A Robust Game for On Spot Price Cloud Markets

Author: **Massimiliano Riva**

Tutor : Danilo Ardagna

Report
Advanced Programming for Scientific Computing
Professor: Luca Formaggia

Mathematical engineering
Academic year 2019-2020

Abstract

In recent years the evolution and the widespread adoption of virtualization, service-oriented architectures, autonomic, and utility computing have converged letting a new paradigm to emerge: The Cloud Computing. Clouds allow the on-demand delivering of software, hardware, and data as services. As Cloud-based services are more numerous and dynamic, the development of efficient service provisioning policies become increasingly challenging. In this project we take the perspective of Software as a Service (SaaS) providers which host their applications at an Infrastructure as a Service (IaaS) provider. Each SaaS needs to comply with quality of service requirements, specified in Service Level Agreement (SLA) contracts with the end-users, which determine the revenues and penalties on the basis of the achieved performance level. SaaS providers want to maximize their revenues from SLAs, while minimizing the cost of use of resources supplied by the IaaS provider. Moreover, SaaS providers compete and bid for the infrastructural resources. On the other hand, the IaaS wants to maximize the revenues obtained providing virtualized resources.
In this project we implement a C++ program which solve the optimization problem using Gurobi.

Contents

Chapter 1

Introduction

Cloud Computing has been a dominant IT news topic over the past few years. It is essentially a way for IT companies to deliver software/hardware on-demand as services through the Internet. Cloud computing applications are generally priced on a subscription model, so end-users may pay a yearly usage fee, for example, rather than the more familiar model of purchasing software licenses. In the SaaS paradigm applications are available over the Web and provide Quality of Service (QoS) guarantees to end-users. The SaaS provider hosts both the application and the data, hence the end-user is able to use and access the service from all over the world. In IaaS systems, virtual computer environments are provided as services and servers, storage, and network equipment can be outsourced by customers without the expertise to operate them. Many Companies, e.g. Google and Amazon, are offering Cloud computing services such as Google's App Engine and Amazon's Elastic Compute Cloud (EC2) or Microsoft Windows Azure. Large data centers provide the infrastructure behind the Cloud and virtualization technology makes Cloud computing resources more efficient and cost-effective both for providers and customers. Indeed, end-users obtain the benefits of the infrastructure without the need to implement and administer it directly adding or removing capacity almost instantaneously on a "pay-as-you-use" basis. Cloud providers can, on the other hand, maximize the utilization of their physical resources also obtaining economies of scale. The development of efficient service provisioning policies is among the major issues in Cloud research. In this context, game theoretic methods and approaches allows to gain an in-depth analytical understanding of the service provisioning problem. Game Theory has been successfully applied to diverse problems such

as Internet pricing, flow and congestion control, routing, and networking. One of the most widely used "solution concept" in Game Theory is the Nash Equilibrium approach: A set of strategies for the players constitute a Nash Equilibrium if no player can benefit by changing his/her strategy while the other players keep their strategies unchanged or, in other words, every player is playing a best response to the strategy choices of his/her opponents. In this project we take the perspective of SaaS providers which host their applications at an IaaS provider. Each SaaS provider want to maximize its profit while complying with QoS requirements, specified in Service Level Agreement (SLA) contracts with the end-users, which determine the revenues and penalties on the basis of the achieved performance level. The profit of the SaaS is given by the revenues from SLAs minus the cost sustained for using the resources supplied by the IaaS. However, each SaaS behaves selfishly and competes with others SaaS for the use of infrastructural resources supplied by the IaaS. The IaaS, in his turn, wants to maximize the revenues obtained providing the resources. To capture the behavior of SaaSs and IaaS in this conflicting situation in which the best choice for one depends on the choices of the others, we recur to the Generalized Nash Equilibrium (GNE) concept , which is an extension of the classical Nash equilibrium. Moreover we will study the problem as a Robust Game, in this context some parameters are affected from uncertanity since they are obtained by a prediction of future requests and runtime monitoring. We want to keep in consideration these uncertanities and construct a Robust formulation of the problem.

Chapter 2

Problem Statement

We consider SaaS providers exploiting Cloud computing facilities according to the IaaS paradigm to offer multiple transactional applications. The hosted applications can be heterogeneous with respect to resource demands, workload intensities and QoS requirements. The set of applications offered by the j-th SaaS is denoted by $\mathcal{A}_j$, while $\mathcal{S}$ will indicate the set of SaaSs. Each application $a \in \mathcal{A}_j$ offers multiple Web Services WSs (e.g. login, browse, buy, ect), $\mathcal{W}_a$ denotes the set of services supported by application a.
For each WS $w \in \mathcal{W}_a$ an SLA contract is established between he SaaS provider and its end users. This contract specifies SLA levels expressed as a linear utility function which specifies the per request revenue (or penalty) in terms of average response time $R_{a,w}$, the average response time thresholds are denoted by $\bar{R}_{a,w}$.
In the following we will consider the runtime resource provisoning at a single IaaS provider. Applications are hosted in virtual machines (VMs), which are dinamically instantiated by the IaaS provider, and multiple VMs implementing the same application can run in parallel.
IaaS providers usually charge the use of their resources on an hourly basis [**8**]. Hence, each SaaS has to face the problem of determining every hour the optimal number of VMs for each applications in order to minimize costs, performing resource allocation based on a prediction of WS workloads. The SaaS needs also an estimate of the future performance of each VM in order to deteermine application average response time. In the following we models, as a frst approximation, each application as an M/G/1 queue in tamndem with a delay center [**11**], as done in [**14**; **12**; **7**]. We assume (as common among application containers) tht requests are served according t the pro-

cessor sharing scheduling policy. As discussed in [**12**] the delay center allows to model network delays and/or protocol delays introduced in establishing connections, ect. Performance parameters are also continuosly updated at runtime (see [**12**] for further details) in order to capture transient behavior, VMs network and I/O interference [**13**], and perfomance time of the day variability of the considered Cloud provider [**6**]. User sessions begin with a class w WS request arriving from an exogenous source with rate $\tilde{\Lambda}_{a,w}$. The analysis of actual e-commerce site traces (see for example [**10**]) has shown that Cloud application workload usually follows a Poisson distribution, hence we assume that the exogenous arrival streams are Poisson processes. The WS request either returns to the system as a class $w^{'}$ request with a probability $p_{w,w'}$ or it completes with probability $1 - \sum_{l \in \mathcal{W}_a} p_{w,l}$. Let $\Lambda_{a,w}$ denote the aggregate rate of arrivals for class w WS requests, $\Lambda_{a,w} = \sum_{l \in \mathcal{W}_a} \Lambda_{a,l} p_{l,w} + \tilde{\Lambda}_{a,w}$.
Multiple VMs can run in parallel to support the same application. In that case, we suppose that the running VMs are homogeneous in terms of RAM and CPU capacity and the workload is evenly shared among multiple instances (see Figure()), which is common for current Cloud solutions [**9**]. For the IaaS provider we consider a pricing model similar to that of Amazon EC2 [**8**]. The IaaS providers offers:

1. *reserved* (or *on flat*) VMs, for which SaaS providers apply for a one-time payment for each instance they want to reserve

2. *on demand* VMs, which allows SaaS to access computing capacity without long-term commitments

3. *on spot* VMs, spawned by a possible unused IaaS capacity and for which SaaS providers bid and compete

We denote with ρ the time unit cost of reserved VMs. The on spot cost σ, is set by the IaaS provider and fluctuates periodically depending on the IaaS time of the day energy costs and also on the supply and demand from SaaS for on spot VMs [**8**], it could be smaller or larger then ρ. Indeed, SaaS providers compete for the use of on spot VMs specifying the maximum cost $\bar{\sigma}_j$ they are willing to pay per instance hour. The IaaS can also decide not to allocate any on spot instance to a SaaS. Finally we denote with δ the cost for on demand VMs, δ is strictly greater than ρ, and we assume $\bar{\sigma}_j \leq q_j \rho$ for all j, with $q_j < 1$. As a matter of fact, since the IaaS provider can arbitrarly terminate on spot instances from a SaaS resource pool [**8**], no one is willing

to pay for a less reliable resource a time unit cost higher then on demand instances that provide a higher availability level. We denote with η_j the maximum fraction of resources allocated as on spot VMs for SaaS $j \in \mathcal{S}$ to allow a reasonable reliability for every application a. Finally, the number of flat VMs each SaaS provider j is guaranteed to acces to by applying to the on-time payment is denoted by R_j, while the total number of VMs available at the IaaS Cloud service center is denoted by N.

2.1 Robust Game Formulation

The goal of SaaS provider j is to determine every hour the number of reserved r_a, on demand d_a, and on spot s_a VMs to be devoted to the execution of all $\mathcal{A}_j$ applications, in order to minimize the operating costs and, at the same time, to satisfy the prediction $\Lambda_{a,w}$ of the arrival rate of the WS w exposed by application $a \in \mathcal{A}_j$. Let us denote with $\mu_{a,w}$ the maximum service rate for the WS w requests of application a. If the woarkload is evenly shared among the VMs, then the average response time for the execution of WS w requests is given by:

$$E\left[R_{a,w}\right] = D_{a,w} + \frac{\frac{1}{\mu_{a,w}}}{1 - \sum_{l \in \mathcal{W}_a} \frac{X_{a,w}}{\mu_{a,w}(r_a + d_a + s_a)}} \tag{2.1}$$

where $D_{a,w}$ denotes the queueing network delay (see Figure()) and we further assume the VMs are not satured (i.e., the equilibrium conditions for the M/G/1 queues hold, $\sum_{l \in \mathcal{W}_a} \frac{X_{a,w}}{\mu_{a,w}(r_a + d_a + s_a)} < 1$).
$X_{a,w}$ is the throughput (acceptance rate) for web service w of the application a and we have $X_{a,w} \leq \Lambda_{a,w}$, because SaaS providers can decide of accepting or rejecting WS requests in order to minimize costs (trade-off between platform cost and rejection penalties [**7**]), assuming that such decisions are taken according to soome i.i.d. probabilistic law. SaaS providers may incur in penalties $\nu_{a,w} \geq 0$ upon rejection of request executions of w. In order to guarantee a minimum availability, SaaSs have to satisfy a minimum throughput $\lambda_{a,w}$.
It's important to note that, parameters $\mu_{a,w}, D_{a,w}, \Lambda_{a,w}$ are subject to uncertainty since they are obtained by a prediction of future requests and runtime monitoring, so we will assume that the realization of these parameters are

containd in a simmetric interval with respect to a nominal value, one of the main goal of this work is to find a solution that remains feasible also for the worst realization of the parameters.

2.1.1 SaaS problem

As said before the SaaSs have to minimize the cost, hence their objective function is the following:

$$\min \sum_{a \in \mathcal{A}_j} \left[\rho r_a + \delta d_a + \sigma s_a + T \sum_{w \in \mathcal{W}_a} \nu_{a,w} y_{a,w} \right]$$

This cost function includes the fees requested by the IaaS for instances used and the penalties incurred when requests are discarded. Note that σ and s_a are IaaS variables hence we can delete them from the objective function of the SaaS since they are constant for it but we will keep them just to give an idea of the minimization of the overall cost for the SaaS.
Let's start analyzing the constraints, the first can be easily derived from equation 2.1 which ensure that the response time of each WS w provided by application a is lower than the threshold $\bar{R}_{a,w}$ established in the SLA constract, setting $A_{a,w} = \bar{R}_{a,w} - D_{a,w}$ $(A_{a,w} > 0)$ we obtain:

$$r_a + d_a + s_a \geq \left[\frac{A_{a,w}\mu_{a,w}}{A_{a,w}\mu_{a,w} - 1} \right] \sum_{w \in \mathcal{W}_a} \frac{X_{a,w}}{\mu_{a,w}} \qquad \forall a \in \mathcal{A}_j, \forall w \in \mathcal{W}_a \tag{2.2}$$

The second one guarantee that resources are not satured:

$$r_a + d_a + s_a > \sum_{w \in \mathcal{W}_a} \frac{X_{a,w}}{\mu_{a,w}} \qquad \forall a \in \mathcal{A}_j \tag{2.3}$$

The other constraints are:

$$X_{a,w} \geq \lambda_{a,w} \qquad \forall a \in A_j, \forall w \in W_a \tag{2.4}$$

$$y_{a,w} + X_{a,w} \geq \Lambda_{a,w} \qquad \forall a \in A_j, \forall w \in W_a \tag{2.5}$$

$$\sum_{a \in A_j} r_a \leq R_j \tag{2.6}$$

$$\sum_{a \in \mathcal{A}} (r_a + d_a) \leq N - \sum_{a \in \mathcal{A}} s_a \tag{2.7}$$

$$\bar{\sigma}_j \leq q_j \rho \tag{2.8}$$

$$\bar{s}_a \leq \frac{\eta_j}{1 - \eta_j} (r_a + d_a) \tag{2.9}$$

$$r_a, d_a, \bar{s}_a, X_{a,w}, y_{a,w} \geq 0 \qquad \forall a \in A_j \forall w \in W_a \tag{2.10}$$

We observe that $A_{a,w}\mu_{a,w} - 1 = \bar{R}_{a,w} - D_{a,w} - 1 > 0$, indeed $\mu_{a,w} > 0$, $\bar{R}_{a,w} \gg D_{a,w}$ and $\bar{R}_{a,w} \gg \mu_{a,w}$ (i.e., the QoS threshold need to be higher then the queueing network delay $D_{a,w}$ and WS requests service time $\frac{1}{\mu_{a,w}}$). As a consequence, $\frac{A_{a,w}\mu_{a,w}}{A_{a,w}\mu_{a,w}} > 1$ and 2.2 dominates 2.3 which can be dropped from the model. Hence the constraint 2.2 ensure both the response time lower than the threshold and that the resources are not satured.
In the model, the variable $y_{a,w}$ represents the rejection rate for WS w provided by application a 2.5 and depends on the throughput $X_{a,w}$ that, in turn, is bounded from below by $\lambda_{a,w}$ 2.4. However, since in our model the predicted workload is affected by uncertainity, the predicted throughput could turn out to be greater that $\Lambda_{a,w}$ for some realizations. For this reason, we imposed $y_{a,w} \geq 0$ in 2.10.
Equation 2.6 imposes that the sum of the reserved VMs to allocate to all the applications cannot be greater than the quantity established by constract. Constraints 2.9 are introduced for fault tolerance reasons, since the on spot VMs can be terminated by the IaaS at each instant, and guarantee that the on spot instances are at most a fraction $\eta_j < 1$ of the total capacity allocated for application a at IaaS. 2.8 defines an upper bound for the price the SaaS j is willing to pay to the IaaS for on spot VMs. 2.7 entails that allocated VMs are less or equal to the maximum number offered or guaranteed by the IaaS provider, note also that this constraint is a link between different SaaSs.
An important remark is that the variables σ_j and $\bar{s}_a$ are not included in the objective function but they have a fundamental role in the IaaS problem.

Ultimately note that, in the formulation of the problem, r_a, d_a and $\bar{s}_a$ are imposed to be greater or equal to zero 2.10, but not integer, as in reality they are. Infact, requiring variables to be integer makes the solution much more difficult (NP-hard), but if the optimal values of the variables are fractional and they are rounded to the closest integer solution, the gap between the solution of the real integer problem and the relaxed one is very small. The notation adopted is summarized in Table 2.1.

2.1.2 Robust SaaS Problem

Setting $B^l_{a,w} = \frac{A_{a,w}\mu_{a,w}}{(1-A_{a,w}\mu_{a,w})\mu_{a,l}}$ and dropping the dominated constraints the problem becomes:

$$\min \sum_{a\in\mathcal{A}_j} \left[\rho r_a + \delta d_a + \sigma s_a + T \sum_{w\in\mathcal{W}_a} \nu_{a,w} y_{a,w} \right] \tag{2.11}$$

subject to

$$r_a + d_a + \sum_{l\in\mathcal{W}_a} B^l_{a,w} X_{a,l} \geq -s_a \qquad \forall a \in \mathcal{A}_j, \forall w \in \mathcal{W}_a \tag{2.12}$$

$$X_{a,w} \geq \lambda_{a,w} \qquad \forall a \in A_j, \forall w \in W_a \tag{2.13}$$

$$y_{a,w} + X_{a,w} \geq \Lambda_{a,w} \qquad \forall a \in A_j, \forall w \in W_a \tag{2.14}$$

$$\sum_{a\in A_j} r_a \leq R_j \tag{2.15}$$

$$\sum_{a\in\mathcal{A}} (r_a + d_a) \leq N - \sum_{a\in\mathcal{A}} s_a \tag{2.16}$$

$$\bar{\sigma}_j \leq q_j \rho \tag{2.17}$$

$$\bar{s}_a \leq \frac{\eta_j}{1-\eta_j} (r_a + d_a) \tag{2.18}$$

$$r_a, d_a, \bar{s}_a, X_{a,w}, y_{a,w} \geq 0 \qquad \forall a \in A_j \forall w \in W_a \tag{2.19}$$

Parameters $\mathbf{B} = (B^l_{a,w})$ are uncertain and we want to compute a single stage robust solution, that is, a solution that is feasible and can be used for any realization of the uncertainity.

Also $\mathbf{\Lambda}$ is affected by uncertainity but this affects only the right-hand-sides, according to 2.14, we can consider a formulation in which $\mathbf{\Lambda}$ is not uncertain and takes value $\Lambda^*_{a,w} = \max_{\Lambda_{a,w} \in P(\Lambda)} \Lambda_{a,w}$, where $P(\Lambda)$ is the uncertainity set describing the possible realizations for $\Lambda_{a,w}$. Therefore, we reduce to a problem where only $\mathbf{B}$ is uncertain. We assume that each uncertain parameter $B^l_{a,w}$ takes value in a symmetric intervall $[\bar{B}^l_{a,w} - \hat{B}^l_{a,w}, \bar{B}^l_{a,w} + \hat{B}^l_{a,w}]$ defined by a maximum deviation $\hat{B}^l_{a,w}$ from a nominal value $\bar{B}^l_{a,w}$ and at most Γ parameters can deviate from the nominal value at the same time.
Let $P(B)$ be the uncertainity set, we want that 2.12 was satisfied $\forall B \in P(B)$, i.e. :

$$r_a + d_a + \sum_{l \in \mathcal{W}_a} B^l_{a,w} X_{a,l} \geq -s_a \qquad \forall B \in P(B) \tag{2.20}$$

In this way, even just considering the vertices of P(B), we would end up with an exponential number of constraints. Otherwise we can rewrite the constraint in such a way that it is satisfied for the worst realization B^*, that is the one which minimize the sum:

$$r_a + d_a + \sum_{l \in \mathcal{W}_a} \bar{B}^l_{a,w} X_{a,l} + \min_{\mathcal{S} \subseteq \mathcal{W}_a : |\mathcal{S}| \leq \Gamma} \left\{ \sum_{l \in \mathcal{S}} (-\hat{B}^l_{a,w}) X_{a,l} \right\} \geq -s_a \tag{2.21}$$

Due to our definition of uncertainity set, the value of B can be decomposed in a nominal part (the ones that contributes to first sum) and an uncertain part (the second sum) that has to be minimized.
If we fix $X = \bar{X}$ the minimization problem can be written as the following:

$$\begin{aligned}
&P && \min \sum_{l \in \mathcal{W}_a} (-\hat{B}^l_{a,w}) X_{a,l} z^l_{a,w} && \\
&(\alpha_{a,w}) && \sum_{l \in \mathcal{W}_a} z^l_{a,w} \leq \Gamma && \\
&(\beta^l_{a,w}) && 0 \leq z^l_{a,w} \leq 1 && \forall l \in \mathcal{W}_a
\end{aligned}$$

P is well posed because of the negativity of B, hence the solution is not the trivial one corresponding to $z = 0$. Subsituting the minimization in 2.21 with

this versione we obtain:

$$
\begin{aligned}
& r_a + d_a + \sum_{l \in \mathcal{W}_a} \bar{B}^l_{a,w} X_{a,l} + \min \sum_{l \in \mathcal{W}_a} (-\hat{B}^l_{a,w}) X_{a,l} z^l_{a,w} \geq -s_a \\
& \sum_{l \in \mathcal{W}_a} z^l_{a,w} \leq \Gamma \\
& 0 \leq z^l_{a,w} \leq 1 \qquad \forall l \in \mathcal{W}_a
\end{aligned}
$$

Unfortunately we cannot simply delete the minimization, because the min is not redoundant due to the discordance between the min and $\geq$.
We have to find another way to delete it, we can consider the dual formulation of the problem:

$$
\begin{aligned}
& \max \left[-\alpha_{a,w} \Gamma - \sum_{l \in \mathcal{W}_a} \beta^l_{a,w} \right] \\
& \alpha_{a,w} + \beta^l_{a,w} \geq \hat{B}^l_{a,w} X_{a,l} && \forall l \in \mathcal{W}_a \\
& \alpha_{a,w} \geq 0 \\
& \beta^l_{a,w} \geq 0 && \forall l \in \mathcal{W}_a
\end{aligned}
$$

Plugging in the new objective function in the initial constraint 2.21, we obtain:

$$
\begin{aligned}
& r_a + d_a + \sum_{l \in \mathcal{W}_a} \bar{B}^l_{a,w} X_{a,l} + \max \left[-\alpha_{a,w} \Gamma - \sum_{l \in \mathcal{W}_a} \beta^l_{a,w} \right] \geq -s_a \\
& \alpha_{a,w} + \beta^l_{a,w} \geq \hat{B}^l_{a,w} X_{a,l} && \forall l \in \mathcal{W}_a \\
& \alpha_{a,w} \geq 0 \\
& \beta^l_{a,w} \geq 0 && \forall l \in \mathcal{W}_a
\end{aligned}
$$

We observe that the max is redundant due to the $\geq$. Hence we can drop the max since the solver will choose the realization which maximizes the dual objective function, in order to satisfy the constraint. Finally the robust version of 2.12 becomes:

$$r_a + d_a + \sum_{l \in \mathcal{W}_a} \bar{B}^l_{a,w} X_{a,l} - \alpha_{a,w}\Gamma - \sum_{l \in \mathcal{W}_a} \beta^l_{a,w} \geq -s_a \tag{2.22}$$

$$\alpha_{a,w} + \beta^l_{a,w} \geq \hat{B}^l_{a,w} X_{a,l} \qquad \forall l \in \mathcal{W}_a \tag{2.23}$$

$$\alpha_{a,w} \geq 0 \tag{2.24}$$

$$\beta^l_{a,w} \geq 0 \qquad \forall l \in \mathcal{W}_a \tag{2.25}$$

Following the discussion of the previous section, the problem that SaaS j has to periodically solve can be formulated as follows:

$$\min_{X,y,r,d,\alpha,\beta} \sum_{a \in A_j} \left[\rho r_a + \delta d_a + T \sum_{w \in W_a} \nu_{a,w} y_{a,w} \right] \tag{2.26}$$

subject to

$$r_a + d_a + \sum_{l \in W_a} \bar{B}^l_{a,w} X_{a,l} - \Gamma \alpha_{a,w} - \sum_{l \in W_a} \beta^l_{a,w} \geq -s_a \qquad \forall a \in A_j, \forall w \in W_a \tag{2.27}$$

$$\alpha_{a,w} + \beta^l_{a,w} \leq \hat{B}^l_{a,w} X_{a,l} \qquad \forall a \in A_j, \forall w, l \in W_a \tag{2.28}$$

$$\alpha_{a,w}, \beta^l_{a,w} \geq 0 \qquad \forall a \in A_j, \forall w, l \in W_a \tag{2.29}$$

$$X_{a,w} \geq \lambda_{a,w} \qquad \forall a \in A_j, \forall w \in W_a \tag{2.30}$$

$$y_{a,w} + X_{a,w} \geq \Lambda_{a,w} \qquad \forall a \in A_j, \forall w \in W_a \tag{2.31}$$

$$\sum_{a \in A_j} r_a \leq R_j \tag{2.32}$$

$$\sum_{a \in \mathcal{A}} (r_a + d_a) \leq N - \sum_{a \in \mathcal{A}} s_a \tag{2.33}$$

$$\bar{\sigma}_j \leq q_j \rho \tag{2.34}$$

$$\bar{s}_a \leq \frac{\eta_j}{1 - \eta_j} (r_a + d_a) \tag{2.35}$$

$$r_a, d_a, \bar{s}_a, X_{a,w}, y_{a,w} \geq 0 \qquad \forall a \in A_j \forall w \in W_a \tag{2.36}$$

$$\tag{2.37}$$

2.2 IaaS Problem

The IaaS has to solve the following optimization problem:

$$\max_{s,z,\sigma} \sum_{a \in A} (\sigma - \omega) s_a \tag{2.38}$$

subject to

$$\sum_{a \in \mathcal{A}} s_a \leq N - \sum_{a \in \mathcal{A}} (r_a + d_a), \tag{2.39}$$

$$\sigma \geq \omega, \tag{2.40}$$

$$\sigma - \bar{\sigma}_j \leq M(1 - z_j), \qquad \forall j \in \mathcal{S} \tag{2.41}$$

$$\bar{\sigma}_j - \sigma \leq M z_j, \qquad \forall j \in \mathcal{S} \tag{2.42}$$

$$s_a \leq \bar{s}_a z_j, \qquad \forall j \in \mathcal{S}, \forall a \in \mathcal{A}_j \tag{2.43}$$

$$s_a \geq 0, \qquad \forall a \in \mathcal{A} \tag{2.44}$$

$$z_j \in \{0, 1\}, \qquad \forall j \in \mathcal{S} \tag{2.45}$$

The IaaS has to maximizes the revenue of the on spot market, it has to periodically set the price for on spot VMs σ taking into account the energy cost ω 2.38, moreover it has to decide how many on spot VMs allocate for the execution of the application a (s_a). 2.39 entails that total number of on spot resources allocated does not exceed the number of avalilable VMs. Equation 2.40 guarantees to the IaaS to have a revenue (the on spot price bigger or equal to the energy cost). 2.41, 2.42 and 2.43 impose that the on spot VMs are given to SaaS j if and only if they can pay at least σ for them. Of course the on spot VMs given for an application a (s_a) has to be less then the number of desired on spot VMs ($\bar{s}_a$) for its execution 2.43. The notation adopted is summarized in Table 2.1.
As you can see, this problem is quadratic 2.38, therefore we used an exact method to solve it instead of Gurobi, as in [**3**], the following algorithm has been implemented:

Algorithm 1 Algorithm (Solving the IaaS optimization problem)

1: Sort $\bar{\sigma}_j$ in decreasing order
2: $S = N - \sum_{a \in \mathcal{A}}(r_a + d_a)$
3: $\sigma = \infty, \mathcal{R}^* = 0, t = 1$
4: $s = \min\left\{S, \sum_{1 \leq j \leq t} \sum_{a \in \mathcal{A}_j} \bar{s}_{a,j}\right\}$, $\mathcal{R} = \bar{\sigma}_t$
5: **if** $\mathcal{R} > \mathcal{R}^*$ **then**
6: $\quad \sigma = \bar{\sigma}_t$, $\mathcal{R}^* = \mathcal{R}$
7: **if** $s < S$ and $t < |\mathcal{S}|$ **then**
8: $\quad$ t= t+1, *go to step 4*
9: **for** $j \in \mathcal{S}$ **do**
10: $\quad$ **if** $\bar{\sigma}_j \geq \sigma$ **then**
11: $\quad\quad z_j = 1$
12: $\quad\quad$ **for** $a \in \mathcal{A}_j$ **do**
13: $\quad\quad\quad s_a = \min\{\bar{s}_a, S\}$, $S = S - s_a$
14: $\quad$ **else**
15: $\quad\quad z_j = 0$
16: $\quad\quad s_a = 0 \quad \forall a \in \mathcal{A}_j$

The algorithm starts sorting the prices $\bar{\sigma}_j$ offered by the SaaSs and determines the number S of on spot VMs that the IaaS can offer (step 1 and 2). During the excution of the algorithm σ represents the current best price for the IaaS, $\mathcal{R}^*$ the corresponding revenue and t is an index to iterate among the SaaSs. These three values are initializedat step 3. Steps 4-8 find the optimal cost σ. The IaaS determines if there is enough capacity to provide to the first t SaaSs the on spot VMs at the price $\bar{\sigma}_t$ offered by the SaaS t.The maximum number s of on spot VMs that the IaaS can sell to the first t SaaSs is the minimum between the available capacity S and the total number of on spot VMs requested by the first t SaaSs; that is, $\sum_{1 \leq j \leq t} \sum_{a \in \mathcal{A}_j} \bar{s}_a$. The corresponding revenue is $\mathcal{R} = \bar{\sigma}_t s$ (step 4). Then in steps 5 and 6, σ and $\mathcal{R}^*$ are updated accordingly. If the IaaS capacity is not satured ($s < S$) and there are still SaaS providers to consider ($t < |\mathcal{S}|$), then steps 4-6 are repeated; otherwise σ is the optimal cost (steps 7 and 8). The optimal values of z_j and s_a are assigned in steps 9-16 according to the price offered by SaaS providers and their WS applications requirementes.
Notice that the time complexity of Algorithm 1 is $\mathcal{O}(\max\{|\mathcal{S}|log(|\mathcal{S}|), |\mathcal{A}|\})$ because of the sorting at step 1 and the assignment of s_a at step 13.

System Parameters

$\mathcal{S}$	Set of SaaS providers
$\mathcal{A}$	Set of applications of all the SaaS providers
$\mathcal{A}_j$	Set of applications of the SaaS provider j
$\mathcal{W}_a$	Set of WSs provided by application a
$\Lambda_{a,w}$	Overall prediction of the arrival rate for WS w of application a
$\lambda_{a,w}$	Minimum arrival rate to be guaranteed for WS class w of application a
$\mu_{a,w}$	Maximum service rate for executing WS class w of application a
$D_{a,w}$	Queueing delay for executing WS class w of application a
$\bar{R}_{a,w}$	WS w average response time threshold
$\nu_{a,w}$	Penalty for rejecting a single WS w request for application a
ρ	Time unit cost for reserved VMs
δ	Time unit cost for on demand VMs
q_j	Maximum fraction of reserved VMs price for on spot VMs price SaaS provider j is willing to pay
ω	VM time unit energy cost for IaaS provider
η_j	Maximum fraction of total resources allocated as on spot VMs for SaaS provider j
N	Maximum number of VMs that can be executed at the IaaS
R_j	Maximum number of reserved VMs that can be executed fot the SaaS j
T	Control time horizon

SaaS Decision Variables

r_a	Number of reserved VMs used for application a
d_a	Number of on demand VMs used for application a
$\bar{s}_a$	Number of desired on spot VMs for application a
$X_{a,w}$	Throughput for WS w requests of application a
$\bar{\sigma}_j$	Time unit cost threshold for SaaS j for on spot VM instances

IaaS Decision Variables

s_a	Number of on spot VMs used for application a
σ	Time unit cost offered for on spot VM instances

Table 2.1: Parameters

Chapter 3

Implementation

3.1 Gurobi Installation

You will need a license in order to install and use the Gurobi Optimizer, if you are an academic user, you can obtain a free academic license on `www.gurobi.com`, then go to the download page, find your platform (we'll assume Linux in this document), and choose the correspondig file to download.
Your next step is to choose a destionation directory, we reccomend **/opt**, but other directories will work as well. Copy the Gurobi distribution to the destination file directory and extract the contents typing the follwing:

```
$ tar xvfz gurobi9.0.1_linux64.tar.gz
```

This command will create a sub-directory **/opt/gurobi901/linux64** that contains the complete Gurobi distribution.
Your <**installdir**> will be **/opt/gurobi901/linux64**, the Gurobi Optimizer makes use of several executable files. In order to allow these files to be found when needed, you will have to modify a few environment variables:

- GUROBI_HOME should point to your <**installdir**>
- PATH should be extended to include <**installdir**>/**bin**
- LD_LIBRARY_PATH should be extended to include <**installdir**>/**lib**

If you are **bash** shell users you should add the following lines to the **.bashrc** file:

```
export GUROBI_HOME="/opt/gurobi901/linux64"
export PATH="${PATH}:${GUROBI_HOME}/bin"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${GUROBI_HOME}/
 lib"
```

Then procede typing the following:

```
$ cd /opt/gurobi901/linux64/src/build
$ sudo make
```

The static library **libgurobi_c++.a** is now created, you have to copy it in **/opt/gurobi901/linux64/lib** and then you have done.

3.1.1 Compiling

For the compilation we developed a makefile, in order to use it, you have to modify all the lines that contain the address of the library you just created with the address of the location of your library.

3.2 Code Structure

In this section we will describe the code and the classes designed and developed in order to solve the problems introduced in the previous chapter.
Before starting we just explain the general functionalities of the code, we can run the program in three different ways passing different types of parameters from the command line :

- ./exe g config_file.csv
- ./exe r config_file.csv
- ./exe e config_file.csv global_parameters.csv SaaSs_parameters.csv WSs_parameters.csv

Using the first option we will only generate (g = generate) the parameters of the problem which will be stored in the files used in the third option, we will talk about this files talking about *Set_System class*. The second option generates randomly the parameters and solves the problem using them (r = random). Finally the third option uses the values in *global_parameters.csv SaaSs_parameters.csv WSs_parameters.csv* to fill the data structures and then it solves the problem. All the option need the *config_file.csv* which contains the information for the size of the data structures (n° of SaaSs, n° of apps per SaaS and n° of WSs per app).

3.2.1 WS class

This is the smallest class designed, it represent a web service object hence it contains the main informations of the WS, i.e. $\Lambda_{a,w}, \lambda_{a,w}, D_{a,w}, \mu_{a,w}, \bar{R}_{a,w}, \nu_{a,w}$.

3.2.2 Application class

This class represent an application, it contains a vector of web services (WS objects)

3.2.3 SaaS class

The SaaS class starts to become bigger, indeed it contains a vector of applications objects which represents the apps provided by the SaaS, and also it keeps track of the results fo the SaaS problem. The class contains four vectors of double, initialized as empty, which will be filled when the SaaS Problem will be solved. Each cell of the vectors contains the number of VMs needed for the corresponding application in the application vector. The vectors are:

1. *on_flat*: it contains the number of on flat VMs needed for each application (r_a)

2. *on_demand*: it contains the number of on demand VMs needed for each application (d_a)

3. *on_spot*: it contains the number of on spot VMs needed for each application (s_a)

4. *desired_on_spot*: it contains the number of desired on spot VMs desired by the SaaS for the application a ($\bar{s}_a$)

Finally the class contains also the *cost_threshold* ($\bar{\sigma}_j$).

3.2.4 IaaS Class

The IaaS class represents the concept of IaaS, it stores just two information regarding the IaaS variables (see Table 2.1), i.e. the time unit cost offered for on spot VM instances (σ) and the number of on spot VMs used for application a (s_a).

3.2.5 SaaS_Problem Class

This is the first big class of the code structure, here we use the Gurobi solver to solve the Robust SaaS problem. It is composed by a shared pointer to a SaaS, all the Problem parameters and a vector of double (*cost_thresholds*) in which we store σ_j at each iteration in order to keep track of the changes in the offer.
Regarding the methods implemented, the class has two private functions to compute $\bar{B}$ and $\hat{B}$ starting from the parameters, then we find the constructors in which we also set the first σ_j, the *check* method that check if the request of the SaaS in terms of on_spot VMs have been satisfied, the *rounding* method which is needed after solving the SaaS Problem because $r_a, d_a, \bar{s}_a$ are double but since they represent the number of VMs they have to be integer so we need to perform a rounding keeping attention to not violate the problem constraints.
Finally the most important method is *solve()* which use the Gurobi solver to compute the solution of the Robust SaaS Problem. This function starts creating a Gurobi environment, an empty model and setting the objective function equal to zero:

```
GRBEnv env = GRBEnv(); // create a gurobi environment

GRBModel model = GRBModel(env); //create an empty model

GRBLinExpr obj = 0.0; // create a linear expression to construct my objective function
```

Then we procede starting creating all the variables ($GRBVar$), first we create some $GRBLinExpr$ which are linear expression of $GRBVar$ in order to implement 2.6, 2.7 and to create the parameters $\bar{B}$ and $\hat{B}$ of 2.27, 2.28:

```
GRBLinExpr total_on_flat = 0;   // create a linear expression which will be the sum of all on_flat VMs
GRBLinExpr total_VM = 0;        // create a linear expression which will be the sum of all VMs
std::vector<GRBVar> x_v;        // here we store the variables X_a_w because we need them for the computation of the parameters B
```

We procede with a for cycle over all the application of the SaaS where we create r_a, d_a, we update the objective function and the expression created before:

```
GRBVar r  = model.addVar(0.0, GRB_INFINITY, 0.0  , GRB_CONTINUOUS, r_a);      // creating r_a
GRBVar d  = model.addVar(0.0, GRB_INFINITY, 0.0  , GRB_CONTINUOUS, d_a);      // creating d_a

obj += rho * r;                                                               // adding r_a to the objecting funcion
obj += delta * d;                                                             // adding d_a to the objecting funcion

total_on_flat += r;                                                           // updating the sum of all the on flat VMs
total_VM += ( r + d );                                                        //updating the sum of all the VMs
```

In the following the constraints regarding all the web services of considered application are imposed, exept for the robust constraints 2.27, 2.28 :

```
for(unsigned j = 0; j < app.get_size(); j++)                                  // cicle over all the WSs of the application i
{

  auto web_service = app.get_web_services()[j];                               // j-th WS of the i-th application

  GRBVar y = model.addVar(0.0, GRB_INFINITY, 0.0, GRB_CONTINUOUS);            // creating y_a_w

  obj += (T * web_service.get_nu_a_w() * y);                                  // adding it to the objecting funcion

  GRBVar x = model.addVar(0.0, GRB_INFINITY, 0.0  , GRB_CONTINUOUS);          // creating X_a_w

  model.addConstr( x >= web_service.get_lambda_a_w() );                       // X_a_w >= lambda_a_w
  model.addConstr( y + x >= web_service.get_LAMBDA_a_w() );                   // X_a_w + y_a_w >= LAMBDA_a_w

}
```

In order to set the robust constraints we need to end the cycle and start another one, because we need the sum of all the $X_{a,w}$ for the equation 2.27 :

```
for(unsigned j = 0; j < app.get_size(); j++)                    // cicle over all the WSs of the application i
{

  auto web_service = app.get_web_services()[j];                 // j-th WS of the i-th application

  GRBLinExpr BX = 0.0;                                          // linear expression = sum of ( B_bar_a_w_l * X_a_l )

  auto  B_a_w = set_param_B(web_service, app);                   // computing the parameters B_bar and B_hat using the function set_param_B

  GRBVar alpha = model.addVar(0.0, GRB_INFINITY, 0.0  , GRB_CONTINUOUS);   // creating alpha_a_w

  GRBLinExpr betas = 0.0;                                       // linear expression = sum of beta_a_w_l

  for(unsigned l = 0; l < B_a_w.size(); l++)                    // cicle over WS l of the application i
  {

    GRBVar beta = model.addVar(0.0, GRB_INFINITY, 0.0  , GRB_CONTINUOUS);   // creating beta_a_w_l
    betas += beta;                                              // computng the sum of all betas


    model.addConstr( alpha + beta >= B_a_w[l].second * x_v[l] );  // second robust constraint

    BX += B_a_w[l].first * x_v[l];                              // computing B_bar_a_w_l * X_a_l

  }

  model.addConstr( - r - d - s -> get_on_spot(app) - BX + gamma*alpha + betas <= 0 );    //first robust constraint

}
```

At the end of the cycle over all the applications of the SaaS, we set the last costraints, the first one is simply the equation 2.6 (where we use the *GRBLinExpr* created at the beginning), the other one implements the equation 2.7 partially, i.e. the sum is over all the application of the SaaS ($\mathcal{A}_j$) and not over all the application of the problem ($\mathcal{A}$), so this constraint guarantee that each SaaS will not exceed the total number N of available VMs at the IaaS. In the class Game we check that the sum of all the VMs requested from all the SaaSs does not exceed N, otherwise the program will print the error message *"Insufficient Resources"*.

```
model.addConstr( total_on_flat <= R_j);                         // sum of r_a less or equal to R_j_bar
model.addConstr(total_VM <= N - total_on_spot );                //  the sum the on flat and on demand VMs lessor equal to N - on_spot VM
```

Once all the variables are created and the constraits are added to the model, we set objective function and we perform the optimization, finally we will simply store the results.

3.2.6 IaaS_Problem class

The IaaS_Problem class contains an IaaS object and all the parameter for the IaaS Problem, in particular the most important thing is the *solve_greedy()*

```
model.setObjective(obj, GRB_MINIMIZE);                    // set the objective function

model.optimize();                                         // optimize the model
```

function in which the algorithm 1 is implemented, the algorithm simply sort the SaaSs with respect to *cost_threshold* (σ_j) and starting from the first one it computes how much is the revenue of the IaaS, then it checks if using the threshold of the other SaaSs the revenue would be more, in the positive case it changes the *final_sigma* (σ) and updates the *given_on_spot* VMs accordingly.

3.2.7 Set_System class

This class is used to initialize all the parameters of the problem and to build all the structure needed. We have two options, both start with a *config_file.csv* which is passed as input to the constructor. This file contains the number of SaaSs, the number of apps per SaaS, the number of WSs per app and an additional value *time* which will be discussed later on. The first option, calling the function *set()*, creates all the parameters randomly usign a seed, the second one, calling the function *set_from_files()*, takes as additional inputs three csv files *global_parameters.csv, SaaSs_parameters.csv, WSs_parameters.csv* which contain all the parameters of the problem, the function only fills the data structures usign these values, this allow us to set the parameters manually. In the following we can see better the content of the files:

1. *global_parameters.csv*: $T, N, \Gamma, \rho, \delta, \hat{\mu}, \hat{D}, \omega$

2. *SaaSs_parameters.csv*: $q_j, \eta_j, R_j \quad \forall j \in \mathcal{S}$

3. *WSs_parameters*: $\Lambda_{a,w}, \lambda_{a,w}, D_{a,w}, \mu_{a,w}, \bar{R}_{a,w}, \nu_{a,w} \quad \forall w \in \mathcal{W}_a, \forall a \in \mathcal{A}_j$

3.2.8 Game class

The main class of the problem is the Game class, here we create an object in which we have a vector of shared pointers to all the SaaS_Problems con-

sidered and also a IaaS_Problem object.
We have two different ways to intialize a Game object, as said at the beginning of this chapter we have three ways to run the code, using the first option we pass to the constructor only the *config_file.csv* and a *seed* then the constructor creates a Set_System object and usign its function *set()* it creates all the parameters, then it stores a shared pointers the IaaS_Problem object and a vector of SaaS_Problem objects just created, finally it calls its function *create_files()* which creates *global_parameters.csv, SaaSs_parameters.csv, WSs_parameters.csv* and stores all the parameters in them. The most important function is *solve()*, this function solves the entire problem once the parameters are stored in the data structures. The algorithm starts solving the SaaS_Problem for each SaaS (SaaS_Problem.solve()) then it calls the function *rounding()* of the SaaS_Problem class, this function only transform the number of virtual machines from double to integers values in a smart way. At this point the IaaS_Problem is solved for the first time (IaaS_Problem.solve_greedy()), the IaaS assign to each SaaS a number of on spot VMs per app (this number is stored in the *given_on_spot* vector of each SaaS), then a SaaS has two option, if its request of on spot VMs has been satisfied it doesn't do anything otherwise it can increase its *cost_threshold* (σ_j) in order to give a new offer to the IaaS, this can be done a maximum number of times since the maximum σ_j is controlled by q_j. The algorithm controls if not all the SaaSs have been satisfied and if at least one changed its *cost_threshold*, if this condition holds it solves the IaaS Problem again until either all the SaaSs are statisfied or the unsatisfied SaaSs have reached their maximum σ_j. Now all the SaaSs have the final number of *given_on_spot*, therefore they store them as *on_spot* VMs (SaaS_Problem.update()) and solve the last time the SaaS Problem in order to find the exact number of *on_flat* and *on_demand* VMs then they call again *rounding()*. Finally the results are printed in *results.txt*.

We will conduct three different analysis, the first one regards the robustness of the problem, we will consider only one SaaS (10 apps and 5 WSs per app) and we will look how the number of VMs varies with different value of Γ, the second analysis is a scalability analysis, we implemented a parallel version of the problem and we will look for the speed up :

parallel Game.solve()

The parallel version of the code is very simple, the only change is the *solve()* method of the Game class. Instead of solving all the SaaS Problems sequentially, each core solves some SaaS problems using a round robin for the splitting of the SaaSs vector, then they send the results $(s_a, d_a, \bar{s}_a \ \forall a \in \mathcal{A}_j)$ to the $rank_0$ which solves the IaaS problem, finding the final σ and the number of on spot VM intended to each SaaS (*given_ on_ spot*), then $rank_0$ sends to the others ranks the number of *given_ on_ spot* of their SaaSs, they update the results storing the *given_ on_ spot* VMs as *on_ spot* VMs, they solve again the SaaS Problem in order to find the exact number of on flat and on demand VMs. Finally the ranks send the final results to the $rank_0$ which prints them.

3.2.9 run.py, setup.py

In the third analysis we observe the trend of the number of VMs during a day in which Λ_a follows some specific traces (we will talk about details in the next chapter). For this purpose we implemented some python scripts in order to setup the environment e run the instances automatically. There are two python files, the first one is *setup.py* which creates in the folder *instances* 24 subfolders (one per hour of the day), then it copies the *config_file.csv* in each subfolder and it modifies its cell *time*, this last value multiplies Λ_a forcing it to follow some specific traces, we will go depth into this analysis later.
Then we have *run.py* which just run the code for each of the 24 instances, first it generates the parameters randomly using the *config_file.csv* specific of the folder. Finally it runs the code usign the parameters and stores the results (in order to use this script you have to change the variable *path* with the path to the folder in your computer).

Chapter 4

Numerical Analysis

4.1 Parameters Initialization

As we mention before, we can initialize our parameters from files or randomly, but we need to initialize them randomly at least once, the first time we lunch the program. In order to do this, we pass a *seed* to the Game class constructor which create a Set_System class object and calls its function set. This function exploits the seed to initialize all the parameters and set the problem instances. Let is talk about parameters of web services (WS), for the moment leave out $\Lambda_{a,w}$ (and also $\lambda_{a,w}$) we will return on this topic later. As done in **[5]** $\mu_{a,w}$ is selected randomly between 200 and 400 requests per second and $D_{a,w} \in [0.001, 0.05] sec.$
$\nu_{a,w}$ is set to 0.0000045 $ per rejected request, finally in order to have the QoS threshold higher than the queueing network delay $D_{a,w}$ and WS requests service time $\dfrac{1}{\mu_{a,w}}$ we set $\bar{R}_{a,w} = \frac{3}{2}\left(\frac{1}{\mu_{a,w}} + D_{a,w}\right)$.
Regarding $\Lambda_{a,w}$ we initialize it in a particular way, we search $\Lambda_{a,w}$ in order to saturate K machines (with K random integer number selected between 1 and 2), i.e. :

$$\sum_{w \in \mathcal{W}_a} \frac{\Lambda_{a,w}}{\mu_{a,w}} = K \tag{4.1}$$

Then we define an overall prediction of the arrival rate (Λ_a) for the whole application such that $\Lambda_{a,w} = \Lambda_a p_w \ \forall w \in \mathcal{W}_a$ and $\sum_{w \in \mathcal{W}_a} p_w = 1$.

Finally we compute Λ_a in the following way:

$$\Lambda_a \sum_{w \in \mathcal{W}_a} \frac{p_w}{\mu_{a,w}} = K \tag{4.2}$$

$$\Lambda_a = \frac{K}{\sum_{w \in \mathcal{W}_a} \frac{p_w}{\mu_{a,w}}} \tag{4.3}$$

Hence setting randomly p_w we can easily compute $\Lambda_{a,w}$. Moreover in the third analysis we performed we look for the trend of the number of the VMs during a day, with this purpose the file *setup.py* creates in the *instances* folder, the subfolders for the parameters and the results for each hours of the day. During this period we force Λ_a to follow the following traces:

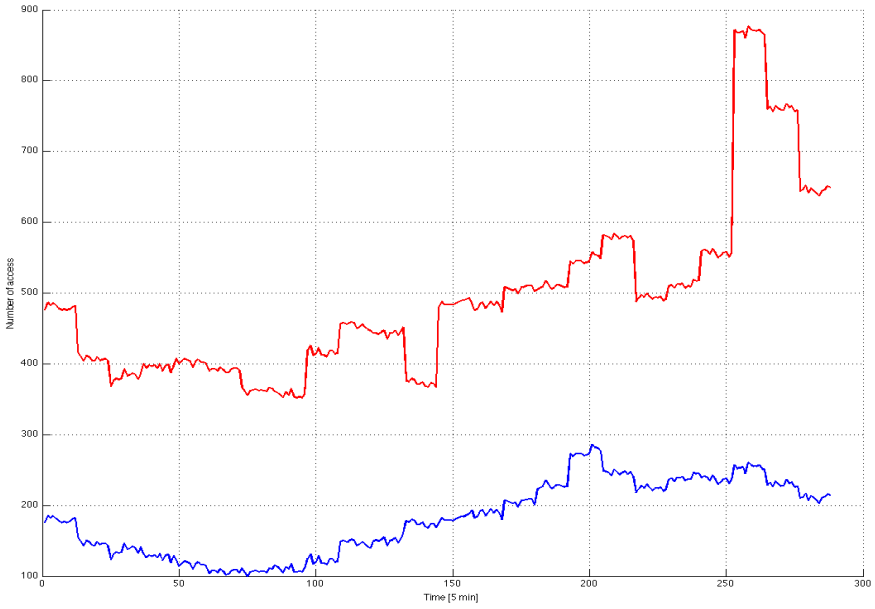

Figure 4.1

This is done multiplying Λ_a by a variable *time*, which corresponds to the proportion of the traffic in that specific hour with respect to the maximum

arrival rate, the variable time is contained in the *config_file.csv* specific of the subforlder. The red trace represents a spiky day and the blue trace a more smooth one. Finally we set $\lambda_{a,w} = 0.8\Lambda_{a,w}$ as in [**1**].The value of the WSs parameters are collected in *WSs_parameters.csv*.
For what concerns the SaaS specific parameters we set $R_j = \frac{1}{3}\frac{N}{Number\ of\ SaaSs}$ where N indicates the total number of VMs which can be bought at the IaaS. q_j is set randomly between 25% and 90% and $\eta_j = 0.25$, all the values are stored in *SaaSs_parameters.csv*.
Finally let's talk about the global parameters, the total number of available VMs at the IaaS (N) is setted to 10000 and the time horizon T to 3600 seconds (i.e. 1 hour), Γ by default we initialize it to 1, ρ is selected randomly between 0.015 and 4.00, $\delta = 4\rho$ and $\omega = \frac{1}{2}\sigma_{\min}$ where $\sigma_{\min}$ is the minimum of the cost thresholds of the SaaSs in order to allow each SaaS to receive some on spot VMs.
Finally we random initialize the maximum deviation from the nominal value of the arrival rate and the delay ($\hat{\mu}$ and $\hat{D}$) using as lower and upper bound the minimum and the maximum of the means of the estimates computed in [**4**].

4.1.1 Scalability Analysis

We performed a scalability analysis, in this section we analyze the effect of our parallel implementation. As you can see from the Figure 4.2 the parallel version is not so effective when we increase the number of SaaS consistently. This is due to the fact that the time of the parts which remains sequential increase exponentially with the number of SaaSs considered. We need to find a more effective parallel implementation, I will continue this analysis during my thesis.

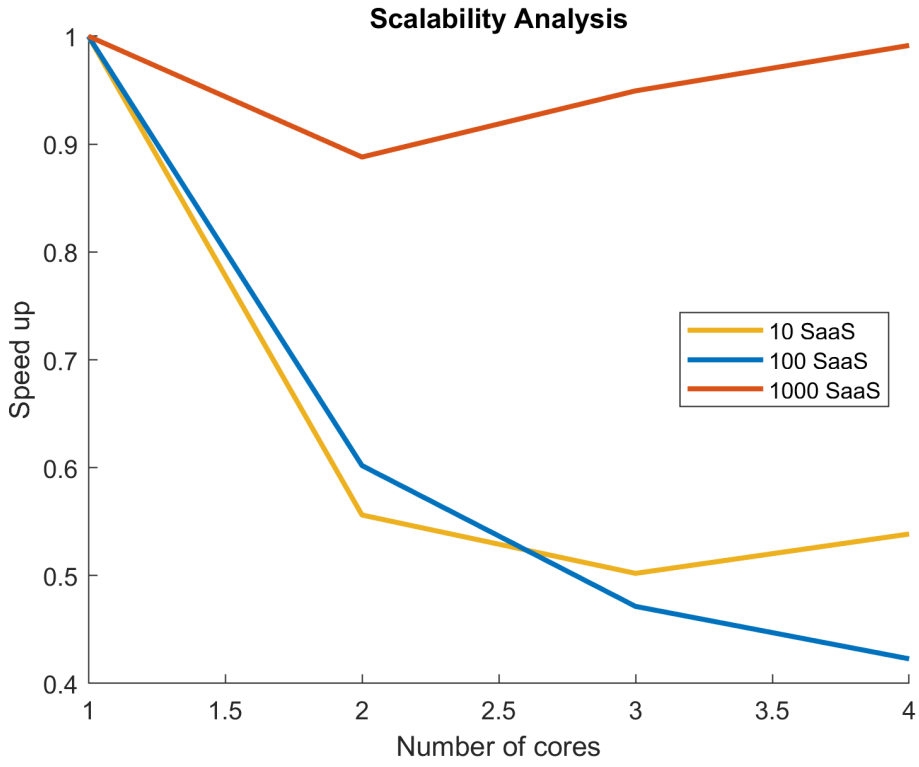

Figure 4.2

4.1.2 The effect of Γ

In this second analysis we observe how the SaaS reacts against the parameters uncertainty, this is done by changing the value of Γ, as we said before this parameters controls the number of uncertain parameters. Γ controls the uncertainty of the problem and it can varies between 0 (no uncertain parameters) and *Number of WSs* (all the parameters are uncertain).
For this analysis we consider the following seetings:

Number of SaaSs	1
Number of apps per SaaS	10
Number of WSs per app	5
N	180
Γ	0,1,2,3,4,5

We obtained the results in Figure 4.3, the number of VMs requested by the SaaS increase as Γ increase, the SaaS need more resources in order to protect itself against parameter uncertainty. At the beginning the SaaSs instantiate only on_flat and on_spot VMs until we saturate the available on_flat VMs, look equation 2.6 (in this case we have $R_j = 60$), this point is reached with $\Gamma = 3$. From the image seams that the previous point is reached at $\Gamma = 2$ but the exact value for the on_spot VMs with this value of Γ is 59. Hence with $\Gamma = 3$ we start allocating also on_demand VMs (the most expensive one). As expected the robustness of the problem turned out with an increase of the number of allocated resources, from the results file we can see that the number of rejected request remains constant for all the value of Γ, this because with $\Gamma = 0$ the variable $y_{a,w}$ is already set at the maximum value (see equantions 2.4 and 2.5), the throughput ($X_{a,w}$) is set equal to the minimum arrival rate $\lambda_{a,w}$ so $y_{a,w}$ reaches its maximum value, this is due to the initialization parameter in particular $\nu_{a,w}$.

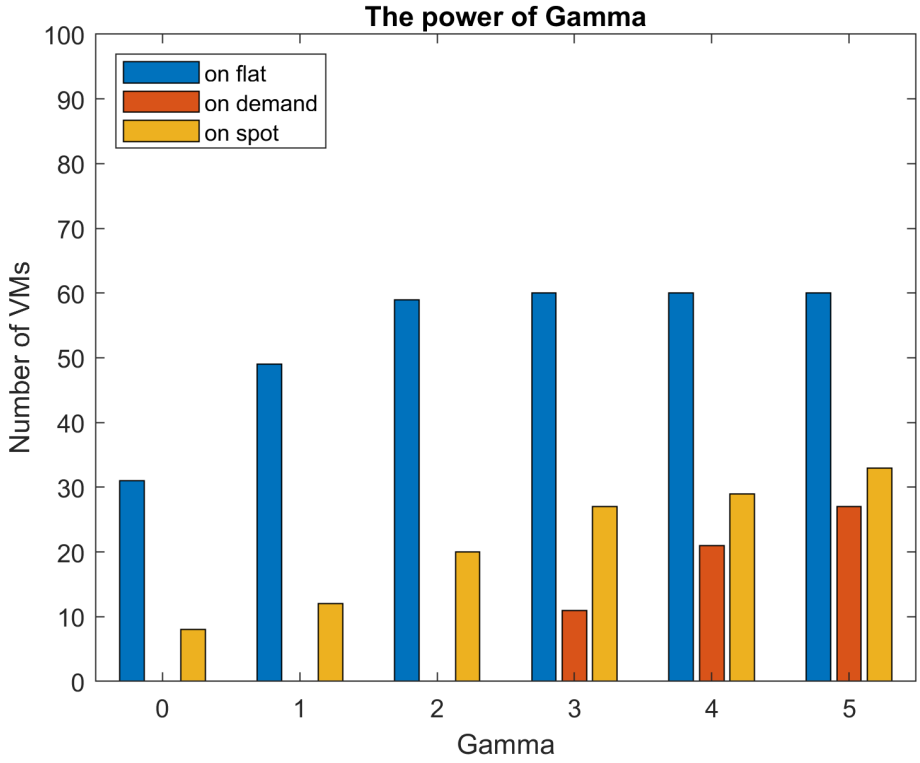

Figure 4.3

4.1.3 Daily Analysis

Finally we observe the behaviour of the SaaS forcing $\Lambda_{a,w}$ to follow the trend of the traces in Figure 4.1.As said in the previous chapter this is done explotting *setup.py* and *run.py*, if you want to perform this analysis you can change the trace in the setup file. Pay attention to the fact that we are following only the trend of the traces and not the exact values.
We used the default parameters of the problem ($N = 10000$ and $\Gamma = 1$) and only 1 SaaS with 10 applications and 5 web services per apps.

We collected the results in Figure 4.4 and 4.5, during the spiky day (Trace n°1) the number of on_flat VMs is more stable and it follows the trend of Λ_a in particular we can see the peak at the end. On the other hand the number of on_spot VMs is really unstable and it seams to follow the noise of Λ_a.

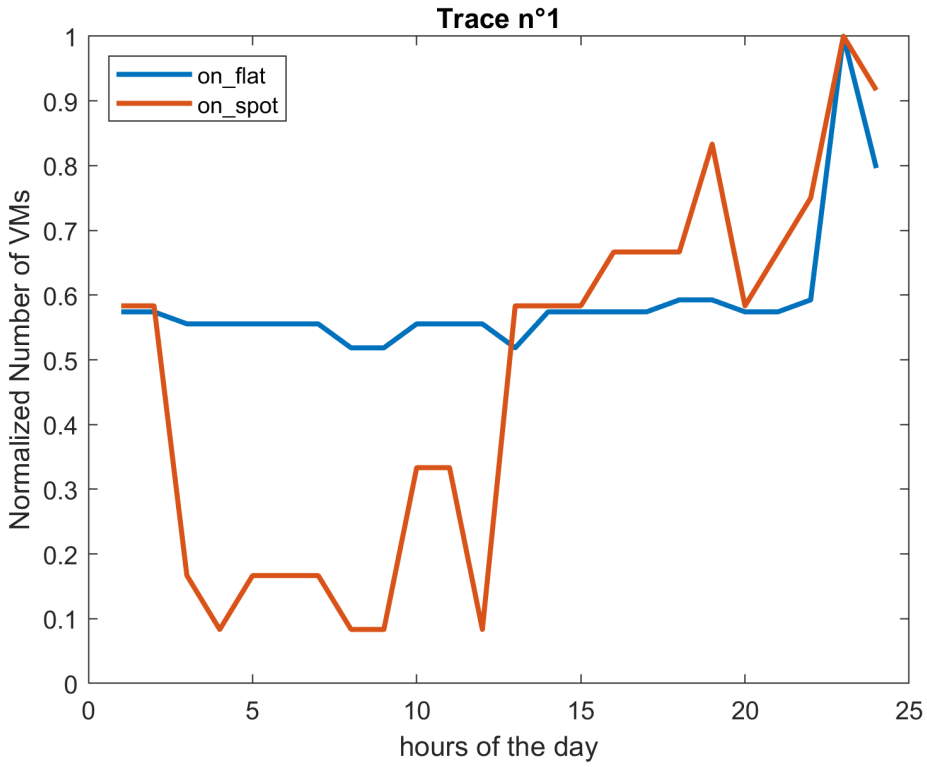

Figure 4.4

In the second case both on_flat and _spot VMs follow the general pattern of Λ_a but it's more accentuated for the on_spot VMs which goes to zero when the arrival rate is minimum. The difference between the two case is not so evident, in both simulation the changes in the number of on_flat VMs are smoother than the changes in the on_spot one, the cause could be the reliability constraints, we need to go deeper into this behaviour during my master thesis.

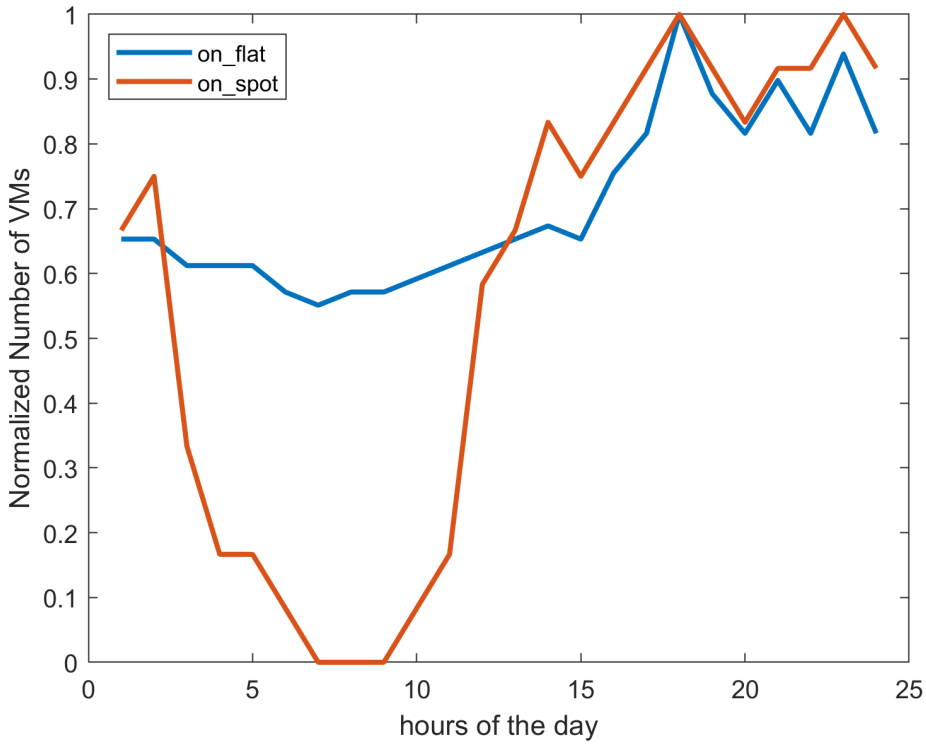
on_flat
on_spot
Normalized Number of VMs
hours of the day
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0
5
10
15
20
25

Figure 4.5

Bibliography

[1] Danilo Ardagna, Barbara Panicucci Mauro Passacantando; Generalized Nash Equilibria for the Service Provisioning Problem in Cloud Systems *2014*

[2] Dimitris Bertsimas, Melvyn Sim; The Price of Robustness *2002*

[3] Mauro Passacantando, Danilo Ardagna, Anna Savi; Service Provisioning Problem in Cloud and Multi-Cloud Systems. *INFORMS Journal on Computing, 2016*

[4] Simon Spinner, Giuliano Casale, Fabian Brosig, Samuel Kounev; Evaluating approaches to resource demand estimation. *Elsevier, 2015*

[5] Danilo Ardagna, Michele Ciavotta, Riccardo Lancellotti, Michele Guerriero; A Hierarchical Receding Horizon Algorithm for QoS-driven control of Multi-IaaS Applications.

[6] Alistar Croll; Cloud performance from the end user perspective.

[7] J. Almeida, D. Ardagna, I. Cunha, C. Francalanci, M. Trubian; Joint admission control and resource allocation in virtualized servers. *Journal of Parallel and Distributed Computing, 2010.*

[8] Amazon Inc. Amazon Elastic Cloud. *http://aws.amazon.com/ec2/.*

[9] Amazon Inc. AWS Elastic Beanstalk. *http://aws.amazon.com/elasticbeanstalk/.*

[10] R. Birke, A. podzimek, L. Y. Chen, E. Smirni. State-of-the-practice in data center virtualization : Toward a better understanding of VM usage. *43rd IEEE/IFIP International Conference on Dependable System and Networks (DSN), 2013.*

[11] G. Bolch, S. Greiner, H. de Meer, K. Trivedi. Queueing Networks and Markov Chains. *1998*.

[12] S. Kumar, V.Talwar, V. Kumar, P. Ranganathan, K. Schwan. vManage: loosely coupled platform and virtualization management in data centers. In *ICAC2009 Proc., 2009*.

[13] Y. Mei, L. Liu, X. Pu, S. Sivathanu, X. Dong. Performance analysis of network i/o workloads in virtualized data centers. *Services Computing, IEEE Transactions on, 6(1):48-63, 2013*.

[14] D. A. Menascé and V. Dubey. Utility-based QoS Brokering in Service Oriented Architectures. In *IEEE ICWS Proc., pages 422-430, 2007*.