Business-Driven Optimization of Policy-Based Management solutions

Issam Aib and Raouf Boutaba
David R. Cheriton School of Computer Science,
University of Waterloo, Canada
{iaib;rboutaba}@uwaterloo.ca

Abstract— **We consider whether the off-line compilation of a set of Service Level Agreements (SLAs) into low-level management policies can lead to the runtime maximization of the overall business profit for a service provider. Using a simple web application hosting SLA template for a utility service provider, we derive low-level QoS management policies and validate their consistency. We show how the default first come first served (FCFS) mechanism for the runtime scheduling of triggered policies fails to deliver an all times maximum business profit for the service provider.**

To achieve a better business profit, first a penalty/reward model that is derived from the SLA Service Level Objectives (SLOs) is used to assign runtime utility tags to triggered policies. Then three policy scheduling algorithms, which are based on the prediction of the future state of the running SLAs, are used to drive the runtime actions of the Policy Decision Point (PDP). The prediction function per see involved the unsolved problem of predicting in realtime the evolution of the transient state of a variant of an $M/M/C_t/C_t$ queue. A simple approximative solution to the latter problem is provided. Finally, using the $\mathcal{PS}$ policy simulator tool, comparative simulation results for the business profit generated by each of the proposed policy scheduling algorithms are presented. $\mathcal{PS}$ is a novel tool which we have developed to respond to the increasing need of benchmarking SLA and policy-based management solutions.

I. Introduction

By separating the rules that govern the behavior of an information system from the functionality it supports [8], policy-based management (PBM) promises to reduce management costs while simultaneously improving quality of service (QoS) and dynamic adaptability to change. It is currently present at the heart of a multitude of management architectures and paradigms with such diversified prefixes as SLA-driven , business-driven, autonomous, adaptive, and self management, to name a few. In this paper we only focus on QoS related policies. However, policies are also extensively used in the security arena. Hence being able to use policies to govern an information system brings the key advantage of uniform management by using one unified paradigm.

Although research in policy-based management has been going on for more than a decade, it is still not easy to put into practice. This owes much to the theoretical and practical difficulties in proving not only the correctness but also the efficiency of policy-based solutions when it comes to the management of real scale systems with hundreds or even millions of policies interacting in a dynamic way. In addition, A multitude of policy languages and architectures have been proposed but techniques for refinement, and consistency/completeness analysis remain scarce. It is then understandable why venturing into a full PBM solution for managing one's information infrastructure remains difficult to justify.

System performance also lies at an equal level of interest. While it is true that policy-based solutions promise dynamism and flexibility, they often come with no guarantee of high performance. Verma [11] states that PBM solutions should not be considered a case of expert systems because of the strong weight of the performance parameter they have to carry with them. Work on the benchmarking of PBM solutions is also marked by great scarcity. It is in fact insufficient to provide a PBM solution which works, also must it be as efficient as existing legacy solutions if not better. In this regard, the maximization of business profit should represent the crucial goal that any QoS PBM solution should target.

In this work, we develop a detailed use case for the business-driven SLA refinement into low-level management policies. In addition, an implementation which maximizes the business profit of the service provider is presented. The use case spans all of the business-driven SLA management loop. We show how the refinement process is done and how the static consistency and stability analysis can help detect anomalies which are not easy to detect. We then introduce a new approach to business-driven dynamic policy analysis in which we emphasize the need for incorporating business (and SLA) related data, encoded mainly within metrics generated during the refinement process, to handle the orchestration of policies at runtime. This analysis proves to be crucial in making the same set of generated policies achieve best performance at runtime.

By its nature, our presented work intersects with many related efforts in the business, SLA, and policy-based management. QoS Policy refinement has been addressed in [3] with a use case for DiffServ QoS management. It stresses the need of application specific policy refinement patterns and presents a tool that is being developed for that purpose. Our work agrees with this and is complementary to it by detailing the consistency checking into several sub-constituents (section VI) such as policy set stability (loop/deadlock free). Our study also emphasizes the necessity of application specific policy refinement patterns.

In another respect, policy stability has been addressed recently in [4]. The paper uses control theory for studying

1) Customer $\mathcal{C}$ is provided a web application hosting service with schedule sc
2) Max Capacity is of cp_{max} simultaneous connections
3) $\mathcal{C}$ is charged $\$ch = a \times cp_{max}$ monthly
4) Monthly average availability of the hosted service $\geq av_{min}$
   - An i^{th}successive availability violation incurs a reward of $r_i \times ch$
   - At the 3^{rd}successive availability violation the SLA is considered void
5) Minimum average time to process customer service requests $= rt$ ms
   - Otherwise, $\mathcal{C}$ is rewarded $rt.ref \times rt$

Fig. 1. Generic web application hosting SLA: $\mathcal{AP}$ SLA

the stability of feedback regulation on SLA-like policies in bilateral provider agreements. Our work tackles also the stability issue and shows, among others, a technique for detecting and preventing policy loops that might occur between different QoS policies (section VI).

On the monitoring loop side, the automated generation of metrics for SLA monitoring has been addressed in [7] [9] for the special case of web service SLAs. [7] stresses the need for customer side monitoring and provides a comprehensive XML based notation for the specification, but not for the automatic generation, of composite metrics for web services. In our work we show through a detailed example, but do not develop a full theory of, how metrics are identified and generated from a Service Level Specification (SLS) (section V-C).

Finally, for the business-driven decision making during SLA execution, [5] discusses a technique, independent of any PBM solution, to automate and/or assist service personnel to prioritize the processing of action-demanding quality management alerts as per provider's service level management objective.

This article will proceed as follows. Section II presents the generic SLA use case. Section III defines the business profit function we would like to optimize and section IV presents a lazy strategy to enforce it. Section V derives a formal SLS for the generic SLA and discusses the different stages of the proposed refinement process. After that, a static analysis is conducted on the generated SLS and shows how oscillation anomalies can be detected and fixed. Next, we present our dynamic analysis approach in which we provide an approximate solution of the transient state of a variant of an $M/M/C_t/C_t$ queue. We then use this solution to derive better runtime scheduling algorithms of triggered policies and hence a better generated business profit for the same low-level policy set. Sections X and XI present the simulation environment we used, the conducted simulations and finally summarized results of the performance of our policy scheduling algorithms.

II. Generic $\mathcal{AP}$ SLA

We consider a web Application hosting service Provider $\mathcal{AP}$ which offers a set of SLA contracts to its customers. The $\mathcal{AP}$ operates in its information infrastructure a pool of server units of size $sp.cp$ with fixed cpu capacity and which it can allocate to different SLA instances. The advertised SLA types are derived from the simple generic SLA of figure 1. This SLA states, in 5 clauses, that $\mathcal{AP}$ offers a web application hosting service supporting a load (capacity) of cp_{max} simultaneous end client connections to the system, an availability average of av_{min}, an average response time rt, all with a monthly cost ch.

Each instance of the tuple (sc, cp_{max}, a, av_{min}, r_1, r_2, r_3, rt, $rt.ref$) can generate an SLA type which the $\mathcal{AP}$ can advertise to its potential customers. In the following, sla_i denotes an SLA type and $sla_{i,j}$ denotes SLA instance j of SLA type i, all of which are derived from the generic SLA of figure 1.

Before going further into the SLA refinement process, the first step is to define the business profit function the $\mathcal{AP}$ intends to maximize.

III. Defining the Business Profit Function

Denoted in this paper as Ψ, the business profit function reflects the measure of the profitability of the service provided by the $\mathcal{AP}$. In order to keep the use case simple, we define Ψ as the sum of the raw financial profit gained from each contracted SLA. This assumes that managing the $\mathcal{AP}$'s information infrastructure incurs a fixed cost which is independent of the number of contracted SLAs. Hence,

$$\Psi = \sum_{i \in \mathcal{SLA}} \left(\sum_{j \in \mathcal{SLA}_i} (RP(SLA_{i,j})) \right) \tag{1}$$

where $\mathcal{SLA}$ represents the set of all SLA types the $\mathcal{AP}$ supports, and $\mathcal{SLA}_i$ the number of contracted instances of SLA type SLA_i. The dynamic problem the $\mathcal{AP}$ has to solve is the maximization of Ψ. We will elaborate more on this in section VIII.

IV. Lazy enforcement strategy

The clauses of figure 2 help identify the set of Service Level Objectives (SLOs) the $\mathcal{AP}$ has to enforce. Using a policy-based approach, the $\mathcal{AP}$ still has a variety of choices as to how to enforce them. In this paper we assume that the $\mathcal{AP}$ adopts a lazy strategy in enforcing its contracted SLAs. This means that it will allocate server units to each SLA on a per need basis. Similar to the statistical multiplexing of traffic crossing a shared physical network cable, this technique allows the $\mathcal{AP}$ to contract a number of SLAs with a total maximum sever pool capacity beyond the actual capacity it possesses.

At SLA instantiation time a number of n_0 server units is allocated. When the connection intensity (number of simultaneous end customer connections) reaches a certain threshold thA of the current SLA capacity, a request is made to the server pool to get an additional server unit. Conversely, if a low threshold thR is reached an action is triggered to release a server unit to the pool of free server units. With this technique the $\mathcal{AP}$ aims to achieve a higher business profit than if it used a guaranteed enforcement approach (where all resources are allocated before SLA instantiation).

```
sloSet sloSetL = {
  slo $slo_1$ = $(ws.cp \leq cp_{max})$;
  slo $slo_2$ = $(ws.av \geq av_{min})$;
  slo $slo_3$ = $(ws.rt \leq rt)$;
}
```

Fig. 2. SLO set for the lazy enforcement of the $\mathcal{AP}$ SLA

```
role $\mathcal{AP}$ = {
 policyGroup $pg1$ = {
  double $thA$;  constraint $0 < thA \leq 1$;
  double $thR$;  constraint $0 < thR < thA$;
  int $n_0$;  constraint $1 \leq n_0$;
  double $av_w$ = 1 month; // availability window
  event not($slo_2$) $e1, e2, e3$;
  policy $p_1$ = { at $sc.deployTime$ do $ws.add(n_0)$ }
  policy $p_2$ = {
   on $(ws.load \geq thA)$ do $ws.add(1)$ where $(ws.cp \leq cp_{max})$}
  policy $p_3$ = {
   on $(ws.load \leq thR)$ do $ws.free(1)$ where $(|ws.su| > 1\}$ }
  policy $p_4$ = {
   on $e1$  do $c.credit(r1)$ where $not(p_5 \vee p_6)$}
  policy $p_5$ = {
   on $(e1 \rightarrow e2)$ // $e1$ followed by $e2$
   do $c.credit(r2)$
   where $((time(e2) - time(e1) < av_w) \wedge not(p_6))$}
  policy $p_6$ = {
   on $(e1 \rightarrow e2 \rightarrow e3)$
   do {$c.credit(r3)$;SLA.terminate()}
   where $((time(e3) - time(e1) < av_w))$}
  policy $p_7$ = { at $sc.undeployTime$ do SLA.undeploy()}
 } // end of policyGroup pg1 } // end of role $\mathcal{AP}$
```

Fig. 3. Initial policy set for the Lazy enforcement of the $\mathcal{AP}$ SLA

V. Generating the $\mathcal{AP}$ Generic SLS

The $\mathcal{AP}$ Service Level Specification (SLS) is the result of the refinement of the $\mathcal{AP}$ SLA. In the following, this will be done using the lazy enforcement strategy.

A. SLO set specification

Service Level Objectives (SLOs) represent logical constraints on SLA parameters. When an SLO is violated, an event is generated which triggers concerned policy rule(s). The policy rule(s) will implement the penalty mechanism specified in the SLA as a result of the SLO violation. Figure 2 gives the generated SLO set for the generic $\mathcal{AP}$ SLA.

B. Policy set Generation

The $\mathcal{AP}$ will have to enforce for each SLA instance the set of policies of figure 3. The notation we used for policy rules is inspired from [8]. The generation of this set of policies was done manually as there is currently no automated technique for doing so [3].

p_1 is a deployment-time policy which requests one server unit from the pool of server units and initializes the web application of the SLA. At SLA undeployment time, policy p_7 liberates reserved resources, deactivates active policies, in addition to some reporting. p_2 and p_3 implement the lazy enforcement approach by tracking the load of the server units available to the SLA and making necessary actions each time a threshold is crossed.

Availability is implemented as a sliding window average of length av_w. It is initialized to one month as indicated by clause 4 of the $\mathcal{AP}$ SLA (figure 1).

The latter clause also specifies the penalties incurred in case of availability violation. The policy set $\{p_4, p_5, p_6\}$ implements these penalties. p_4 states that on the occurrence of an event e_1 of type not(slo_2), meaning a violation of slo_2, the customer account needs to be credited with $r1$ monetary units. p_5 and p_6 respectively implement the penalties for the 2^{nd} and 3^{rd} successive violations. The **where** clause enforces the one month "memory" on successive violations and makes sure that only one of $\{p_4, p_5, p_6\}$ can be triggered at a time.

slo_3 (figure 2) is implicitly implemented by limiting the maximum load on each server unit to $su(rt)$.

C. Metrics definition and policy set completion

Metrics are needed in order to track SLO states and compute the business profit. Knowing the formula for computing the business profit Ψ, and having all SLOs and policies specified in detail can help automate the metrics generation process. Figure 4 exlains how this is done for the $\mathcal{AP}$ SLA.

```
• $slo_1 \Rightarrow$
   def metric $mWSCP = ws.cp$
   policy pmWSCP = {
    on $(mWSCP > cp_{max})$ do $generate(not(slo_1))$}
• $slo_2 \Rightarrow$
   def metric $mAv = ws.av$
   policy pmAv = {
    on $(mAv < av_{min})$ do $generate(not(slo_2))$}
• $slo_3 \Rightarrow$
   def metric $mRT = ws.rt$
   policy pmRT = {
    on $(mRT < rt)$ do $generate(not(slo_3))$}
• $p_2 \Rightarrow$
   def metric $mP2ThAdd = ws.load$
   policy pmP2ThAdd = {
    on $(mP2ThAdd \geq thA)$
    do  generate(mP2ThAddEv) }
    update $p_2$ = { on $mP2ThAddEv$ }
• $p_3 \Rightarrow$
   def metric $mP3ThRem = ws.load$
   policy pmP2ThRem = {
    on $(mP3ThRem \geq thR)$
    do  generate(mP2ThRemEv) }
    update $p_3$ = { on $mP2ThRemEv$ }
```

Fig. 4. Metric generation and policy set update

D. The completed $\mathcal{AP}$ SLS

Figure 5 shows the completed $\mathcal{AP}$ SLS, which is the result of the refinement of the $\mathcal{AP}$ SLA. The generation of this SLS involved three sub processes. It is interesting to note that part of the SLS needs to be enforced at the $\mathcal{AP}$ side, i.e. role $\mathcal{AP}$, while a second one needs to be enforced at customer side (role $\mathcal{C}$). The inclusion of third parties in the SLS specification is also possible but will not be discussed in this paper.

```
sls $\mathcal{SLS}$ = {
  // service provide role  role $\mathcal{AP}$ = {
  def metric $mWSCP = ws.cp$
  def metric $mP3ThRem = ws.load$
  ...
  policyGroup pg1 = {
   policy pmWSCP = {...}
   policy pmP2ThAdd = {...}
   ...
   policy $p_1$ = { ...
   policy $p_2$ = { on $mP2ThAddEv$...}}
 }
 // Customer role (to enforce by customer)
 role $\mathcal{C}$ = {
  policyGroup pgC = {
   policy pC1 = {
    on every month do $\mathcal{AP}$.credit(ch)
    start at sc.activationTime }
}} }
```

Fig. 5. Completed $\mathcal{AP}$ SLS

The resulting SLS hence contains two roles for each party of the SLS with the $\mathcal{AP}$ role composed of thirteen policies and four metrics.

VI. SLS Static analysis

The second phase after the generic SLS is generated consists of conducting a static analysis to test the consistency of the generated policy set. For the static analysis phase we identify five main types of tests: *action conflicts*, *deadlocks*, *oscillation* (loops), *unreachable states* (dead code (i.e policies)), and *erratic behavior*. The $\mathcal{AP}$ needs to test the generated policies to make sure they are free from any of these defects.

In the *action conflicts* test, policies with conflicting actions are checked for potential concurrent execution. For the $\mathcal{AP}$ SLS, p_1 and p_2 request additional server units but p_1 executes only once at SLA deployment time while p_2 becomes active only after the SLA activation. p_3 releases one server unit, which is an action expected to be always successful. p_4, p_5 and p_6 cannot execute at the same time even though this does not cause trouble. In the implementation, this translates to make sure method *c.credit()* is synchronized. When several SLA instances are running, policies p_1 and p_2 can conflict with each other but this is a runtime conflict that derives from the lazy enforcement strategy. So the policy set is action conflict free from the static analysis point of view.

For the *deadlock analysis*, it is straightforward that the policy set is deadlock free as there is only one possible blocking action *ws.add()*, which requests a number of server units from the pool of free server units. So a deadlock situation at runtime is not possible.

The next step in this phase is to check for potential *static loops*. A system static loop occurs if there exists at least one system state $\mathcal{S}_l \neq \mathcal{S}_{final}$ which when reached, the probability that the system will come back to $\mathcal{S}_l$ after some time $t > 0$ is equal to one.

Let $(ws.cp, rp)$ denote the runtime state of an SLA, where $ws.cp$ is the capacity in number of allocated server units and rp the accumulated raw profit. rp is affected by operations $\mathcal{AP}$.credit() of policy pC1 (figure 5) and c.credit() of policies p_5 to p_7 (figure 3).

We also consider the state of the $\mathcal{AP}$'s system to be the sum of the states of all the SLAs contracted by the $\mathcal{AP}$ augmented by the state of the free server units pool and all the business metrics the $\mathcal{AP}$ maintains.

Policies p_4 to p_5 cannot be the cause of a system state loop because they cannot occur more than once during any availability interval (one month) and if they occur all the three of them during the same availability interval, the corresponding SLA is terminated.

For policies p_1, p_2 and p_3 there is a possible static loop. This is because p_1 and p_2 request additional server units while p_3 requests an operation which nullifies their actions by freeing one server unit. So further analysis is required on these policies.

The constraints on the definition of thA and thR (figure 3) have been set following the intuition that the threshold to request a new server unit should be strictly greater than the one which frees an acquired one. Without these constraints and if thR was fixed, possibly due to a mistake by the system operator, to a value $> thA$, then the related SLA instance might never free any acquired server unit until it is terminated or on the other extreme it might show erratic behavior in case of shortage in server units.

For the first case, $ws.load = thA \Rightarrow p_2$ triggered $\Rightarrow$ new $ws.load < thA < thR$. So if the number of connections decreases, no action will be taken by the SLA and it will continue to hold resources it is actually not using.

The latter case, however, is more harmful. It occurs when p_2 is triggered while no resources are available in the system and $ws.load$ continues to grow until reaching thR. At this moment p_3 is triggered reducing $su.size$ by one.

Since we have

$$ws.load = \frac{|connections|}{(su.cp \times su.size())}$$

This implies that $ws.load$ increases. Hence, p_3 gets triggered again, and so on, until all but one of the server units are freed (because of the *where* clause in p_3). It is then expected that availability violations occur leading possibly to SLA termination (policy p_6). With a slight chance, the SLA can still survive if before p_6 is triggered the load on the unique left server unit diminishes. A way to detect this type of erratic behavior is to specify a rule for the static analyzer which states:

```
//fsupl = free server units pool
policy pS1 = {
  on $(triggered(p_6) \wedge (fsupl.size > 0))$
  do signal(erraticBehavior);}
```

With this semi-formal static analysis, the $\mathcal{AP}$ should be relatively assured that the generated SLS will achieve the goal of enforcing the contracted SLAs.

However, there is still a subtle error which we discovered only after running a batch of randomly generated simulations.

For some randomly generated input parameters which respect all the above stated constraints, the system still enters an infinite loop oscillating between policies p_2 and p_3. By analyzing closely this case we found that the constraint $0 < thR < thA \leq 1$ is not sufficient. This leads us to consider when p_3 can be automatically triggered once p_2 is triggered and vice versa.

Through mathematical analysis we prove that:

$$\mathcal{AP} \text{ SLS is loop free} \Leftrightarrow thA > 2 \times thR \tag{2}$$

This result completes the static analysis phase. The output of a static analyzer should be a recommendation to change the constraint on thR to become compliant with eq.2.

The next step will focus on exploring in depth some aspects of what we call a "business-driven dynamic analysis of the SLS". We provided the intuition around this new type of analysis in [2]. Although it can result in complex mathematical models, the following section stresses the need for considering this phase for any QoS PBM solution which aspires to be business-driven.

VII. Business-driven dynamic analysis

The dynamic analysis we conduct here stems from the conjecture that, however complex the technique employed to compile an SLA into low-level SLS, one cannot be sure, and actually has no proof, that assigning a set of priority levels to low-level policies would be sufficient to maximize the overall business profit of the SLA provider. Looking in the literature on policy-based management only reinforced our belief in this conjecture. In what follows we develop a technique for handling policies at runtime which proved (Section XI-B) to generate a much higher performance at runtime with the same generated set of policies.

At normal SLA operation all triggered policies should execute correctly. Conceptually, a triggered policy needs the approval of the policy decision point (PDP) before it can execute [11]. This implies the existence of a conceptual queue, or waiting room, for triggered policies which the PDP serves by scheduling them according to some predefined scheme. In practice, the PDP can be implemented as a hierarchy of PDPs distributed within the information infrastructure. Our analysis will be based on the conceptual PDP of the $\mathcal{AP}$'s information infrastructure.

The default service the PDP offers to the Triggered Policies Queue (TPQ) is a FCFS service. In the case a triggered policy cannot execute because of insufficient system resources, which can occur for p_1 or p_2, the PDP skips it until resources become available for its execution.

If the $\mathcal{AP}$ chooses to contract a number of SLAs with a total maximum capacity exceeding the actual capacity it has in its servers pool, it can be proved that by carefully scheduling the execution of policies in (or from) the TPQ we can achieve a better overall business profit.

VIII. Business-driven TPQ scheduling

In this section we develop a technique for TPQ scheduling, which provides a better handling of peak utilization times for the $\mathcal{AP}$'s resources and yields a better overall business revenue. This technique takes into consideration the runtime states of instantiated SLAs in servicing the TPQ.

Since the only policies which may incur delay are p_2 and p_1 we will only consider them for this analysis. The other policies can hence be serviced according to the usual FCFS discipline without loss of performance or business value.

The decision problem the PDP has to solve when faced with a number of policies in TPQ requesting more resources than available (here server units) is to determine which policies to grant resources to, i.e. allow to execute, and which policies it will delay hoping that enough resources will be freed. Delaying a triggered policy can lead to a violation of SLOs and violating an SLO can cause penalties paid to the $\mathcal{AP}$ customers. The aim of the $\mathcal{AP}$ is to configure its PDP's decision algorithm so as to reflect the goal of maximizing the business profit function Ψ defined in eq.1.

A. Mathematical formulation of the Business-aware TPQ scheduling decision problem

Delaying p_1 or p_2 can lead to the violation of the availability SLO (slo_2 in figure 2). slo_2 is defined over the monthly average availability of the web application to end customers. Based on figure 1 , availability ($= ws.av$ in figure 2) of each SLA instance is defined as the fraction of successful service requests to the fraction of total service requests of end customers computed over a month time window.

Definition*: monthly availability of web service*

$$ws.av = \frac{|\text{processed requests}|_{\text{in}av_w}}{|\text{total number of requests}|_{\text{in}av_w}} \tag{3}$$

When confronted with a sequence of p_1 and p_2 policies in the TPQ, the PDP can utilize the information on the availability metric for the SLA to which each policy is associated in order to predict the impact time associated to each policy.

We will make use of a *greedy approach* to business profit maximization. We consider the maximization of Ψ to be approximated as the minimization of impact (i.e. loss) on Ψ for each decision cycle the PDP performs on the TPQ.

Let P_{tpq} be the set of policies of type p_1 or p_2 that are queued in TPQ at time t_0. The PDP constructs for each policy $p_i \in TPQ$ a tuple $(p_i, r_i, I(p_i))$. r_i is the time p_i was triggered. $I(p_i)$ is an impact probability function which gives for each $t \geq 0$ the expected impact on Ψ in the case p_i is delayed t time units after the current system time t_0. Based on these sets of tuples, the PDP has to make a decision which will lead to the best minimization of impact on Ψ.

B. Mathematical model of the $\mathcal{AP}$ SLA

Predicting the impact of delaying p_1 or p_2 implies predicting the violation probability of slo_2 in time. This implies predicting the evolution of availability ws_{av} in time. In its turn, this means predicting the number of processed requests and the total number of requests during the last availability window av_w of the SLA associated to each $p_i \in P_{tpq}$.

To do so we model the state of an SLA instance as a tuple $M/M/C_t/C_t\,|A_t|D_t$. $M/M/C_t/C_t$ models a variable capacity markovian queue where:

- $\lambda =$rate of end client requests
- $\mu =$service rate for a single client request
- the number of available server slots at time t $C_t = ws.cp = su(rt) \times su.size()$
- no waiting queue, all requests arriving at 100% load time are lost

A_t denotes for each t the number of granted end customer requests during the last availability window $[t - av_w, t]$. D_t denotes the number of the denied ones. $T_t = A_t + D_t$ represents the total number of arrivals during the last availability window.

Requests arrival is modeled as a poisson source as it reflects the most common type of arrivals. Exponential service times denote the time the customer remains connected to the web service. In the case of a web site this can model the time a customer spends surfing into the server web site. A similar case applies to other types of servers such as audio/video streaming servers. Note that this does not contradict with the response time SLO as the response time represents the responsiveness of the web service to end customer queries during one end customer session.

The servers' capacity varies within the discrete set $\{su(rt)$, $su(rt)$, $2su(rt)$, $\ldots$, $cp_{max}\}$.

In what follows we will focus more on policy p_2. The study of p_1 follows a similar approach.

Let N_t denote the number of end customer requests being serviced at time t. We have at any time $0 \leq N_t \leq C_t$

Note that all of A_t, D_t, T_t, C_t, and N_t can be easily obtained at runtime by defining corresponding metrics at the SLS level.

Policy p_2 is triggered when the SLA load exceeds thA. Let t_0 denote this time. In the following, when t_0 is used as a subscript we will omit the t for clarity.

Because markovian processes are memoryless, the PDP can take as input to its Impact Prediction Algorithm (IPA) the tuple $(t_0, A_0, D_0, C_0, N_0)$. The output is the probability function I. As a simplification of I, the IPA can determine the time it will take starting from t_0 until av_t drops bellow av_{min}, hence triggering an availability violation.

In spite of all the simplification done still remains the difficult problem of predicting the evolution of a Markovian process at transient state.

On page 78 of his book *Queueing Systems, Vol. 1*, Leonard Kleinrock, commenting on the transient solution of an $M/M/1/\infty$ queue, says 'This last expression is most disheartening. What it has to say is that an appropriate model for the simplest interesting queueing system leads to an ugly expression for the time dependent behavior of its state probabilities.'

More recently, Sharma describes in his book *Markovian Queues* [10] a novel approach to the transient analysis problem. He was the first to provide the transient solution for the $M/M/\infty$, $M/M/N$ and $M/M/2/N$ queues. Sharma states that for higher order queues the problem becomes much more complicated to be handled by currently known Algebraic techniques. This problem remains open until today.

In order to make the policy decision-making process business aware, and based on the above literature, we attempt here to find an acceptable solution to the transient analysis of an $M/M/C_t/C_t$ queue. Indeed, this is still feasible even with the very short time frame (near realtime) available to the PDP. The decision making problem is further complicated by the fact that the PDP is not just concerned with predicting when the servers become 100% loaded. It is in fact concerned with predicting when, after all servers are fully loaded, will the availability drop bellow its minimum contracted value.

In the following two subsections we present the technique we developed and for which we provide performance results in section XI-B.

C. *Mathematical approximation of the first time to violation*

As a first approximation we divide the prediction process into two phases, the fill up phase and the time to violation phase. In the first phase the system starts with configuration $(t_0, A_0, D_0, C_0, N_0)$ and evolves to the new configuration $(t_f, A_f, D_f = D_0, C_f = C_0, N_f = C_0)$, where t_f represents the time when the SLA reaches full load ($N_f = C_0$). During interval $[t_0, t_f]$ it is expected that no service request denial occurs as the SLA can still handle more load. The time to violation phase starts at time t_f and lasts until reaching the violation of availability SLO (slo$_2$) and is described by configuration $(t_v, A_v, D_v, C_v, N_v)$ where $\frac{A_v}{A_v+D_v} \leq av_{min}$ (and also $\simeq av_{min}$). Given this information, the PDP can predict that if it delays policy p_2 the corresponding SLA will experience a violation of slo$_2$ at time t_v, i.e. after a duration of $t_v - t_0$ from the current evaluation time.

The problem is hence amenable to providing an approximate solution of the fill up and time to violation phases.

1) Fill up phase - predicting t_f: The computation of t_f is based on the average behavior of the $M/M/C_t/C_t$ queue. The incoming flow λ of end customer requests is subdivided into two sub flows. A first sub flow with rate $\mu \times N_0$ works on average to keep busy the N_0 occupied server slots, as their aggregate average service rate is $\mu \times N_0$.

Let's denote the remaining sub flow as

$$\lambda' = \lambda - \mu \times N_0$$

λ' constitutes the new flow of incoming connections for the servers of rank $> N_0$ (after a simple reordering of server slots).

We now consider this new set of servers slots separately. At time t the number of connected customers is N'_t. At time $t + dt$ this number will increase by dN'_t where:

$$dN'_t = \lambda' dt - \mu N'_t dt$$

$$\Rightarrow \frac{dN'_t}{dt} + \mu N'_t = \lambda' \tag{4}$$

We define $\rho = \frac{\lambda}{\mu}$ and $\rho' = \frac{\lambda'}{\mu} = \rho - N_0$. With the condition $N'_0 = 0$ we get:

$$\Rightarrow N'_t = \rho'(1 - e^{-\mu t}) \tag{5}$$

By putting:

$$t_f = t_0 + t'_f \tag{6}$$

t'_f is then the solution in t of $N_t = C_0$. Hence,

$$t'_f = \frac{-1}{\mu} ln \left(1 - \frac{C_0}{\rho'}\right)$$

$$\Rightarrow t'_f = \frac{1}{\mu} ln \left(\frac{\rho - N_0}{\rho - N_0 - C_0}\right) \tag{7}$$

2) Time to violation phase - predicting t_v: Following the same reasoning, we assume that λ gets divided into two sub flows. The first sub flow of rate $\mu \times C_0$ works on keeping all server units busy. The second sub flow represents the loss flow and serves to count down towards availability violation.

As in the first case, we define our modified incoming flow as:

$$\lambda'' = \lambda - \mu \times C_0 \text{ , and } \rho'' = \frac{\lambda''}{\mu} = \rho - C_0$$

All customers in the poisson flow of rate λ'' get rejected. Hence, on average availability is expected to be violated at $t_v = t_f + t'_v$ where:

$$av_{min} = \frac{A_f + \mu \times C_0 t'_v}{T_f + \lambda \times t'_v}$$

Let $\triangle av = (av_{min} - av_0)$. Using algebraic computations we find the expected absolute time t_v at which a violation of the availability SLO will occur as follows:

$$\begin{aligned} t_v &= t_0 + \frac{(C_0 - \rho)}{\mu\,(C_0 - \rho\, av_{min})} \\ &\quad \times \left(\frac{T_0 \times \triangle av}{(C_0 - \rho)} - ln\left(1 - \frac{C_0}{\rho - N_0}\right)\right) \end{aligned} \tag{8}$$

□

t_v represents, in average, the time at which a violation is expected to occur if p_2 is not granted execution privilege. It does not necessarily reflect the actual time of first violation at runtime.

However, it is interesting to note that this formula can be computed in O(1) if the values of $\{C_0,\ T_0,\ N_0,\ av_0\}$ are available. Fortunately, the metrics instantiated from the developed $\mathcal{AP}$ SLS (figure 5) provide this information instantly at runtime. This has a definite advantage at runtime over any formula which predicts the exact transient evolution of an $M/M/C_t/C_t\,|A_t|D_t$ queue, even though such a formula has not been discovered yet [10].

D. Impact Minimization Scheduling (IMS) Algorithms

Provided with an approximation for t_f and t_v, the PDP can be configured to use several possible algorithms for the minimization of impact on the business profit function Ψ. In this work we developed three different algorithms the performance of which is evaluated through simulations in section XI-B.

First, for each triggered policy $p_i \in SLA_{p_i}$, the PDP will create a tuple $(p_i, SLA_{p_i}, tt_{p_i}, td_{p_i}, tv_{p_i}, Pn_{pi})$, where:

- tt_{p_i} corresponds to the triggering time of p_i,
- td_{p_i} the time of the next service degradation phase (corresponds to t'_f in eq.6),
- tv_{p_i} the expected time of availability SLO violation (t_v in eq.8) in case p_i is delayed,
- and Pn_{p_i} which is the penalty incurred based on the rules defined in SLA_{p_i} (one of $\{p_4, p_5, p_6\}$ depending on the runtime state of SLA_{p_i}).

The PDP has, among other possibilities, the following set of different scheduling policies (techniques) to select the next triggered policy to execute from TPQ :

- select the one with the first(lowest) time to violation, $Min(tv_{p_i})$
- select the one with the first(lowest) time to degradation, $Min(td_{p_i})$
- select the one with highest penalty first, $Max(Pn_{pi})$

This selection is applied of course to those policies whose action part can be satisfied in terms of resource availability. Policies whose actions require more resources than available will be delayed until there are enough resources to execute them.

In the remaining of this paper, we will use simulations to evaluate the performance of these algorithms and study how they compare to the default FCFS scheduling.

IX. Simulation environment: The $\mathcal{PS}$ Policy Simulator

The first problem faced when implementing the $\mathcal{AP}$ generic SLA use case is the lack of a simulation environment for testing the performance, correctness, and other properties of policy-based solutions. We therefore developed a full fledged policy simulation tool to use for the $\mathcal{AP}$ generic SLA use case. However, we have designed $\mathcal{PS}$ in such a way that it can be used by the wider SLA and Policy-based management research community. The detailed description of $\mathcal{PS}$ is beyond the scope of this paper and will be subject to a separate publication. In this section, we will describe briefly $\mathcal{PS}$ main characteristics.

The policy simulator $\mathcal{PS}$ implements a discrete event simulation system based on the process interaction world view. The design of it follows that of the business-driven management framework we proposed in [2]. $\mathcal{PS}$ builds on the open source package *javaSimulation* [6], which is a Java package for process-based discrete event simulation. The design of javaSimulation follows closely the design of the SIMULA programming language.

$\mathcal{PS}$ features the core of a policy-based management architecture augmented by SLA and business awareness [2]. These include the support for SLS specification, definition of business objectives, specification of metrics at resource, service and up to the overall business profit function Ψ. Also, included is the connection with the business profit maximization (impact minimization) engine at the PDP level by determining which decision making algorithm to apply with respect to the scheduling of TPQ.

SLAs are modeled according to the simple GSLA information model, introduced in [1]. In this model, an SLA is composed of a service package and a set of parties. Each party plays a (set of) role(s) related to a service provision/consumption. A Role defines exactly the duties of a party. As an example, in the $\mathcal{AP}$ SLS specification of figure 5 two roles were defined, the service provider role $\mathcal{AP}$ and the service Customer role $\mathcal{C}$.

X. Generic $\mathcal{AP}$ SLA simulation package

This package (figure 6) was built as a simulation instance which we run over $\mathcal{PS}$. Also, because of the relative completeness of the $\mathcal{AP}$ SLA test case, in terms of the exhaustive usage of $\mathcal{PS}$ components, the $\mathcal{AP}$ package served as a validation test unit many subtle tunings of the policy simulator.

The $\mathcal{AP}$ generic SLS was implemented as a single class descendent of the $\mathcal{PS}$ SLA class. The class contains two instances of the Role class implementing the $\mathcal{AP}$ role and $\mathcal{C}$ role respectively. The same hierarchy is constructed for policy groups, policies, metrics, and events. Each $\mathcal{AP}$ role has a serverGroup instance which manages a set of server units it gets from the $\mathcal{AP}$ serverPool component. A Poisson traffic source is attached to each serverGroup and is used to simulate session requests of end customers. At the reception of each session request the severGroup object calls an exponential random generator to set the duration of the new session.

Almost all communications between the simulation components are done via events. The event service allows any component to register as a source of a given event type. Time events (timeout counters) are also supported as a special event type. Another component can register as a listener to the same event type from that event source and the event service maintains this relationship. Policies are triggered by events. The server pool, for example, generates an event each time it receives, accepts, denies or terminates a session. Leaf metrics propagate information they receive from server pool events and other components up to higher level metrics $(A_t, D_t, T_t, Av_t, RP_t, N_t, C_t, etc.)$ until reaching the overall business profit function Ψ. To help analyze/debug simulation execution graph components can be hooked to any metric and report their evolution in time and record them into matlab scripts developed for this purpose. Matlab has been selected as the graph plotter because of its ability to handle relatively large graph files.

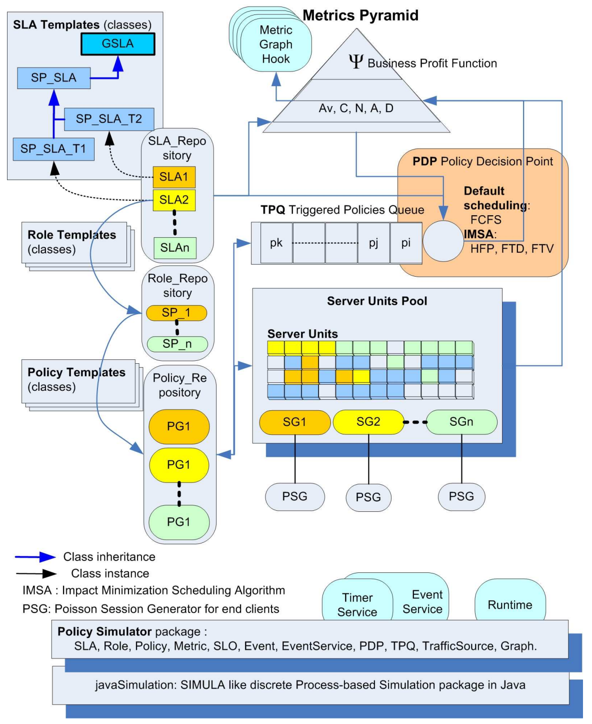
SLA Templates (classes)
GSLA
SP_SLA
SP_SLA_T2
SP_SLA_T1
Metric Graph Hook
Metrics Pyramid
Ψ Business Profit Function
Av, C, N, A, D
SLA_Repository
SLA1
SLA2
SLAn
PDP Policy Decision Point
Default scheduling: FCFS
IMSA: HFP, FTD, FTV
TPQ Triggered Policies Queue
pk
pj
pi
Role Templates (classes)
Role_Repository
SP_1
SP_n
Server Units Pool
Server Units
Policy Templates (classes)
Policy_Repository
PG1
SG1
SG2
SGn
PSG
Class inheritance
Class instance
IMSA : Impact Minimization Scheduling Algorithm
PSG: Poisson Session Generator for end clients
Timer Service
Event Service
Runtime
Policy Simulator package : SLA, Role, Policy, Metric, SLO, Event, EventService, PDP, TPQ, TrafficSource, Graph.
javaSimulation: SIMULA like discrete Process-based Simulation package in Java

Fig. 6. The $\mathcal{AP}$ testbed over $\mathcal{PS}$

XI. Simulation Results

A. Generation of input batches

In order to validate the policy-based implementation of the generic $\mathcal{AP}$ SLS solution and more importantly to compare the performance of each of the impact minimization scheduling algorithms (HPF, FTD, and FTV) against the basic FCFS scheduling, there is a need to generate an acceptable number of simulation instances each with different values for the main simulation parameters.

For each simulation instance we fixed the values for the following sets of parameters:

- *Simulation wide parameters*
  Including simulation life time (start time, duration (was fixed to 6 months)), location of simulation input property files, output folder, etc.
- $\mathcal{AP}$ parameters
  Including the scheduling algorithm to use (i.e., {FCFS, HPF, FTD, FTV}) the number of supported SLA types, number of instances of each SLA type, server pool capacity in number of server units,
- *SLA type parameters*
  Including cp_{max}, a, rt, $rt.ref$, av_{min}, av_w, λ, μ, $su(rt)$, penalties $\{r_1, r_2, r_3\}$, thA, thR and availability probe interval.

In addition to the above mentioned parameters, a number of simulator and policy specific parameters are used but not described here for the sake of clarity.

For statistical significance, we run an acceptable number of simulations using a random input generator which respects the

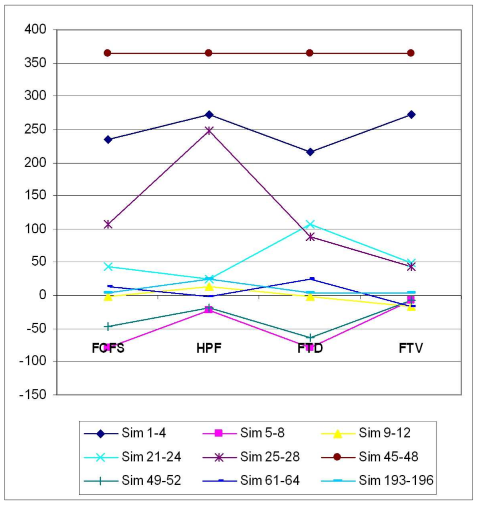

Fig. 7. Generated Ψ by the different TPQ scheduling Algorithms

above constraints. An MS excel VB script was used to generate an acceptable number of simulation inputs. Spread sheets were useful because they allow to automatically update cells through existing formulae. We generated an actual number of 384 simulation inputs and estimate that running roughly 5% ($\simeq$20) of this number (by random selection within the space of the 384 simulation inputs) can give a good first impression of the performance of the proposed scheduling algorithms. Running the whole set of simulations would certainly provide a better view but will not be feasible within a reasonable amount of time (with 15 parameters, each varying for instance through 5 values only, will yield a simulation space of 5 power 15, where one simulation instance runs within an interval of 12 hours to several days).

B. Summarized results

A number of thirty six simulations was conducted. Several computers have been used to run small batches of four simulations each (for the four scheduling algorithms). Two SLA types were used with $av_{min} = 99\%$ and 99.9% respectively. Ten SLA instances were used for type 1 SLA and Eight for type 2. In addition, the following key parameters $<$ratio of actual to expected Max(server pool size), ρ, thA, $thR >$ ranged, respectively, within the values $< 70\% - 100\%$, $< 6.3, 69, 90, 100 >$, $< 74\%, 99\% >$, $< 25\%, 34\% >>$.

Three P4 1.6GH Windows machines, two Sun OS SUNW-Ultra-4 300MHZ machines, and two Sun Ultra 60 (512 MB RAM, 450 MHz) Solaris 8 machines were used. The simulations run in an accumulated cpu time of $\sim$one month, giving an average of $\sim$16 hours per simulation.

Figure 7 depicts the generated business profit Ψ for each of the three proposed impact minimization scheduling algorithms against the default FCFS scheduling.

According to Figure 7, it appears that FCFS never achieves the best performance except for simulation batch 45-48 were all scheduling algorithms produce the same value for Ψ. Second, no single IMS algorithm performs best at all times.

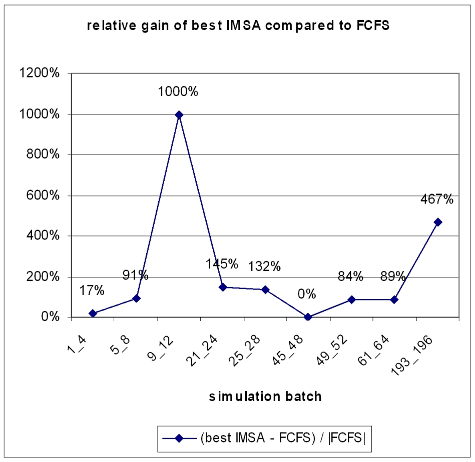

Fig. 8. Performance of FCFS against Min/Max IMSA

HPF performs best 5 times, FTD 3 times, and FTV 4 times. Surprisingly, FCFS, although it never outperformed the maximum output of the three IMS algorithms, in Sim 1-4 it actually outperformed FTD; in Sim 9-12 it outperformed FTV and in Sim 61-64 it performed better than HPF. Hence, FCFS is not that bad after all! Figure 8 traces the performance of FCFS against the best performance produced by the three IMS algorithms as well as the worst one. FCFS can hence be viewed to provide average performance between the worst and best achieved Ψs.

To better illustrate the importance of this search for Max(IMSA), figure 9 plots, for each simulation batch:

$$\frac{Max(\Psi_{IMSA}) - \Psi_{FCFS}}{|\Psi_{FCFS}|}$$

It shows the relative gain of the best Ψ produced by the three IMS algorithms and Ψ produced by FCFS. The gain reaches up to 1000% in batch 9-12 and 467% in batch 193-196.

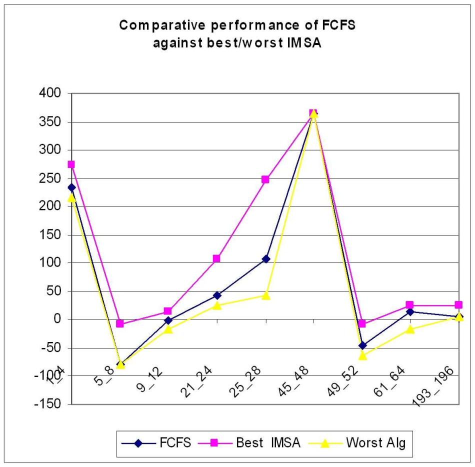

Fig. 9. Relative gain of *Max(IMSA)* compared to *FCFS*

XII. Conclusion

In this paper, we presented a detailed case for the business-driven refinement and implementation of a generic web application hosting SLA into low-level QoS management policies. The refinement process involved an iterative approach the output of which was a set of metrics and low-level QoS policies structured into roles. The static analysis phase served in detecting static anomalies in the generated SLS as well as discovering additional constraints important for the runtime stability of the SLS. In the dynamic analysis phase, we attempted to bridge the gap between low-level management actions and the high-level business profit of the service provider. The bridging involved the difficult problem of the realtime resolution of the transient state of a variant of an $M/M/C_t/C_t$ queue. We provided a mathematical approximation which respected the realtime requirement and achieved acceptable results. Using the results of the dynamic analysis phase, three greedy algorithms were proposed for the runtime maximization of the business profit. These algorithms scheduled the queue of triggered policies by minimizing the impact of local penalties in the hope to achieve a minimal loss in the overall business profit.

Using $\mathcal{PS}$, a new tool we developed for the simulation of policy-based management solutions, we implemented all the required components for the $\mathcal{AP}$ SLA use case as well as all the proposed policy scheduling algorithms. The simulations showed interesting results, where the impact minimization scheduling algorithms managed to provide an overall business profit that is up to three orders of magnitude higher than the default behavior. Another interesting result was that no single algorithm provided best performance at all times.

Bridging the gap between high-level business goals and low-level management actions is a difficult research subject. This work provided an insight on how it can be tackled. We believe that the policy simulator tool $\mathcal{PS}$ can be helpful in this regard by providing an environment where policy-based management solutions can be assessed for their business-related viability and efficiency. It is therefore our intention to make the tool available for the research community at large.

References

[1] I. Aib, N. Agoulmine, and G. Pujolle. A multi-party approach to SLA modeling, application to WLANs. In *Second IEEE Consumer Communications and Networking Conference (CCNC)*, pages 451 – 455. IEEE, Jan 3-5 2005.

[2] I. Aib, M. Salle, C. Bartolini, and A. Boulmakoul. A business driven management framework for utility computing environments. In *proceedings of the Ninth IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)*. IEEE, May 16-19 2005.

[3] A. K. Bandara, E. C. Lupu, A. Russo, N. Dulay, M. Sloman, P. Flegkas, M. Charalambides, and G. Pavlou. Policy refinement for diffserv quality of service management. *IEEE eTransactions on Network and Service Management (eTNSM)*, 3(2):12, 2nd quarter 2006.

[4] K. Begnum, M. Burgess, T. M. Jonassen, and S. Fagernes. On the stability of adaptive service level agreements. *eTransactions on Network and Service Management (eTNSM)*, 2(1):13–21, Jan 2006.

[5] M. J. Buco, R. N. Chang, L. Z. Luan, C. Ward, J. L. Wolf, and P. S. Yu. Utility computing SLA management based upon business objectives. *IBM Systems Journal*, 43(1):159–178, 2004.

[6] K. Helsgaun. Discrete event simulation in java. Technical Report 1-1, Department of Computer Science, Roskilde University, DK-4000 Roskilde, Denmark, Department of Computer Science, Roskilde University, DK-4000 Roskilde, Denmark, March 2004.

[7] A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Journal of Networks and Systems Management*, 11(1), 2003.

[8] L. Lymberopoulos, E. Lupu, and M. Sloman. An adaptive policy based framework for network services management. *Journal of Networks and Systems Management (JNSM), Special Issue on Policy Based Management of Networks and Services*, 11(3):277–303, September 2003.

[9] A. Sahai, V. Machiraju, M. Sayal, A. P. A. van Moorsel, and F. Casati. Automated SLA monitoring for web services. In M. Feridun, P. G. Kropf, and G. Babin, editors, *DSOM*, volume 2506 of *Lecture Notes in Computer Science*, pages 28–41. Springer, Nov. 16 2002.

[10] O. Sharma. *Markovian Queues*. Mathematics and its applications. Ellis Horwood, 1990.

[11] D. C. Verma. *Policy-Based Networking: Architecture and Algorithms*. Technology series. Sams, 2000 edition, Novermber 14 2000.