Performance Aware Reconfiguration of Software Systems

Moreno Marzolla[1] and Raffaela Mirandola[2]

[1] Università di Bologna, Dipartimento di Scienze dell'Informazione
Mura Anteo Zamboni 7, I-40127 Bologna, Italy
`marzolla@cs.unibo.it`

[2] Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, I-20133 Milano, Italy
`mirandola@elet.polimi.it`

Abstract. In this paper we address the problem of building a scalable component-based system by means of dynamic reconfiguration. Specifically, we consider the system response time as the performance metric; we assume that the system components can be dynamically reconfigured to provide a degraded service with lower response time. Each component operating at one of the available quality levels is assigned a *utility*. Higher quality levels are associated to higher utility. We propose an approach for performance-aware reconfiguration of degradable software systems called PARSY (Performance Aware Reconfiguration of software SYstems). PARSY tunes individual components in order to maximize the system utility with the constraint of keeping the system response time below a pre defined threshold. PARSY uses a closed Queueing Network model to select the components to upgrade or degrade.

1 Introduction

The Component Based software development paradigm allows complex systems to be built by assembling a number of independent components, each one providing a specific functionality. This paradigm has many advantages, including the ability of reducing development costs by allowing components reuse.

One crucial problem is building the system such that non functional requirements, such as reliability or performance, can be satisfied as well. This is difficult for several reasons. First, it is not easy to infer properties of the whole system by considering its components in isolation. Furthermore, performance depends on the external workload: even if the system has been implemented to sustain the expected workload, there may be unexpected variabilities which were not accounted for, and cause sporadic slowdowns. Thus, in order to ensure that performance related non functional requirements are satisfied, both proper capacity planning and some form of adaptation are necessary.

In this paper we focus on a specific performance metric, which is the *system response time*. This is a useful metric, because it can be easily measured, and because it impacts directly on users of the system. The system response time increases with the number of users concurrently accessing the system, unless the system adopts appropriate strategies to reconfigure itself in order to cope with spikes in the workload.

One commonly used solution is based on dynamic scalability using more physical resources. For example, an E-commerce Web site might react to an increased load by deploying the front-end across more Web servers, and load-balancing the requests among the available servers [1]. However, there are situations where this approach is not practical. Not every system can scale by simply adding more resources; furthermore, it could be difficult to react quickly to spikes in the workload, as allocating new resources and starting new application instances is not instantaneous.

In this paper we propose an approach, called Performance Aware Reconfiguration of software SYstems (PARSY), whose goal is the reduction of the system response time through a performance-aware degradation of the application, driven by the solution of performance models at runtime. Specifically, we consider a component-based software system, where some of the components can provide a *degraded* (but still acceptable) service with reduced response time. For example, let us go back to the example of E-commerce site under heavy load. Instead of relying on more server instances, the system might provide a degraded service for some non critical components. For example, the system could show heavily compressed images to customers in order to cut transfer time. As another example, the system might switch to a faster, but less precise, algorithm for searching products in the catalog which might retrieve also items not strictly related to the user query. The decision of which components can be degraded while still providing an acceptable service quality, is of course application-dependent.

The system administrators define a maximum allowed system response time. The system is enhanced with a monitoring component which identifies violations of the response time constraint. When violations occur, PARSY uses a Queueing Network (QN) performance model to decide which component of the system should be degraded. As the workload fluctuates, PARSY is able to degrade or upgrade individual components in order to satisfy, if possible, the response time constraint. The system administrators can associate weights to each configuration, such that PARSY can choose to degrade less important components first.

This paper is structured as follows: in Section 2 we give a high level overview of PARSY. In Section 3 we formally define the optimization problem we are addressing. In Section 4 we describe how a solution to the optimization problem can be efficiently computed. In Section 5 we evaluate the solution algorithm on some test instances, while a short survey of the related works is presented in Section 6. Finally, conclusions and future improvements are discussed in Section 7.

2 PARSY Overview

PARSY is an approach capable of driving the dynamic degradation of software systems. PARSY uses models at runtime to decide how the system can be upgraded or degraded. Specifically, the goal of PARSY is to selectively degrade or upgrade individual components in a component-based software system such that the estimated overall response time R is below a predefined threshold $R_{\max}$. PARSY tries to degrade less important components first, where the "importance" of each component can be defined by the system administrators.

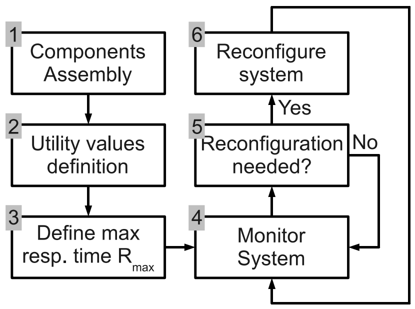

Fig. 1. PARSY Process Overview

The PARSY process is shown in Figure 1, and consists of the following steps:

1. PARSY starts by considering as input an architectural model of the system. The system is made of K components $\mathcal{C}_1, \ldots, \mathcal{C}_K$. Component $\mathcal{C}_k$ can be configured to provide service at different quality levels $1, \ldots, L_k$ (1 denotes the worst quality level, and $L_k \geq 1$ denotes the best quality level). Quality levels directly impact the execution time of components: the higher the quality level, the longer it takes to each component to compute the result. When a component is degraded, it produces a result more quickly; however, the result could be only an approximation of the correct one; if $\mathcal{C}_k$ can not be degraded, we let $L_k = 1$. A system configuration $\boldsymbol{\ell}$ is a vector $(\ell_1, \ldots, \ell_K)$ such that ℓ_k denotes the quality level of component $\mathcal{C}_k$.
2. Each component $\mathcal{C}_k$ operating at quality level j is labeled with a positive *utility value* $\mathrm{UT}(k, j)$. Utility values are defined by the system administrators; roughly speaking, utility values tell PARSY how much "important" a component is.
3. In this step the maximum response time $R_{\max}$ is defined. This parameter is strictly application-dependent, and it is normally defined early in the software development process as part of the non functional requirements.
4. The system is enhanced with a monitoring component, which continuously measures the average system response time.
5. If the response time is above or below the threshold, a system reconfiguration can be triggered. It is important to ensure that reconfigurations are not too frequent, so that the system has enough time to settle down.
6. A system reconfiguration involves upgrading/degrading one or more components. A QN performance model is used to evaluate different configurations.

Specifically, the QN model is used to solve the optimization problem of identifying the configuration with maximum utility, subject to the constraint that the estimated system response time is below the threshold $R_{\max}$.

Steps 1–3 are described in Section 3, while steps 4–6 are detailed in Section 4. Numerical examples of the PARSY application are then illustrated in Section 5.

3 Assumptions and Problem Definition

In PARSY we consider a software system made of K components $\mathcal{C}_1, \ldots, \mathcal{C}_K$. Each component $\mathcal{C}_k$ can be configured to provide service at different quality levels $1, \ldots, L_k$ (1=worst, L_k=best). The idea is that each component can be degraded to provide sub-optimal service faster. So the system exhibits a tradeoff between fast but less accurate computations, and slow but accurate computations.

In the following we assume that each component $\mathcal{C}_k$ performs a single operation. We further assume that all components are independent, in the sense that each one is hosted on a different physical resource. Note that both these limitations can easily be addressed. A single component implementing multiple operations can be viewed as a set of "simple" components–each one implementing a single operation–sharing the same physical resource. Then, multiple components sharing the same resource can be modeled by taking their aggregate service demands, as will be described in a few moments.

A *system configuration* is a vector $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_K)$ such that for each $k = 1, \ldots, K$, $\ell_k \in \{1, \ldots, L_k\}$; this indicates that component $\mathcal{C}_k$ is operating at level ℓ_k. We denote with $D(k, j)$ the *mean service demand* of component $\mathcal{C}_k$ when operating at level $j \in \{1, \ldots, L_k\}$. We require that, for each component $\mathcal{C}_k$, $0 < D(k, 1) < D(k, 2) < \ldots < D(k, L_k)$, that is, service demands for progressively degraded quality levels are strictly monotonically decreasing. Note that if multiple components, say $\mathcal{C}_{i_1}, \mathcal{C}_{i_2}, \ldots \mathcal{C}_{i_m}$ share the same physical resources, they can be aggregated into a single component whose service demand is $D(i_1, \ell_{i_1}) + D(i_2, \ell_{i_2}), + \ldots + D(i_m, \ell_{i_m})$.

In PARSY Step 2 we define a positive *utility value* $\mathrm{UT}(k, j)$ for component $\mathcal{C}_k$ operating at quality level j. For each $\mathcal{C}_k$ we require that $0 < \mathrm{UT}(k, 1) < \mathrm{UT}(k, 2) < \ldots < \mathrm{UT}(k, L_k)$. The utility value allows system administrators to define weights associated with quality levels, such that unimportant services are considered for degradation before important ones.

With a slight abuse of notation, the utility of system configuration $\boldsymbol{\ell}$ is defined as the sum of utilities of each component:

$$\mathrm{UT}(\boldsymbol{\ell}) = \sum_{k=1}^{K} \mathrm{UT}(k, \ell_k) \tag{1}$$

The initial system configuration is $\boldsymbol{\ell} = (L_1, \ldots, L_K)$, such that all components operate at the best quality level.

The goal of PARSY (Step 3) is to selectively upgrade or degrade some of the components such that the estimated system response time is kept below a

Table 1. Symbols used in this paper

$\mathcal{C}_1, \ldots, \mathcal{C}_K$	Components
$R_{\max}$	Maximum allowed system response time
$\hat{X}$	Measured system throughput
$\hat{R}$	Measured system response time
N	Number of concurrent users (computed according to Eq. (4))
$\boldsymbol{\ell} = (\ell_1, \ldots, \ell_K)$	System configuration (component $\mathcal{C}_k$ is operating at level ℓ_k)
L_k	Number of quality levels offered by component $\mathcal{C}_k$
$R(N; \boldsymbol{\ell})$	Estimated system response time at configuration $\boldsymbol{\ell}$ with N requests
$D(k, j)$	Average service demand of component $\mathcal{C}_k$ at quality level j
$\mathrm{UT}(k, j)$	Utility of component $\mathcal{C}_k$ when operating at quality level j

predefined threshold $R_{\max}$ and the system utility is the highest possible. In other words, we seek a solution to the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & \mathrm{UT}(\boldsymbol{\ell}) \\ \text{subject to} \quad & R(N; \boldsymbol{\ell}) < R_{\max} \\ & \ell_k \in \{1, \ldots, L_k\} \quad k = 1, \ldots, K \end{aligned} \tag{2}$$

where $R(N; \boldsymbol{\ell})$ is the estimated system response time with configuration $\boldsymbol{\ell}$, when there are N users. We are seeking the configuration which maximizes the total utility and keeps the estimated response time below the threshold $R_{\max}$.

Finding an exact solution to (2) is computationally difficult, as (2) can be viewed as an instance of the multiple-choice knapsack problem [2]. Finding the optimal solution requires the estimation of the response time for all possible configurations, which takes time $O(f(N, K) \prod_{k=1}^{K} L_k)$, where $f(N, K)$ is the cost of computing $R(N; \boldsymbol{\ell})$.

Also, observe that the problem (2) could even have no solution at all. In fact, the lowest possible response time can be achieved when the system is in its configuration $(1, \ldots, 1)$ (all components have been degraded as much as possible). A basic result from queueing theory states that the response time $R(N; 1, \ldots, 1)$ is asymptotically bounded by [3]:

$$R(N; 1, \ldots, 1) > N \max_k \{D(k, 1)\} \tag{3}$$

Thus, from (3) we conclude that, for any configuration $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_K)$, $\lim_{N \to \infty} R(N; \boldsymbol{\ell}) = \infty$, which means that for sufficiently high loads the constraint $R(N; \boldsymbol{\ell}) < R_{\max}$ can not be satisfied. In this case, PARSY will select the configuration $(1, \ldots, 1)$ as solution of the optimization problem (2).

Table 1 summarizes the symbols used in this paper.

4 Solving the Optimization Problem

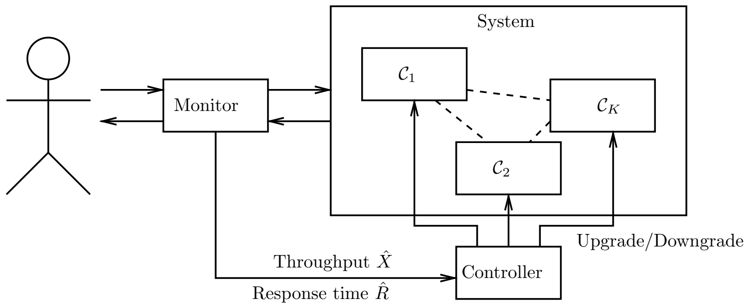

Fig. 2. System architecture, which includes the monitor and the controller

We now propose a practical way to find an approximate solution to the problem (2) above. The main points of PARSY are the following:

– We enhance the system with a *monitoring component* which triggers a reconfiguration whenever the measured system response time deviates from the threshold $R_{\max}$ (Step 4); this is described in detail in Section 4.1.
– We use a closed QN model to estimate response time $R(N; \boldsymbol{\ell})$ for a given system configuration $\boldsymbol{\ell}$ (Step 5). The computation of estimated response time with the QN model is used to find a solution to the optimization problem as will be illustrated in Section 4.2.

4.1 Identifying Reconfiguration Times

To perform the reconfiguration (Step 6) the first problem we address is how to decide when a reconfiguration should occur. To do so, we enhance the software system with a monitoring component (see Fig. 2). The monitor is a passive observer that measures the system response time $\hat{R}$ and throughput $\hat{X}$ (the use of throughput will be illustrated shortly). Then, the monitor notifies a separate component (the *controller*) when a reconfiguration should take place.

Specifically, if the measured response time is less than the threshold ($\hat{R} < R_{\max}$), the monitor notifies the controller to trigger a possible upgrade of one or more components; if the measured response time is greater than the threshold ($\hat{R} > R_{\max}$) the monitor notifies the controller to trigger a possible downgrade of some components.

Attention must be paid to avoid unnecessary reconfigurations when the measured response time $\hat{R}$ bounces above and below the threshold $R_{\max}$. A common approach to deal with this situation is to trigger a reconfiguration after the event $\hat{R} > R_{\max}$ (resp. $\hat{R} < R_{\max}$) has been observed multiple consecutive times. Alternatively, we can define two thresholds $R_{\min}$ and $R_{\max}$ such that an upgrade is triggered when $\hat{R} < R_{\min}$, and a downgrade is triggered when $\hat{R} > R_{\max}$. Furthermore, it is important to wait for the system to settle down after a reconfiguration. For one recent result, see [4].

4.2 Finding a New Configuration

We now describe how the controller identifies a new system configuration. The controller can estimate the system response time for different configurations by using a single-class, closed QN model. Each queueing center in the model represents a system component. We assume that the QN model has product-form solution, which in general could not be true for the system being modeled. However, the assumption of product-form solution allows us to solve the QN model efficiently; this is important because PARSY needs to reconfigure the system on-line.

The QN model contains K service centers, where center k corresponds to component $\mathcal{C}_k$. If the system configuration is $\boldsymbol{\ell} = (\ell_1, \ldots, \ell_K)$, then the service demand of queueing center k is $D(k, \ell_k)$.

To analyze the closed QN model, we need the number N of requests in the system at the time of reconfiguration. N can be computed from the observed values $\hat{X}$ and $\hat{R}$ using Little's law [5]:

$$N = \hat{X}\hat{R} \tag{4}$$

Now that we have all parameters for the QN model, we can estimate $R(N; \boldsymbol{\ell})$ using Mean Value Analysis (MVA) [6] according to the pseudo-code shown in Algorithm 3 in the Appendix. The computational complexity of MVA is $O(NK)$, where N is the number of requests and K is the number of service centers (which is equal to the number of components in the system). A faster way to estimate the response time is to compute asymptotic upper and lower bounds on the response time of QN model. For example, bounds on the response time can be computed in time $O(K)$ [3]; of course, performance bounds do not provide the exact value for $R(N; \boldsymbol{\ell})$, but only upper and lower limits. We can then estimate the system response time as the average of the upper and lower limit (see Algorithm 4).

We now describe how the performance model is used to identify a new system configuration. Specifically, we describe an approximate solution technique, based on the greedy paradigm, which identifies a feasible solution to (2). The solution technique works as follows:

- If $\hat{R} > R_{\max}$, we identify components which can be degraded. We keep degrading components until the estimated system response time, as computed using the QN model, becomes less than the threshold $R_{\max}$.
- If $\hat{R} < R_{\max}$, we identify components which can be upgraded. We keep upgrading components as long as the estimated system response time remains less than the threshold $R_{\max}$.

Let us analyze the two cases in detail.

Degrading Components When $\hat{R} > R_{\max}$, a reconfiguration is triggered by executing Algorithm 1. This is a greedy algorithm which, at each step, selects a component to degrade. The algorithm stops either when (i) the estimated response time is below the threshold, or (ii) all components have been degraded at quality level 1, so that no further degradation is possible.

Algorithm 1 Degrade configuration

```
Require: $\boldsymbol{\ell}$ current system configuration
Require: $N$ number of requests in the system
Require: $D(k, j)$ service demand of component $\mathcal{C}_k$ operating at quality level $j$
Require: $\mathrm{UT}(k, j)$ utility of component $\mathcal{C}_k$ operating at quality level $j$
Ensure: $\boldsymbol{\ell}^{\text{new}}$ is the new system configuration
  $\boldsymbol{\ell}^{\text{new}} \leftarrow \boldsymbol{\ell}$
  $C \leftarrow \{k \mid \ell_k^{\text{new}} > 1\}$          {Candidate set of components which can be degraded}
  while $C \neq \emptyset$ do
    $B \leftarrow \arg\max_k \{D(k, \ell_k^{\text{new}})/\mathrm{UT}(k, \ell_k^{\text{new}}) \mid k \in C\}$
    $\ell_B^{\text{new}} \leftarrow \ell_B^{\text{new}} - 1$          {Degrade $\mathcal{C}_B$}
    Compute $R(N; \boldsymbol{\ell}^{\text{new}})$ using Algorithm 3 (MVA)
    if $R(N; \boldsymbol{\ell}^{\text{new}}) < R_{\max}$ then
      Break          {Downgrade complete}
    else
      $C \leftarrow \{k \mid \ell_k^{\text{new}} > 1\}$          {Recompute the candidate set}
  Return $\boldsymbol{\ell}^{\text{new}}$
```

At each iteration, the component to be degraded $\mathcal{C}_B$ is the one for which the ratio $D(B, \ell_B)/\mathrm{UT}(B, \ell_B)$ is maximum, where ℓ_B is the quality level of $\mathcal{C}_B$ *after it has been degraded*. The idea is to degrade the component with both a high service demand and a low utility. In queueing theory it is well known that the center with maximum demand is the bottleneck device; here we also take into account the utility of the degraded component.

Upgrading Components Algorithm 2 is used to upgrade components as long as the estimated system response time $R(N; \boldsymbol{\ell})$ remains below the threshold $R_{\max}$. Again, we use a greedy approach in which the component $\mathcal{C}_U$ to be upgraded is chosen at each iteration. Let ℓ_U be the quality level of component $\mathcal{C}_U$ before the reconfiguration. Then, $\mathcal{C}_U$ is chosen to satisfy the following two properties:

– After upgrading $\mathcal{C}_U$ at configuration $\ell_U + 1$, the new estimated system response time is below the threshold $R_{\max}$;
– $\mathcal{C}_U$ is the component whose upgrade provides the maximum utility with the minimum increase in system response time.

This approach is similar to the greedy algorithm for solving the knapsack problem [2], where items are tried in order of decreasing unitary value.

4.3 Computational Complexity

Both Algorithms 1 and 2 execute a number of iterations in which a single component is downgraded/upgraded; in particular, at each iteration one component $\mathcal{C}_k$ can be upgraded from level ℓ_k to level $\ell_k + 1$, or degraded from level ℓ_k to level $\ell_k - 1$. The worst case happens when the whole system is degraded from configuration $(L_1, \ldots, L_K)$ to configuration $(1, \ldots, 1)$, or the other way. Thus, in the worst case at most $\sum_{k=1}^{K} L_k$ iterations are performed. The cost of each iteration

Algorithm 2 Upgrade configuration

```
Require: $\boldsymbol{\ell}$ current system configuration
Require: $N$ number of requests in the system
Require: $D(k,j)$ service demand of component $\mathcal{C}_k$ operating at quality level $j$
Require: $\mathrm{UT}(k,j)$ utility of component $\mathcal{C}_k$ operating at quality level $j$
Ensure: $\boldsymbol{\ell}^{\text{new}}$ is the new system configuration
  $\boldsymbol{\ell}^{\text{new}} \leftarrow \boldsymbol{\ell}$
  $C \leftarrow \{k \mid \ell_k^{\text{new}} < L_k\}$        {Candidate set of components which can be upgraded}
  while $C \neq \emptyset$ do
    $U \leftarrow \arg\min_k \{D(k, \ell_k^{\text{new}}+1)/\mathrm{UT}(k, \ell_k^{\text{new}}+1) \mid k \in C\}$
    $\ell_U^{\text{new}} \leftarrow \ell_U^{\text{new}} + 1$        {Try to upgrade $\mathcal{C}_U$}
    Compute $R(N; \boldsymbol{\ell}^{\text{new}})$ using Algorithm 3 (MVA)
    if $R(N; \boldsymbol{\ell}^{\text{new}}) > R_{\max}$ then
      $\ell_U^{\text{new}} \leftarrow \ell_U^{\text{new}} - 1$        {Rollback configuration for $\mathcal{C}_U$}
      $C \leftarrow C \setminus \{U\}$
    else
      $C \leftarrow \{k \mid \ell_k^{\text{new}} < L_k\}$        {Recompute the candidate set}
  Return $\boldsymbol{\ell}^{\text{new}}$
```

is dominated by the cost $f(N,K)$ of evaluating $R(N;\boldsymbol{\ell})$. If the system response time is estimated using the MVA Algorithm 3, we have that $f(N,K) = O(NK)$ which implies that the computational complexity of Algorithms 1 and 2 is $O(NK\sum_k L_k)$. If the system response time is estimated by computing upper and lower bounds R^+ and R^- [3] and letting $R(N;\boldsymbol{\ell}) \approx (R^+ + R^-)/2$, then we have $f(N,K) = O(K)$. In this case we can reduce the computational complexity of Algorithms 1 and 2 to $O(K\sum_k L_k)$, which is also independent from the number of requests N.

5 Numerical Example

In this section we assess the effectiveness of PARSY by means of a set of synthetic test cases which is numerically evaluated. We consider a software system with K components such that each component can operate at L different quality levels. We experiment with multiple combinations of K and L: we use $K = 10, 20, 30, 50$ and $L = 2, 3, 5$. Service demands $D(k,j)$ and utility $\mathrm{UT}(k,j)$ of component $\mathcal{C}_k$ operating at quality level j, for all $k = 1, \ldots, K$, $j = 1, \ldots, L$, are randomly generated when each model is created.

We evaluate each system for $T = 200$ time steps. The number of users (requests) N_t at step $t = 1, \ldots, T$ is produced using a random walk model. For each experiment the threshold $R_{\max}$is defined as:

$$R_{\max} = \max\{R(N_t; 1, \ldots, 1) \mid t = 1, \ldots, T\} \qquad (5)$$

that is, $R_{\max}$is the maximum system response time when all components operate at quality level 1 (worst). This ensures that there exists a configuration such that the system response time is kept below $R_{\max}$for all values of N_t.

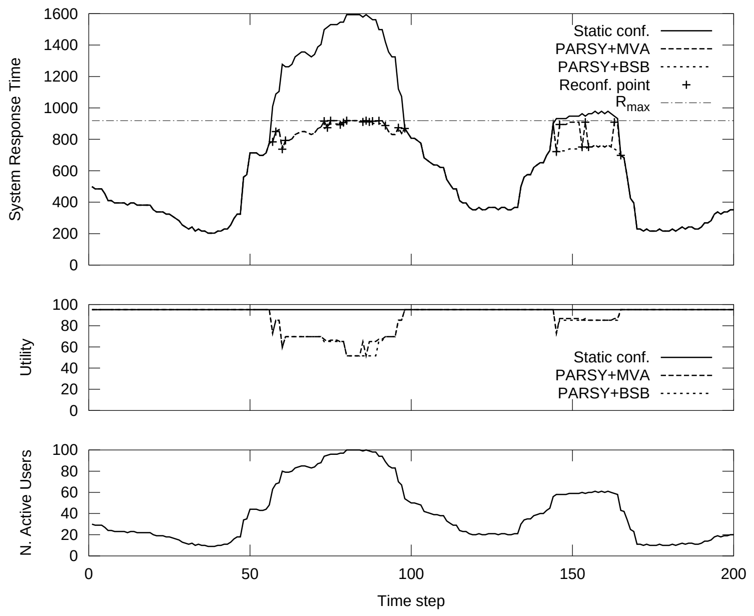

Fig. 3. Response time, utility and number of requests for $K = 10$, $L = 2$.

We implemented Algorithms 1 and 2 in GNU Octave [7]. Initially, all components are set at the best quality level. We tested two different techniques to estimate $R(N; \boldsymbol{\ell})$: (i) using MVA (Algorithm 3), and (ii) using Balanced System Bounds (BSB) (Algorithm 4).

PARSY is executed on-line, that is it finds a new configuration at time step t by considering only the configuration at the previous step $t-1$ and the number of currently active requests N_t. The observed system response time $\hat{R}(t)$ at time t is computed using MVA. The value of $\hat{R}(t)$ is then used to decide whether the configuration should be upgraded or downgraded, as described in Section 4. For each configuration we also compute the utility according to (1).

Figure 3 shows an example of the results for a system with $K = 10$ components operating at $L = 2$ quality levels. The top part of the plot shows the observed system response time for the static configuration $(L, \ldots, L)$ (solid line), PARSY+MVA and PARSY+BSB (dashed lines). Reconfiguration points are also shown. The middle part of Figure 3 shows the utility over time for the static system (solid line), as well as using PARSY (dashed lines). Note that the utility of the system with configuration $(L, \ldots, L)$ is, by construction, an upper bound of the utility of any other valid configuration. Finally, the bottom part of Figure 3 shows the number of users N_i at time step i.

In order to compare PARSY+MVA and PARSY+BSB we consider two metrics: the *total utility* UT and the *overflow response time* ΔR. The total utility

Table 2. Results of all experiment sets. K is the number of components; L is the number of quality levels; UT is the total utility and ΔR the response time overflow.

		No Adaptation		**PARSY+MVA**		**PARSY+BSB**	
K	L	UT	ΔR	UT	ΔR	UT	ΔR
5	2	5884.17	5890.15	5712.94	0.00	5705.07	0.00
5	3	16363.16	9962.44	15044.46	0.00	15011.59	0.00
5	5	20757.87	169979.45	13316.75	0.00	13214.68	0.00
10	2	19040.52	20594.15	17694.10	0.00	17615.86	0.00
10	3	32711.11	65785.59	27722.41	0.00	27490.38	0.00
10	5	49114.04	182799.96	34705.66	0.00	34356.43	30.83
20	2	39493.98	22550.31	37410.67	0.00	37204.16	0.00
20	3	63899.70	67996.03	55610.41	0.00	55120.90	0.00
20	5	94426.88	137734.24	70743.20	0.00	69628.72	0.00
50	2	92983.84	35439.53	85082.02	0.00	83692.37	0.00
50	3	141622.46	90643.03	114941.53	0.00	112057.56	0.00
50	5	240703.53	252987.50	128606.66	0.00	126259.66	55.45

is the sum of utilities of all system configurations produced by PARSY. The overflow response time is defined as the sum of $(\hat{R}(t) - R_{\max})$ over all t for which $\hat{R}(t) > R_{\max}$. Intuitively, the overflow response time is the area which lies above the line $y = R_{\max}$ and below $y = \hat{R}(t)$. Observe that the choice of $R_{\max}$ (see (5)) ensures that an optimal reconfiguration algorithm is able to achieve $\Delta R = 0$.

We report in Table 2 the results of all experiments; for a better visual comparison, the same results are shown in Figure 4. We observe that PARSY is very effective in reducing the response time overflow; at the same time the total utility is kept as a fraction of the maximum possible value. It is interesting to observe that, in the considered test cases, PARSY+BSB produces only marginally worse reconfigurations than those produced by PARSY+MVA. This means that configurations identified by PARSY+BSB have on average slightly lower utility, while the response time overflow in both cases is basically zero. The difference in the utility value is quite small, and is compensated by the fact that BSB are faster to compute, and thus are better suited for very large systems with many components. Note that efficiency of the reconfiguration algorithms is hardly an issue for small to medium size system. Our tests have been performed on a Linux PC (kernel 2.6.24) with an AMD Athlon 64 X2 Dual Core processor 3800+ with 2GB of RAM, using GNU Octave version 3.2.3. On this system, a single reconfiguration step requires a fraction of a second using MVA, even for the largest test case with $K = 50$ components and $L = 5$ levels; note that in this case the total number of possible configurations is 5^{50}.

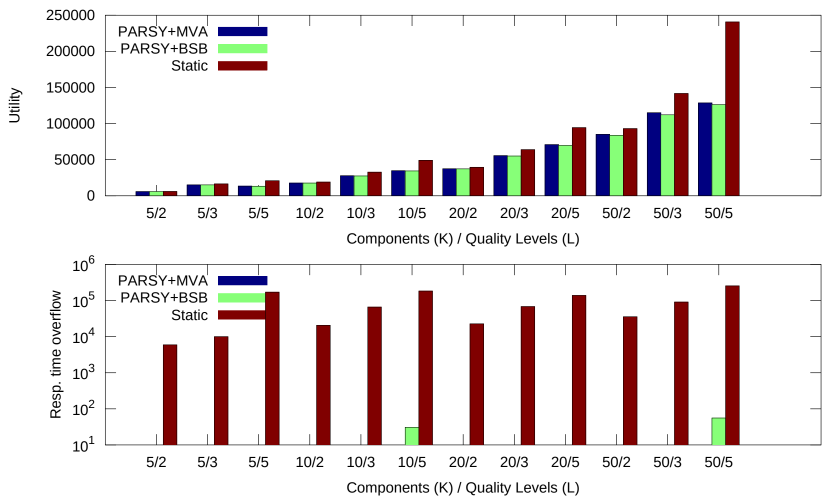

Fig. 4. Utility and response time overflow for the experiments; labels on the horizontal axes denote the parameters K/L used for that experiment. The response times overflow is shown in log scale.

6 Related Works

In the last years, as outlined in [8], the topic of reconfigurable and self-adaptive computing systems has been studied in several communities and from different perspectives. The *autonomic computing* framework is a notable example of general approach to the design of such systems [9, 10]. Hereafter, we focus on works appeared in the literature dealing with the self-adaptation of software systems to guarantee the fulfillment of QoS requirements. Specifically, we focus on works that make use of models to perform this step.

GPAC (General-Purpose Autonomic Computing), for example, is a tool-supported methodology for the model-driven development of self-managing IT systems [11]. The core component of GPAC is a generic autonomic manager capable of augmenting existing IT systems with a MAPE autonomic computing loop. The GPAC tools and the probabilistic model checker PRISM [12] are used together successfully to develop autonomic systems involving dynamic power management and adaptive allocation of data-center resources [13]. KAMI [14] is another framework for model evolution by runtime parameter adaptation. KAMI focuses on Discrete Time Markov Chain models that are used to reason about non-functional properties of the system. The authors adapt the QoS properties of the model using Bayesian estimations based on runtime information, and the updated model allows the verification of QoS requirements. The approach presented in [15] considers the QoS properties of a system in a web-service environment. The authors provide a language called SLAng, which allows the specification of QoS to be monitored. The Models@Run.Time approach [16] proposes to leverage software models and to extend the applicability of model-driven engineering techniques to the runtime environment to enhance systems with dynamic adapting capabilities. In [17], the authors use an architecture-based approach to support dynamic adaptation. Rainbow [18] also updates architectural models to detect inconsistencies and in this way it is able to correct certain types of faults. A different use of models at runtime for system adaptation is taken in [19]. The authors update the model based on execution traces of the system. In [20] the authors describe a methodology for estimation of model parameters through Kalman filtering. This work is based on a continuous monitoring that provides run-time data feeding a Kalman filter, aimed at updating the performance model.

In the area of service-based systems (SBS), devising QoS-driven adaptation methodologies is of utmost importance in the envisaged dynamic environment in which they operate. Most of the proposed methodologies for QoS-driven adaptation of SBS address this problem as a service selection problem (e.g., [21–24]). Other papers have instead considered service-based adaptation through workflow restructuring, exploiting the inherent redundancy of SBS (e.g., [25–27].) In [28] a unified framework is proposed where service selection is integrated with other kinds of workflow restructuring, to achieve a greater flexibility in the adaptation.

The works closest to our approach are [29–31]. In [29], the authors propose a conceptual model dealing with changes in dynamic software evolution. Besides, they apply this model to a simple case study, in order to evaluate the effectiveness of fine-grained adaptation changes like service-level degrading/upgrading action considering also the possibility to perform actions involving the overall resource management. The approach proposed in [30] deals with QoS-based reconfigurations at design time. The authors propose a method based on evolutionary algorithms where different design alternatives are automatically generated and evaluated for different quality attributes. In this way, the software architect is provided with a decision making tool enabling the selection of the design alternatives that best fits multiple quality objectives. Menascé et al. [31] have developed the SASSY framework for generating service-oriented architectures based on quality requirements. Based on an initial model of the required service types and their communication, SASSY generates an optimal architecture by selecting the best services and potentially adding patterns such as replication or load balancing.

With respect to existing approaches, PARSY lies in the research line fostering the usage of models at runtime to drive the QoS-based system adaptation. The proposed approach uses two very efficient modeling and analysis techniques that can then be used at runtime without undermining the system behavior and its overall performance.

7 Conclusions

In this paper we presented PARSY, a framework for runtime performance aware reconfiguration of component-based software systems. The idea underlying PARSY is to selectively degrade and upgrade system components to guarantee that the overall response time does not exceed a predefined threshold. The capability of driving this dynamic degradation is achieved through the introduction of a monitoring component that triggers a reconfiguration whenever the measured response time exceeds the threshold, and the use of a QN model to estimate, at runtime, the response time of various reconfiguration scenarios. The response times are used to feed the optimization model whose solution gives the system configuration which maximizes the total utility while keeping the response time below the threshold.

The methodology proposed in this paper can be improved along several directions. We are extending the framework to include multiple classes of requests, and to be able to deal with multiple components sharing the same physical resource. We are also exploring the use of forecasting techniques as a mean to trigger system reconfiguration in a proactive way. Another direction that deserve further investigation is the use of different numerical techniques for an efficient and accurate solution of the optimization problem. Finally, we are working on the implementation of our methodology on a real testbed, to assess its effectiveness through a more comprehensive set of real experiments.

References

1. Chieu, T.C., Mohindra, A., Karve, A.A., Segal, A.: Dynamic scaling of web applications in a virtualized cloud computing environment. In: E-Business Engineering, IEEE International Conference on, Los Alamitos, CA, USA, IEEE Computer Society (2009) 281–286
2. Martello, S., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley and Sons (1990)
3. Zahorjan, J., Sevcick, K.C., Eager, D.L., Galler, B.I.: Balanced job bound analysis of queueing networks. Comm. ACM **25**(2) (February 1982) 134–141
4. Casolari, S., Colajanni, M., Lo Presti, F.: Runtime state change detector of computer system resources under non stationary conditions. In: Proc. 17th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomunication Systems (MASCOTS 2009), London (September 2009)
5. Little, J.D.C.: A proof for the queuing formula: $L = \lambda W$. Operations Research **9**(3) (1961) 383–387
6. Reiser, M., Lavenberg, S.S.: Mean-value analysis of closed multichain queuing networks. Journal of the ACM **27**(2) (April 1980) 313–322
7. Eaton, J.W.: GNU Octave Manual. Network Theory Limited (2002)
8. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., eds.: Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]. In Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., eds.: Software Engineering for Self-Adaptive Systems. Volume 5525 of Lecture Notes in Computer Science., Springer (2009)
9. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer **36**(1) (2003) 41–50
10. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing–degrees, models, and applications. ACM Comput. Surv. **40**(3) (2008)
11. Calinescu, R.: General-purpose autonomic computing. In Denko, M.K., Yang, L.T., Zhang, Y., eds.: Autonomic Computing and Networking. Springer (2009) 3–30

12. PRISM web site `http://www.prismmodelchecker.org/`.
13. Calinescu, R., Kwiatkowska, M.: Using quantitative analysis to implement autonomic it systems. In: ICSE '09: Proceedings of the 31st International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2009) 100–110
14. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by runtime parameter adaptation. In: Proc. 31st International Conference on Software Engineering (ICSE09), IEEE Computer Society (2009) 111–121
15. Raimondi, F., Skene, J., Emmerich, W.: Efficient online monitoring of web-service slas. In: SIGSOFT FSE, ACM (2008) 170–180
16. Morin, B., Barais, O., Jézéquel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. IEEE Computer **42**(10) (2009) 44–51
17. Taylor, R.N., Medvidovic, N., Oreizy, P.: Architectural styles for runtime software adaptation. In: WICSA/ECSA, IEEE (2009) 171–180
18. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B.R., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer **37**(10) (2004) 46–54
19. Maoz, S.: Using model-based traces as runtime models. IEEE Computer **42**(10) (2009) 28–36
20. Zheng, T., Woodside, C.M., Litoiu, M.: Performance model estimation and tracking using optimal filters. IEEE Trans. Soft. Eng **34**(3) (2008) 391–406
21. Ardagna, D., Pernici, B.: Adaptive service composition in flexible processes. IEEE Trans. Soft. Eng **33**(6) (2007) 369–384
22. Canfora, G., Penta, M.D., Esposito, R., Villani, M.L.: A framework for QoS-aware binding and re-binding of composite web services. Journal of Systems and Software **81**(10) (2008) 1754–1769
23. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F.: Scalable service selection for web service composition supporting differentiated QoS classes. Technical Report RR-07.59, Dip. di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata (2007)
24. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H.: QoS-aware middleware for web services composition. IEEE Trans. Soft. Eng **30**(5) (2004) 311–327
25. Chafle, G., Doshi, P., Harney, J., Mittal, S., Srivastava, B.: Improved adaptation of web service compositions using value of changed information. In: ICWS, IEEE Computer Society (2007) 784–791
26. Guo, H., Huai, J., Li, H., Deng, T., Li, Y., Du, Z.: Angel: Optimal configuration for high available service composition. In: 2007 IEEE International Conference on Web Services (ICWS 2007), IEEE Computer Society (2007) 280–287
27. Harney, J., Doshi, P.: Speeding up adaptation of web service compositions using expiration times. In: WWW '07: Proceedings of the 16th international conference on World Wide Web, New York, NY, USA, ACM (2007) 1023–1032
28. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F., Mirandola, R.: Qos-driven runtime adaptation of service oriented architectures. In: ESEC/FSE '09: Proc. 7th joint meeting of the European softw. eng. conf. and the ACM SIGSOFT symp. on The foundations of softw. eng., ACM (2009) 131–140
29. Salehie, M., Li, S., Asadollahi, R., Tahvildari, L.: Change support in adaptive software: A case study for fine-grained adaptation. In: EASE '09: Proc. Sixth IEEE Conf. and Workshops on Engineering of Autonomic and Autonomous Systems, Washington, DC, USA, IEEE Computer Society (2009) 35–44

30. Martens, A., Koziolek, H., Becker, S., Reussner, R.: Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. In: Proc. first joint WOSP/SIPEW international conference on Performance engineering, New York, NY, USA, ACM (2010) 105–116
31. Menascé, D.A., Ewing, J.M., Gomaa, H., Malex, S., Sousa, J.P.: A framework for utility-based service oriented design in sassy. In: Proc. first joint WOSP/SIPEW int. conf. on Performance engineering, New York, NY, USA, ACM (2010) 27–36

A Estimation of the System Response Time

Algorithm 3 Estimation of $R(N, \boldsymbol{\ell})$ using MVA

```
Require: $N$ number of users
Require: $\boldsymbol{\ell} = (\ell_1, \ell_2, \dots \ell_K)$ current system configuration
Require: $D(k, j)$ service demand of component $\mathcal{C}_k$ operating at quality level $j$
Ensure: $R$ is the system response time
  for all $k = 1, 2, \dots K$ do
    $Q_k \leftarrow 0$
  for all $n = 1, 2, \dots N$ do
    for all $k = 1, 2, \dots K$ do
      $R_k \leftarrow D(k, \ell_k) \times (1 + Q_k)$        {Residence time at $\mathcal{C}_k$}
    $R \leftarrow \sum_{k=1}^{K} R_k$        {System response time}
    $X \leftarrow n/R$        {System throughput}
    for all $k = 1, 2, \dots K$ do
      $Q_k \leftarrow X R_k$        {Average number of requests at $\mathcal{C}_k$}
  Return $R$
```

Algorithm 4 Estimation of $R(N, \boldsymbol{\ell})$ using BSB

```
Require: $N$ number of users
Require: $\boldsymbol{\ell} = (\ell_1, \ell_2, \dots \ell_K)$ current system configuration
Require: $D(k, j)$ service demand of component $\mathcal{C}_k$ operating at quality level $j$
Ensure: $R$ is the system response time
  $D_{\max} \leftarrow \max\{D(k, \ell_k) \mid k = 1, \dots, K\}$
  $D_{\text{tot}} \leftarrow \sum_k D(k, \ell_k)$
  $R^- \leftarrow \max\{N D_{\max}, D_{\text{tot}}(1 + (N-1)/N)\}$        {Lower bound on response time}
  $R^+ \leftarrow D_{\text{tot}} + (N-1) D_{\max}$        {Upper bound on response time}
  $R \leftarrow (R^+ + R^-)/2$
  Return $R$
```