SLA and Profit-aware SaaS Provisioning through Proactive Renegotiation

Aya Omezzine[1,2,3] and Narjes Bellamine Ben Saoud[1]
[1]Univ. Manouba, ENSI, RIADI LR99ES26
Campus Universitaire Manouba, 2010, Tunisie
[2]Université Fédérale Toulouse Midi-Pyrénées, CNRS/LAAS
F-31400 Toulouse, France
Email: aya.omezzine@gmail.com, Narjes.bellamine@ensi.rnu.tn

Said Tazi[2,3] and Gene Cooperman[2,4]
[3]Univ. de Toulouse, UT1, LAAS
F-31000 Toulouse, France
[4]College of Computer and Information Science
Northeastern University, Boston, MA / USA
Email: tazi@laas.fr, gene@ccs.neu.edu

***Abstract*—Software-as-a-Service (SaaS) providers offer on-demand, highly scalable applications to the end users. To maximize their profit, the providers must make profit-aware scheduling decisions about assigning client requests to virtual resources, while respecting the agreed upon Service-Level Agreement (SLA). Given the highly dynamic nature of the cloud environment, unexpected events may affect the initial scheduling plans, which leads to unanticipated SLA violations. Thus, an unaccounted event may create a lose-lose situation between provider and client. If the SLA is violated the provider must pay the potentially high penalty that is negotiated within the original SLA. But from the client's viewpoint, an SLA violation may cause cancellation of a business-critical job, and no ordinary SLA penalty can compensate for the loss of the client's business. The provider's reputation could also suffers as the number of such SLA violations grows, resulting in loss of future clients. On the contrary of most existing work that assume that once established the SLA cannot be modified, we propose to convert the lose-lose situation into a win-win one through an automated renegotiation mechanism. When an event threatens a lose-lose violation of the SLA, the renegotiation mechanism is launched to establish a new SLA that limits the losses on the two sides. Experiments show that this new approach minimizes the loss in profit of the provider and minimizes the number of cancelled jobs experienced by the client, as compared with enforcing the original SLA.**

***Index Terms*—Cloud computing; SaaS provisioning; Service Level Agreement (SLA); SLA-aware scheduling; Client satisfaction; Automated renegotiation; Decision making strategy;**

I. Introduction

SaaS providers offer highly scalable applications to end users over the Internet. To run their applications, SaaS providers often prefer to rent virtual resources from an Infrastructure-as-a-Service (IaaS) provider instead of in-house hosting. By doing so, they avoid infrastructure maintenance and they can scale their application to serve as many end users as possible. Thus, the end user negotiates with a SaaS provider, while that provider in turn schedules jobs with IaaS providers.

The SaaS application provisioning must satisfy the SLA contract established between the two parties. The SLA contract is a formal representation of the QoS parameters, obligations of the two parties, and provider penalties, that are agreed upon. In order to maximize their profit and to satisfy end users, the SaaS providers use an SLA-aware scheduling algorithm, which efficiently assigns user requests to virtual resources offered by IaaS providers.

978-1-5090-3216-7/16/$31.00 ©2016 IEEE

Cloud computing represents a highly dynamic environment (both at the business level and at the resource level). There may be unforeseen events at the resource level such as catastrophic resource failure, or else there may be unexpected events at the business level coming from the need to share rented resources between new clients that compete for immediate execution. These events may result in violation of the original negotiated SLA, since the schedule originally done (based on the initial SLA) can be modified.

Generally if a contract is violated, a penalty is paid and the service is canceled [1], [2]. But if a contract is violated due to circumstances not accounted for in the original SLA negotiation, the two parties may *both* lose badly. For example, consider the situation in which a job is critical to the success of the business. In principle, the client could have insisted on a penalty in the original SLA that is equal to the value of client's business, as compensation for the losses due to the failure of that business-critical job. But this is usually unrealistic, since such a penalty can be larger even than the total assets of the provider. Hence, the client will never be fully compensated, and the provider faces a loss of future clients due to the loss in reputation as the number of violated jobs accumulates.

For these reasons, the provider and the client would normally prefer to renegotiate using a new SLA with a new deadline (i.e., an extension beyond the first deadline), rather than pay a steep penalty and accept the cancellation of a business-critical job. The new SLA will generally include a discount by the provider on the originally agreed-upon price, as a concession by the provider for avoiding the steep penalty envisaged by the original SLA violation.

Most of the literature assumes that once an SLA is established, it cannot be renegotiated [1], [2], [3], [4]. The concept of *renegotiation* has not yet been well studied [5]. There is some work that tries to enhance the WS-Agreement negotiation protocol using renegotiation [6], [7], [8], and others propose general conceptual renegotiation frameworks [9], [10]. However, the term *renegotiation* in this prior literature always refers to a renegotiation phase within the *original SLA*.

In particular, the prior work mentioned above does not propose any decision-making model to guides the renegotiation process toward a satisfactory agreement. In contrast, the current work proposes decision-making strategies in which the negotiators renegotiate based on a concession (lowering) of the original penalty.

The *key novelty of this paper* is to propose an automated renegotiation-based approach when detecting an unexpected event during the SaaS provisioning process. In our approach, the provider proactively renegotiates with the clients whose jobs may be in violation of the SLAs, in order to minimize the loss in profit and in order to assure the continuity of the service. The renegotiation approach is composed of two steps.

1) The first step happens when the SaaS provider detects an unexpected event. Since the provider may not be able to physically continue some jobs with the same scheduling parameters (VM, completion time, etc.), we consider alternative rescheduling options for the provider. The first step consists of the selection of an option for profit-aware rescheduling. In examining the possible scheduling options, the provider chooses an option leading to a minimum loss in profit while also minimizing the number of canceled jobs. (See Section III-C.)
2) At the second step, the SaaS provider triggers a renegotiation with those end users whose jobs may terminate after deadline. The renegotiation consists of an exchange of offers and counter-offers guided by decision-making strategies using a utility model that is based on economics. The strategies followed by the SaaS provider are based on the rescheduling option selected in the first step, above. (See Section IV-A and following.)

The rest of the paper is organized as follows. Section II presents the basis for negotiation, how it is used in cloud computing, and especially the role of renegotiation. Section III and Section IV describe the first step and second step, respectively, of the renegotiation-based SaaS provisioning process. Section V presents experiments to assess the algorithm. Section VI is dedicated to discussing the related work. Conclusions are presented in Section VII.

II. Motivation and Background

Cloud computing presents a highly dynamic marketplace for delivering IT services on demand. Each cloud actor has its own interests. In particular, the client aims to obtain the most convenient service at the cheapest price, while the provider aims to maximize its profit and to serve the maximum number of clients. The negotiation between cloud actors is an intuitive way to solve conflicts between client and provider and to reach a satisfactory agreement. As the infrastructure and platforms for the cloud become more complex, we need more automated negotiation to handle that interaction.

Automated negotiation can be split into three issues [11]. 1) The *negotiation protocol* expresses the locutions that may be exchanged between the negotiators and defines the rules of interaction. 2) The *negotiated service* is composed of objects (also called issues) about which the participants negotiate. There are specific service issues, which concern the service type, and there are generic issues such as the price. 3) The *decision-making model for negotiation* defines the decision-making strategies for each actor.

Decision making is composed of two main strategies. The first strategy allows one to evaluate a received offer and to decide whether to accept, reject or propose a counter-offer. This first strategy is generally based on a utility function that measures the degree of satisfaction of a received offer according to the preferences. The second strategy enables one to generate offers or counter-offers at each step.

Automated negotiation in the Cloud is primarily used to establish an SLA between clients and providers. It happens generally in the first phase of the service-provisioning process (before the SLA establishment itself). In this paper, we focus on SaaS application provisioning and especially on compute-intensive applications. Some examples are: scientific data processing, and finance data analysis. In the first phase, in order to maximize the number of clients and minimize the costs of renting sufficient computer resources, the SaaS provider adopts a profit- and SLA-aware scheduling algorithm. The schedule must guarantee that the SLA is met, while also maximizing the profit.

After signing the contract, *unexpected events* may later occur that can impact the current scheduling. To avoid an SLA violation, the provider must take rescheduling actions. However, maybe no rescheduling can meet the previously signed SLA. For example, migrating a job to another VM after failure may delay the completion time beyond the agreed upon deadline. The SLA model generally assumes that once the deadline is violated, the job is automatically cancelled and a penalty is paid.

This is the scenario in which automated renegotiation becomes important, as part of a second phase of service provisioning. This second phase occurs after violation of the original SLA due to an unexpected event. We invoke a new renegotiation phase between the end user and the SaaS provider, in order to minimize the SLA penalty costs and in order to ensure the continuity of service on which the provider's reputation depends.

In what follows, we detail how automated renegotiation can be used to handle unexpected events, as part of a second phase in SaaS application provisioning.

III. Selection of an Option for Profit-aware Rescheduling

When detecting an expected event that alters the initial scheduling, the provider takes rescheduling actions in order to avoid SLA violations to the extent possible. Generally, the provider may have more than one rescheduling option. For this reason, we propose an algorithm for the selection of an option for profit-aware rescheduling. In this section, we model first the unexpected event and the rescheduling option. Then, we present our algorithm for selection of a rescheduling option.

A. Definition of an Unexpected Event

An unexpected event leads to a change in the situation under which the already signed SLAs had originally been negotiated. Indeed, the schedule contracted by the SaaS provider in the first phase may be affected, thereby leading to violation of the original SLA. The unexpected events can be classified into two categories: 1) *resource events*, for example VM failure, failure of the currently executing job, etc. The jobs scheduled on that VM may be affected, and so the initial scheduling may be altered. 2) *business events*, such as a new incoming client needing immediate execution with no additional VMs available from the IaaS provider. Thus, the SaaS provider may choose to execute a new job on an already active VM even though there exist prior jobs (either running, or scheduled but not yet started). An unexpected event can be specified using two parameters: the time at which the event occurs, t_{event}; and the set of resources affected by the event, vm_{ID}. The unexpected event is assumed to be detected just prior to SLA violation through a monitoring module.

B. Definition of the Rescheduling Option

The rescheduling option is composed of potential scheduling actions applied to accepted and scheduled jobs (which may either be running or not yet started). Two examples of rescheduling actions are: (i) the provider may proactively migrate the job to a different computer; and (ii) the provider may invoke periodic checkpointing to protect against catastrophic failure. (The provider can then restart from a previous checkpoint image on a new computer, or even directly migrate to a new computer.) A scheduling action defines when and where to place a job. A scheduling action Ac, applied to a job j, can be defined as $(type, j, estimated_start_time, vm_{ID}, compT)$ where: the $type$ denotes the scheduling action type. For example: insert, postpone, cancel/restart, migrate,suspend/restart. The $estimated_start_time$ and vm_{ID} define when and where to start the job, respectively. The $compT$ denotes the estimated completion time, and can be calculated based on the information given by scheduling action.

Hence, a rescheduling option, denoted Op, is defined as follows: $Op = \{Ac_j\}$, where $j \in \{\text{rescheduled_jobs}\}$. Each rescheduling option has as output a list of rescheduled jobs ($resch_List$) and the list's rescheduling information given by the scheduling action.

C. Algorithm for Selection of a Profit-aware Rescheduling Option

For the selection of a rescheduling option, we consider two metrics: 1) the $lossInProfit_p$, which calculates the SaaS provider loss in profit when choosing rescheduling option p; and 2) the number of potential cancelled jobs when choosing option p, denoted by $nbrJobs_p$.

1) *The $lossInProfit_p$ for the provider:* This include two parameters: a) The $actionCost$ define the cost due to the action. For example, if the action is to migrate the job to a new VM, the cost of the action will be equal to the price of provisioning a new VM. b) The $penaltyCost$, defined as the SLA violation cost, which can be calculated for a job j using the following formula:

$$penaltyCost_j = \begin{cases} 0, \text{if } compT_j \le respT_j \\ pr_j * delay_j, \text{if } respT_j \le compT_j \le dl_j \\ fixedPenalty_j, \text{if } dl_j < compT_j \end{cases} \quad (1)$$

where $respT_j$ is the agreed upon *response time*, and dl_j is the agreed upon *deadline*. The value pr_j indicates the *penalty rate*. The $fixedPenalty_j$ denotes the penalty paid in case of violation.

2) *The number of potential cancelled jobs, $nbrJobs_p$:* This calculates the jobs whose estimated completion time $compT_j$ are greater than the deadline dl_j of the initial SLA.

The proposed algorithm, Algorithm 1, below, takes as input the list of possible rescheduling options that the provider can choose after detecting the unexpected event. The algorithm returns a scheduling option and associated rescheduling information for each job such that the number of cancelled jobs is minimized and the loss in profit is minimized.

For each possible rescheduling option: First, Algorithm 1, below, calculates the estimated completion time $compT_j$ for each job j, based on the action Ac_j applied to this job (line 5). Second, the algorithm calculates the $lossInProfit$ as a sum of the loss in profit for each of the rescheduled jobs (lines 6 and 7). Third, the algorithm selects the potential cancelled jobs whose $compT$ are greater than the agreed upon deadline, calculates $nbrJobs$, and stores the rescheduling information for those jobs in $potentialCancelJobs$ (line 8 to 10). Then, the algorithm stores the option results in $optionsResults$ (line 13). Finally, the algorithm selects the option noted $optionsResults_s$ having the minimum $nbrJobs$ and the minimum $lossInProfit$ (line 14) using the $selectBestOption$ function. We propose to use utility functions in order to select the most convenient option. The utility function of an attribute i with value x can be calculated as follows.

$$\text{Let } U(x_i) = \frac{x_i - x_{worst}}{x_{best} - x_{worst}}, \quad (2)$$

where x_{worst} and x_{best} denote the best and worst value, respectively. **For each possible rescheduling option**: the $selectBestOption$ function calculates the utility values $UtLoss$ and $UtNbrJobs$ for $lossInProfit$ and $nbrJobs$ using equation 2 (line 21 and 22). The worst and best values for the $lossInProfit$ are the maximum and minimum $lossInProfit$ values selected from the $optionResults$, respectively. Likewise, the worst and best values for $nbrJobs$ are the maximum and minimum $nbrJobs$ values selected from the $optionResults$, respectively. Then, the function calculates the option's distance to the best option, which has $UtLoss = 1$ and $UtNbrJobs = 1$ (line 23). Finally, the function returns the option having the minimum distance to the best option (line 24 to 27).

Algorithm 1 Pseudo-code for selection of rescheduling option

Input: The list of possible rescheduling options
Output: The rescheduling option leading to the min $lossInProfit$ and min $nbrJobs$

1: **for** each $p \in$ possible rescheduling options **do**
2: $lossInProfit_p = 0$
3: $nbrJobs_p = 0$
4: **for** each $j \in resch_List_p$ **do**
5: $compT_j = getCompT(Ac_j)$
6: $lossInProfit_j = penaltyCost(compT_j) + actionCost(Ac_j)$
7: $lossInProfit_p = lossInProfit_p + lossInProfit_j$
8: **if** $compT_j > dl_j$ **then**
9: $nbrJobs_p + +$
10: Add resch info from $resch_List_j$ to $potentialCancelJobs_p$
11: **else**
12: **continue**
13: Store $lossInProfit_p$,$nbrJobs_p$,$potentialCancelJobs_p$ in $OptionsResults_p$
14: $OptionsResults_s$=selectBestOption($OptionsResults$)
15: **return** $OptionsResults_s$

16:
17: **Function** selectBestOption($OptionsResults$)
18: $minDistance = \sqrt{2}$
19: $optionsResults_s = optionsResults_p$
20: **for** each $p \in OptionsResults$ **do**
21: $UtLoss_p = U(lossInProfit_p)$
22: $UtNbr_p = U(nbrJobs_p)$
23: $Distance_p = \sqrt{(UtLoss_p - 1)^2 + (UtNbr_p - 1)^2}$
24: **if** $Distance_p < minDistance$ **then**
25: $minDistance = Distance_p$
26: $optionsResults_s = optionsResults_p$
27: **return** $optionsResults_s$

IV. The Renegotiation-based Rescheduling Procedure

Once a rescheduling option is selected, the provider will renegotiate with the clients whose jobs may be cancelled by triggering a *renegotiation session* with each client. The values of the renegotiable issues (deadline, compensation) will be guided by the renegotiation decision-making strategy and will be based on the results of the selected rescheduling option. In this section, we present first the overall process for renegotiation. Then we present details about the strategies that will be followed by the provider and the client.

A. The renegotiation overall process

A renegotiation session can be defined as the period covering the time when the interaction between negotiators begins until it stops. The renegotiation session terminates either with an agreement, and in this case the new SLA is applied, or without an agreement, in which case the initial SLA is applied. The different states of the renegotiation session, denoted $renegSessionState$, are: 1) $Active$ (when the two parties are exchanging offers and counter-offers); 2) $Succeeded$ (when the renegotiation session terminates with an agreement if one party accepts the offer received from his opponent); 3) $Failed$ (when the renegotiation session terminates without an agreement). This last situation ($Failed$) occurs when one party rejects the opponent's offer or when the negotiation deadline is reached.

The renegotiation-based rescheduling algorithm, Algorithm 2, takes as input the $potentialCancelJobs$ list (included in the $optionsResults$ returned by Algorithm 1). For each job that may be cancelled, the provider opens a renegotiation session with the client that owns that job. The renegotiation sessions are triggered sequentially. The provider opens a new renegotiation session only if the current one is terminated (lines 3 and 4). If the renegotiation terminates with success, then the SLA is updated to include the new agreed upon deadline and the compensation (lines 5 and 6). If the renegotiation about the job j fails then the provider must update the estimated completion time of the jobs that potentially are rescheduled after job j, in order to avoid the resource wastage due to unused time slots (lines 8 to 10). For that reason, the renegotiation is done sequentially, so that the provider can update the estimated completion time of the rescheduled jobs based on the renegotiation session's output.

Algorithm 2 Pseudo-code for renegotiation-based rescheduling

Input: The list of potential cancelled jobs
Output: The results of each renegotiation session

1: **for** each $j \in potentialCancelJobs$ **do**
2: open renegotiation session j with owner of job j
3: **while** $renegSessionState_j == Active$ **do**
4: **wait**
5: **if** $renegSessionState_j == Succeded$ **then**
6: update the SLA_j
7: **continue**
8: **else if** $renegSessionState_j == Failed$ **then**
9: **for** each $k \in$ rescheduled_jobs after j **do**
10: update $compT_k$ in $potentialCancelJobs_k$

B. The Decision-making Strategies for Renegotiation

During the renegotiation session, the provider and client automatically exchange offers and counter-offers according to their decision-making strategies. The renegotiation strategy should be designed to rapidly achieve agreement, since the participants are generally pressed when renegotiating after an SLA violation. For this reason, we assume that the new deadline proposed by the provider in the first round cannot be modified when exchanging offers and counter-offers. This is because the proposed deadline value is imposed by the rescheduling option selected. So the given deadline value is the best that the provider can offer to the client.

In what follows, we present how the compensation value is evaluated and generated during the renegotiation session.

1) Decision-making by the Provider:

The offer evaluation strategy: The offer evaluation is based on the satisfaction model described in [12]. The utility value of a negotiable attribute i with value x can be calculated using equation 2 where the worst and best values are defined by the negotiator before starting the negotiation as internal preference. In our scenario, based on the SLA model described in equation 1, the values $penaltyCost(deadline)$ and $fixedPenalty$ denote the best and worst values of compensation, respectively.

The acceptance conditions of a received offer from the client during the renegotiation session are: 1) $U(compensation_received) >= U(compensation_proposed)$; and 2) $deadline_received > compT$.

If the offer received from the client does not satisfy the two conditions mentioned above, the provider will propose a counter-offer using the *utility-based offer generation* strategy.

The Strategy for Generation of Utility-based Offers: As mentioned earlier, the proposed new deadline will be equal to the estimated completion time included in $potentialCancelJobs$ list. Given the expected compensation utility for the provider, the compensation value can be generated using equation 2 of Section III-C. The *expected utility* consists of a tradeOff between minimizing the loss in profit and satisfying the client. The expected client utility can vary between 0 and 1. In the special case when the utility is equal to 1, the provider proposes a minimum compensation (the provider's best value) while still managing to relax the deadline. So, in this case the provider prefers minimizing the provider loss over satisfying the client.

And in the special case that the utility is equal to 0, the provider proposes to pay the fixed penalty as compensation while continuing to run the job. So, the provider doesn't minimize the provider loss, but instead satisfies the client by not cancelling his job. The SaaS provider can offer this to the client only because it had obtained additional resources during the rescheduling phase with the IaaS provider, as described in Section III-C. Before renegotiating and according to the provider's internal preferences, the provider has fixed values for *preferred* and *reserved* utility values (upper and lower bounds on the expected utility). Those values are kept secret and are not know by the client. In the first round, the provider generates the initial offer based on the provider's preferred utility. During the later rounds, the provider may back off from its preferred utility until reaching its reserved utility.

2) Decision-making by the Client:

The Strategy for Offer Evaluation: The evaluation is based on the *overall utility* value. The overall utility of a received offer composed of n attributes is calculated as a weighted sum of each single utility using the following equation:

$$U(offer) = \sum_{i=1}^{n} w_i * U(x_i) \tag{3}$$

where w_i is the weight expressing the importance of the attribute i and w_i is in the range $[0, 1]$. For example, for high priority jobs, users may place more importance on the deadline than on compensation. In contrast, for low priority jobs, users may place more importance on the compensation than on the deadline.

The client defines preferred ($preferredUt$) and reserved ($reservedUt$) utility values, as bounds on the overall expected utility. Those clients having urgent business-critical jobs assign low value to the ($reservedUt$). This is because they prefer to accept the job along with a relaxed deadline and a smaller compensation, rather than having the job cancelled. The client preferences are kept secret.

In our scenario, the client accepts an offer only if $U(offer_{received}) \geq reservedUt$. The client rejects an offer if $\exists\, issue\; i, U(x_i) < 0$. Otherwise, the client proposes a counter-offer using the following strategy.

The Strategy for Offer Generation: As mentioned earlier, the client does not change the deadline value proposed by the provider when generating a counter-offer. Since the deadline utility is known (expressed by the provider's initial offer), the compensation utility value can be generated from the expected overall utility using equation 3. As was the case for the provider, the client similarly starts by generating an offer according to the $preferredUt$ value, until reaching the $reservedUt$ value.

V. Evaluation and analysis

A. Experimental settings

To simulate the cloud market and the interaction between the SaaS provider and the final users, we implement a multi-agent system using JAVA and the Java Agent DEvelopment framework (JADE) [13]. Each software agent is acting on behalf of either clients or providers. The agents negotiate through the FIPA iterated contract net protocol, which is a multi-round negotiation protocol [14].

In the SaaS application provisioning process there are two phases: 1) Before the SLA establishment: the provider tries to find a schedule satisfying the client request, and decide whether to accept or reject the request. Once accepted, an SLA is signed between the two parties (between SaaS provider and client). 2) When an unexpected event occurs after the SLA establishment, as we have presented in Section III, there are two steps. The first step happens when the provider initially detects an unexpected event that may alter the initial schedule. The first step deals with choosing an option from several possible rescheduling options. In the second step, the provider triggers a renegotiation session with each user whose SLA may be violated.

Since we are interested in testing and validating the renegotiation approach, we assume in our experiments that:

- The first phase is done according to an existing SLA-aware scheduling algorithm [3]. That algorithm performs more efficiently when evaluated and compared with the reference scheduling algorithms [3]. For each accepted job, the output of the first phase is an SLA with the

required scheduling information. The scheduling information indicates where and when to put the job to satisfy the SLA.

- The first step of the second phase is not explicitly implemented. Instead, we generate an unexpected random event. We implement a rescheduling module simulator that generates the list of potential rescheduled jobs and their estimated completion time given an unexpected event. We assume that the jobs are rescheduled sequentially. The estimated completion for the job running can be generated randomly and for the other jobs using the following formula.

$$compT_j = compT_{jr} + \sum_{k \in \{\text{k between jr et j}\}} procT_{k,l} \quad (4)$$

where $compT_{jr}$ denotes the completion time of the job running jr at t_{event}. And $procT_{k,l}$ denotes the processing time of job k on the VM of type l.
We assume that the rescheduling module simulator chooses the best rescheduling option.

B. Results and Analysis

Our objective is to evaluate the renegotiation-based application provisioning algorithm and to compare it to the basic scenario in which the provider cannot modify the established SLA. For the basic scenario, we assume that the provider tries to execute a rescheduling action (step 1) without any renegotiation. If the SLA is violated the job is cancelled and the SLA penalty is paid. We measure performance using two metrics: 1) the total loss in profit, expressing how much the provider loses when violating an already established SLAs; and 2) the number of cancelled jobs, the number of jobs whose completion time is beyond the agreed upon deadline for the original SLA.

We conduct three types of experiments in which we calculate the loss in profit and the number of cancelled jobs. For these experiments, we assume that each agent (provider or consumer) is able to generate only one offer during the renegotiation session, since the renegotiation must be done in a timely manner. Furthermore, we assume for the expected utility that the agent's reserved utility is equal to the preferred one. So the agents generate one offer according to their expected utility. If the opponent accepts the offer, the renegotiation ends with an agreement. Otherwise the renegotiation fails. This configuration (where preferred utility is equal to the reserved one) is the worst possible configuration in negotiation, since it is the least flexible. By choosing this configuration, we will be sure that for other configurations, our renegotiation algorithm will perform better. Hence, when relaxing the expected utility, there is a greater chance of a request/offer being accepted, and so the number of successful renegotiation sessions will be increased.

For the first and the second experiments, we vary the expected utility for the provider and the client, respectively, while injecting exactly one unexpected event (affecting only one VM). For the third experiment, we vary the number of

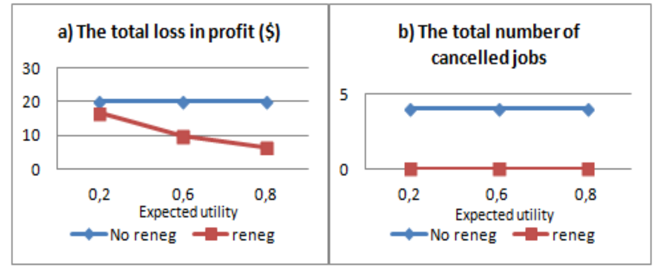

Fig. 1. Impact of provider's expected utility variation

resources affected by the unexpected event. Note that an event may lead to altering the initial scheduling of more than one VM. For example, a failure may affect many VMs.

1) Impact when varying the expected utility of the SaaS provider: Figure 1 shows the different values obtained for the loss in profit and the number of cancelled jobs with respect to the provider expected utility. For those experiments, we generate clients and their initial request with expected utility equal to 0.1 (clients with business-critical jobs). We observe that the loss in profit and the number of cancelled jobs using renegotiation is minimized compared to the basic scenario. Without renegotiation, the loss in profit and the number of cancelled jobs are constant. regardless of the value of the provider utility. This is expected, since the provider's strategy for handling unexpected events does not consider the value of the provider utility.

In Figure 1(a), the loss in profit (red curve) is decreasing when the provider's expected utility increase. This is because the utility is related to the compensation paid to the client. The higher the utility, the less is the compensation that is paid, and so the loss in profit is also less. In Figure 1(b), the number of cancelled jobs (red curve) is constant regardless of the value of the provider utility, this is because the client's reserved utility is at the lower limit. This implies that the client will accept any offer from the provider, even if the compensation is not at the upper limit (not at the upper bound for the provider utility). For those clients, a lower utility is nevertheless better than cancelling the job.

In the next experiments, we will vary the clients' expected utility.

2) Impact when varying the expected utility of the clients: Figure 2 shows the different values obtained for the loss in profit and the number of cancelled jobs, with respect to the clients' expected utility. For those experiments, the provider's expected utility is equal to 0.6. We note, as in Figure 1, that the loss in profit and the number of cancelled jobs are constant in the basic scenario, since the basic scenario does not take into account client satisfaction.

With renegotiation, we notice that the loss in profit and the number of cancelled jobs increase when the client expected utility increases. For the users with low utility values, the renegotiation algorithm performs much better than the basic one. But, for users with high utility values, the renegotiation algorithm results are the same as the basic one. So, when

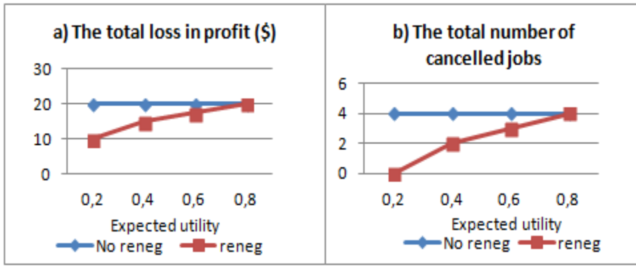

Fig. 2. Impact of variation of clients' expected utility

increasing the clients' expected utility, the renegotiation algorithm performance tends to the performance of the basic algorithm. In contrast, when the expected utility is low, the client has a high-priority business-critical job, and so it accepts any renegotiation offer in order to assure the continuity of its business. In contrast, the client with a high expected utility (i.e., having a less business-critical job) may choose to not accept a renegotiation offer. In this case, the client prefers that the provider should pay the penalty and cancel the job.

3) Impact as the number of resources are varied: Figure 3 shows the different values obtained for the loss in profit and the number of cancelled jobs with respect to the number of affected resources. For those experiments, the provider and the client expected utility are equal to 0.6 and 0.1, respectively.

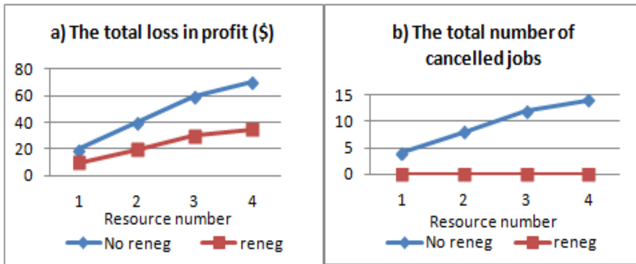

Fig. 3. Impact of variation of number of resources

We notice that the loss in profit (with and without renegotiation) and the number of cancelled jobs (without renegotiation) increase when the number of affected resources increases. Further, when the unexpected event affects many VMs, the number of rescheduled jobs increases which lead to a potentially increased number of cancelled jobs. Consequently, the total loss in profit will increase. In Figure 3(b), the number of cancelled jobs is equal to zero, regardless of the number of resources. This is because, in our configuration, we generate clients whose jobs are highly business-critical. So the clients always accept the renegotiation requests.

For the three experiments, we conclude that: 1) our algorithm's performance exceeds that of the basic algorithm in terms of profit and the number of cancelled jobs when the clients' jobs are highly business-critical (low expected client utility); and 2) our algorithm's performance tends toward the basic algorithm's performance when the clients have jobs that are less business-critical (when the clients' expected utility is high). Thus in the second case, the clients do not accept a renegotiation, and prefer to enforce the initial SLA.

VI. Related Work

Our work is related to SLA-aware Cloud service provisioning. Most of the existing work proposes an approach aiming to guarantee the agreed upon QoS during the service provisioning process. However, there is less work that considers the consequences of SLA violations, and how the service provisioning should be affected by those violations (e.g., the effect on the provider profit and provider reputation).

In [15], Wu et al. propose a negotiation framework that helps both consumers and providers to define QoS parameters values before service provisioning. The proposed framework includes brokers that assist consumers to find SaaS providers satisfying their needs. The provider cost model does not consider the SLA penalties to be assessed in case of violation.

In order to avoid SLA violations and minimize SLA penalties, it is important to design efficient SaaS scheduling algorithms [1], [2], [3], [4]. In [2], Leitner et al. propose a scheduling algorithm that takes as input the incoming job's execution time requests and the current resource load. That algorithm decides for each request whether to launch a new VM or to schedule it on an existing VM. The objective is to minimize the cost of running VMs and to minimize SLA violations. Despite the fact that the provider revenue depends on the budget given by the requests, the authors do not consider this parameter in the scheduling decision. In the same sense as [2], Liu et al. [1] propose a genetic algorithm that aims to maximize revenue by minimizing the costs of rented VMs. This algorithm divides the user's request into sub-tasks, and then tries to find the optimal combination of VMs able to run those sub-tasks without an SLA violation. Although [1], [2] do not consider the client's budget when scheduling, Wu et al. [3] propose admission control strategies that take into account the budget and the deadline to decide whether to accept or reject the client's request. The main goal is to avoid SLA violation and maximize profit. In [4], Wu et al. propose a scheduling algorithm for enterprise-based SaaS application. The algorithm aims not only to minimize the number of rented VMs, but also to maximize the Customer Satisfaction Level (CSL) by considering the consumer's future interest when scheduling his or her initial request.

The works cited above do not consider what to do after an SLA violation, and once established, the SLA cannot be modified. Our work proposes a renegotiation-based approach to handle possible SLA violations. In contrast to cloud service negotiation for SLA establishment, which is well developed, the subject of renegotiation has not yet been well studied. We are interested in work dealing with renegotiating an already signed SLA, in contrast to [16], which considers renegotiation as negotiating a counter-offer (before the SLA establishment). There is some research work that evokes the idea of SLA renegotiation, but without presenting a concrete contribution on how it could be done [5], [17]. That work focusses on showing the importance of adding renegotiation to the

SLA management life-cycle and presents the requirements for doing so. In [9], [10], the authors propose a conceptual framework for renegotiation. Before renegotiation was introduced to the WS-Agreement protocol by the Grid Resource Allocation Agreement Protocol (GRAAP) group [18], many researchers tried to extend the negotiation component of the WS-Agreement standard in order to support renegotiation [6], [7]. Those authors focus on a renegotiation protocol and propose an approach for extending the WS-Agreement standard in order to support renegotiation.

None of the above-mentioned work proposes a decision-making approach for renegotiation. In [8], Sharaf et al. propose a decision-making strategy based on a fuzzy logic decision support system, as part of the AssessGrid project. The proposed strategy enables the evaluation of an offer received during renegotiation. The authors do not provide details on how the offers are generated during the renegotiation.

To the best of our knowledge, no previous decision-making approach for renegotiation handles a SaaS provisioning procedure wherein the SaaS provider is provisioned by a lower-tier IaaS provider. Our work differs from the work above in that we propose a renegotiation process based on SaaS scheduling information. The proposed renegotiation approach aims not only to minimize the loss in profit due to violation, but also to assure the continuity of service.

VII. Conclusion

For scaling purposes, SaaS providers need to rent resources from IaaS providers in order to run their highly scalable applications. In order to maximize their profit and to satisfy clients, a SaaS provider employs an SLA-aware scheduling algorithm that efficiently assigns client requests to rented resources. Since the cloud environment is highly dynamic, unexpected events may occur that alter the originally selected schedule and lead to SLA violations. Most of the literature assumes that once established, an SLA cannot be modified, and when violated the job is automatically cancelled, without first allowing the provider and consumer the option of renegotiation. The provider pays a high penalty and loses reputation, while the consumer may have a business-critical job cancelled.

We have described an SLA renegotiation-based approach to proactively handle such SLA violations. The resulting decision-making model makes possible a win-win situation (ensuring continuity of service and minimizing SLA penalties costs). The decision-making strategies are based on a utility function for the provider and scheduling information generated by the rescheduling option chosen before renegotiation.

In the future, we plan to investigate further the rescheduling options upon detecting an expected event. The impact of these options will be studied in a large-scale environment using a real world application. Finally, we intend that negotiators will be able to automatically choose an appropriate decision-making negotiation strategy based on the situation.

Acknowledgment

This publication is partially supported by the IDEX "Chaire d'attractivité" program of the Université Fédérale Toulouse Midi-Pyrénées under Grant 2014-345.

References

[1] Z. Liu, S. Wang, Q. Sun, H. Zou, and F. Yang, "Cost-aware cloud service request scheduling for saas providers," *The Computer Journal*, p. bxt009, 2013.

[2] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "Cost-efficient and application sla-aware client side request scheduling in an infrastructure-as-a-service cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 213–220.

[3] L. Wu, S. K. Garg, and R. Buyya, "Sla-based admission control for a software-as-a-service provider in cloud computing environments," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1280–1299, 2012.

[4] L. Wu, S. K. Garg, S. Versteeg, and R. Buyya, "Sla-based resource provisioning for hosted software-as-a-service applications in cloud computing environments," *IEEE Transactions on services computing*, vol. 7, no. 3, pp. 465–485, 2014.

[5] A. F. M. Hani, I. V. Paputungan, and M. F. Hassan, "Renegotiation in service level agreement management for a cloud-based system," *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, p. 51, 2015.

[6] M. Parkin, P. Hasselmeyer, and B. Koller, "An sla re-negotiation protocol," in *Proceedings of the 2nd Non Functional Properties and Service Level Agreements in Service Oriented Computing Workshop (NFPSLA-SOC08), CEUR Workshop Proceedings, ISSN 1613-0073, Volume 411*. Citeseer, 2008.

[7] G. Di Modica, O. Tomarchio, and L. Vita, "Dynamic slas management in service oriented environments," *Journal of Systems and Software*, vol. 82, no. 5, pp. 759–771, 2009.

[8] S. Sharaf and K. Djemame, "Extending ws-agreement to support renegotiation of dynamic grid slas," in *eChallenges e-2010 Conference*. IEEE, 2010, pp. 1–8.

[9] A. F. M. Hani, I. V. Paputungan, and M. F. Hassan, "Service level agreement renegotiation framework for trusted cloud-based system," in *Future Information Technology*. Springer, 2014, pp. 55–61.

[10] W. Mach and E. Schikuta, "A generic negotiation and re-negotiation framework for consumer-provider contracting of web services," in *Proceedings of the 14th International Conference on Information Integration and Web-based Applications & Services*. ACM, 2012, pp. 348–351.

[11] N. R. Jennings, P. Faratin, A. R. Lomuscio, S. Parsons, M. Wooldridge, and C. Sierra, "Automated negotiation: prospects, methods and challenges," *Intern. J. of Group Decision and Negotiation*, vol. 10, no. 2, pp. 199–215, 2001.

[12] X. Zheng, P. Martin, and K. Brohman, "Cloud service negotiation: Concession vs. tradeoff approaches," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 515–522.

[13] Jade Site: Java Agent DEvelopment Framework, http://jade.tilab.com/.

[14] FIPA Interaction Protocols, http://www.fipa.org/repository/ips.php3.

[15] L. Wu, S. K. Garg, R. Buyya, C. Chen, and S. Versteeg, "Automated sla negotiation framework for cloud computing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 235–244.

[16] A. Galati, K. Djemame, M. Fletcher, M. Jessop, M. Weeks, S. Hickinbotham, and J. McAvoy, "Designing an sla protocol with renegotiation to maximize revenues for the cmac platform," in *Web Information Systems Engineering–WISE 2011 and 2012 Workshops*. Springer, 2013, pp. 105–117.

[17] T. B. Quillinan, K. P. Clark, M. Warnier, F. M. Brazier, and O. Rana, "Negotiation and monitoring of service level agreements," in *Grids and Service-Oriented Architectures for Service Level Agreements*. Springer, 2010, pp. 167–176.

[18] O. Waeldrich, D. Battré, F. Brazier, K. Clark, M. Oey, A. Papaspyrou, P. Wieder, and W. Ziegler, "Ws-agreement negotiation version 1.0," in *Open Grid Forum*, vol. 35, 2011, p. 41.