proof of concept

In order to validate our approach, we have implemented a cloud application called $LyricsTranslator$ inspired in [12]. This application receives as input the name of a song and returns the translated lyrics if the lyrics are not in the client’s language. The service provided by the application is not free, so that clients have to pay by the search and by the translation of the song. In order to achieve its business goals, the application performs a composition of four SaaS cloud services that are sequentially executed (see Fig. 4): (i) the $LyricsService$ service receives the name of a song and returns its lyric; (ii) the $TranslatorService$ service is responsible for receiving the lyric of a song and returning its translation; (iii) the $EmailService$ service receives an e-mail address and a message, and it sends such a message to the specified address; and (iv) the $BillingService$ service is responsible for charging the client.

The application was deployed in virtual machines instantiated in different cloud platforms located in Europe, USA, and Asia (see Fig. 5), so that several users could use the application over the Internet. In order to enable the $LyricsTranslator$ application to select the service with best quality, three replicas of $LyricsService$ and two replicas of the $EmailService$ and $Billing Service$ were created. It is important to notice that the $RQoS Monitor$ plays an important role in this scenario since it monitors services used by the application, thus enabling the assessment of RQoS parameters used to select the best service, as well as the identification of service failures.

In order to start the monitoring process, the client (application developer) must follow a sequence of steps to configure the $RQoS Monitor$. First, he/she must register the data about the services that will be monitored, namely $name$, $URI$ to the WSDL description of the service, and service access credentials (see Fig. 6). After registering the services to be monitored, the second step is to register information about
the deployed App Monitor. Fig. 7 present a screenshot of the wizard in which clients provide the URI referring to the location where the App Monitor is deployed, the services to be monitored, and the timeInStandby time. With this information, the monitor can start the monitoring process in the App Monitor.

Fig 8 shows a screenshot of the assistant in which the client must select the App Monitor and provide the timeToRequest time (equal to timeInStandby by default), i.e., the time interval in which the Service Monitor will perform requests to retrieve monitored data from the App Monitor. At this point, the RQoS Monitor can configure all App Monitor instances by calling the requestConfig method (see Table I). From this point, clients can monitor the RQoS variation and service failures on the screen. Fig. 9 shows the RQoS of the LyricsService relative to USA and Asia, and also that the monitored service is in failure for the latter.

B. Computational Experiments

We have also performed some computational experiments to show that there are variations in the QoS values depending on the geographic location where the services are monitored. These experiments have also aimed to assess the performance of the monitoring mechanism in terms of time spent for assessing the RQoS parameters.

In order to simulate a real cloud environment, the experiments were performed by using virtual machines instantiated in the Amazon Web Services [8] and Windows Azure [13] public cloud platforms, which are physically located in different regions of the world. Aiming at ensuring that QoS can really depend on the location of services and clients, we have created two other scenarios (see Fig. 10) in addition to the one depicted in Fig. 5. In each of these three scenarios, services run in different locations of the world (respectively, USA, Europe, and Asia) that is close to one virtual machine in which the application is deployed.

In the configured scenarios, the LyricsTranslator application with an instance of the App Monitor were deployed in three different virtual machines, each one allocated in the used cloud platforms. All machines used in the experiment had the same configuration, namely AMD Opteron™ Processor 4171 HE 2.1 GHz, 8 GB of RAM, and Linux Ubuntu operating system version 14.04. Moreover, the services consumed by the application were running in different locations of the globe (USA, Europe or Asia).

RQoS. In order to generate a significant amount of requests to services, the LyricsTranslator application was called every five seconds in a period of ten minutes. After ten minutes of monitoring, the application has performed 120 requests and the measurement of the RQoS parameters was conducted based on the data retrieved by these requests. In this experiment, we have chosen to analyze the response time because only it has had variation since other measured parameters depend upon the unavailability of the services and this has not happened in the experiments. Therefore, the uptime, MTTR, and MTBF
parameters were respectively set with values 1, 0, and 0 for all services.

Considering the scenario depicted in Fig. 5, Fig. 11a shows the average response time of services (in milliseconds) for each of the three machines (Europe, Asia, and USA). It is possible to observe that the average response time is much smaller for the machine located in USA for all services, followed by the machine located in Europe, and then the machine located in Asia, which has presented the highest response time. This happens because the services are deployed in a machine also located in USA, so that machines that are geographically closer to the region where the services are running have a smaller response time. In turn, considering the second scenario depicted in Fig. 10 (left), Fig. 11b shows that the average response time is much smaller for the machine located in Europe in comparison with the machines located in USA and Asia since the services are running in a machine located in Europe. Finally, considering the third scenario (Fig. 10, right) in which the services are deployed in a virtual machine instantiated in a cloud platform located in Asia, Fig. 11c shows that the average response time is much smaller for the machine located in Asia. Therefore, we can conclude that the quality of service can really vary depending on the geographic location in which it is being monitored/consumed.

**Parameters assessment.** The other conducted experiment aimed to calculate the time required to assess the RQoS parameters by the Service Monitor. This time is calculated from the moment in which the `getInfo` method is called until the moment when the metadata are saved in the database after their assessment. In the performed experiment considering the scenario presented in Fig. 5, the RQoS parameters were assessed for a particular service to a single machine. Moreover, as the Service Monitor performs periodic measurements to recover information monitored by the App Monitor, we have considered three different variations for `timeInStandBy` (waiting time), namely 10, 20, and 30 seconds. This variation of time enables the App Monitor to monitor more requests to a service in a machine, so that the amount of data to be used increases with the variation on the time between measurements.

As shown in Fig. 12, the process for assessing the RQoS parameters was divided into eight operations, namely: (i) `getInfo`, which refers to calls to the App Monitor for retrieving monitored information; (ii) `search`, which refers to searches for previously assessed service metadata in the Metadata Repository database; (iii) assessment of the MTTR parame-
ter; (iv) assessment of the response time; (v) assessment of the MTBF parameter; (vi) uptime, assessment of the uptime parameter; (vii) save, which refers to the operation for saving metadata in the Metadata Repository database; and (viii) the total assessment time disregarding the getInfo operation. These measured times shown in Fig. 12 refer to each operation in the assessment process of RQoS parameters of a service for a specific machine by varying the value of timeInStandBy. The chart shows that the getTime operation (that fetches data monitored by the App Monitor) is responsible for most of the assessment time. The time spent for the getInfo operation depends on the value of timeInStandBy since greater is the time spent by App Monitor for monitoring services, greater is the amount of data to be transferred over the network.

It is also possible to observe in Fig. 12 that the time spent for the search and save operations (that access a database) has interfered in the assessment time, even though spending a much smaller time than the getInfo operation. In addition, the time spent for the operations that calculate the values of the RQoS parameters has not interfered in the time for the overall operation. Finally, the difference between the total assessment time (disregarding the time spent by the getInfo operation) for a configuration with 10 seconds and for other configuration with 30 seconds was only 1 ms, whereas the time for the getInfo operation has varied in 73 ms. This means that the data transfer over the network is the factor with major impact on the assessment time.

VI. RELATED WORK

Li et al. [14] propose a service selection algorithm based on QoS by considering the location of the application that consumes the cloud services and using the response time as a metric to evaluate the service quality. As client and service are distributed in different locations, the latency in the communication between them interferes in the service response time noticed by the client. Although the proposal is an algorithm and not a monitoring mechanism, it is similar to our proposal as it uses the location of clients and services to assess QoS parameters. However, it is not presented when the response time between the client and the service is calculated or if any monitoring mechanism is available for keeping the service information updated. Finally, the Li et al.’s work only deals with response time as a QoS parameter, which may be not enough to define the service quality.

Although there are no many studies concerned with the relative QoS concept, the literature also presents a significant amount of work regarding cloud service monitoring. For instance, Alhamazani et al. [6] develop monitoring techniques for optimizing and ensuring QoS of the applications that use cloud services. The monitoring process proposed in such a work can be divided into two parallel processes, namely (i) monitoring QoS parameters related to the virtual cloud services, and (ii) monitoring the application itself. They also propose that the monitoring strategy should be implemented using a model that gathers and assesses QoS information by capturing exchanged messages between the distributed components of the application and the cloud services. Despite the Alhamazani et al.’s work is theoretical and it does not present details about the assessment of the QoS parameters or how to capture the messages that will enable the monitoring process, it is possible to state that using data (packages, messages, etc.) already generated between applications and services is a feasible strategy in the Cloud Computing context.

Finally, Mdhaffar et al. [15] propose AOP4CSM, a monitoring strategy for cloud services that does not require accessing or changing the source code of the services. For this purpose, the proposed monitor uses aspect-oriented programming (AOP) techniques for gathering QoS parameters of SaaS cloud services, as in our work. By using AOP, AOP4CSM defines join points to intercept the calls to methods of the clients and services and to retrieve data in four time instants: (i) when the client performs the request; (ii) when the request is received on the server; (iii) when the response is sent; and (iv) when the client receives the response. The use of AOP for retrieving relevant data about the monitoring process can be seen as a passive monitoring strategy (as our monitor does), but the Mdhaffar et al.’s work does not consider the scenario in which clients do not make periodical requests to the service and the monitor will not have up-to-date QoS data. In contrast, our proposal automatically starts an active strategy in this case.

VII. FINAL REMARKS

In this paper, we have presented the relative QoS (RQoS) concept, which defines that the service quality is relative to the geographical location where the service is monitored. In order to show the applicability of the concept, we have also developed RQoS Monitor, a monitoring mechanism that provides RQoS parameters regarding cloud services in the SaaS (Software as a Service) layer. This monitoring mechanism is composed by two different tools that work together, the App Monitor and the Service Monitor. The former is deployed with the clients (applications) that consume the cloud services and it adopts a hybrid (passive and active) monitoring approach, whereas the latter is responsible for retrieving monitored data from App Monitor instances and assessing RQoS parameters. As the developed monitor provides RQoS parameters regarding cloud services, clients can choose the services that better suit their needs.
In order to validate the concept and the monitoring mechanism, we have implemented a proof of concept application and performed some computational experiments. The obtained results have shown that QoS parameters can really vary depending on the location of clients and services. We have also noticed that the data transfer from the App Monitor to the Service Monitor is the factor with major impact on the assessment time of the RQoS parameters. As future work, we intend to expand the developed approach to consider IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) services, as the RQoS concept is also valid for these cloud layers.

REFERENCES


