Online prediction: four case studies

Katja Gilly[1], Fabian Brosig[2], Ramon Nou[3], Samuel Kounev[2], and Carlos Juiz[4]

[1] Universidad Miguel Hernandez, 03202 Elche, Spain
katya@umh.es

[2] Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
{fabian.brosig,kounev}@kit.edu

[3] Barcelona Supercomputing Center, 08034 Barcelona, Spain
ramon.nou@bsc.es

[4] Universitat de les Illes Balears, 07004 Palma, Spain
cjuiz@uib.es

Abstract Current computing systems are becoming increasingly complex in nature and exhibit large variations in workloads. These changing environments create challenges to the design of systems that can adapt themselves while maintaining desired Quality of Service (QoS), security, dependability, availability and other non-functional requirements. The next generation of resilient systems will be highly distributed, component-based and service-oriented. They will need to operate in unattended mode and possibly in hostile environments, will be composed of a large number of interchangeable components discoverable at run-time, and will have to run on a multitude of unknown and heterogeneous hardware and network platforms. These computer systems will adapt themselves to cope with changes in the operating conditions and to meet the service-level agreements with a minimum of resources. Changes in operating conditions include hardware and software failures, load variation and variations in user interaction with the system, including security attacks and overwhelming situations. This self adaptation of next resilient systems can be achieved by first online predicting how these situations would be by observation of the current environment. This chapter focuses on the use of online predicting methods, techniques and tools for resilient systems. Thus, we survey online QoS adaptive models in several environments as grid environments, service-oriented architectures and ambient intelligence using different approaches based on queueing networks, model checking, ontology engineering among others.

1 Introduction

New resilient systems have to consider QoS variations that occur and then react to these changes online acting accordingly to maintain a certain Service Level Agreement (SLA). Consequently, these systems need to predict these variations found even at the risk of being wrong on a certain value.

Predictions are based on a model that has to be representative in the sense that it reflects the system's QoS-relevant behaviour. Typically, the user behaviour is an input of such a model. Thus, the user behaviour has to be predicted as well when obtaining model predictions to anticipate QoS problems. In the context of performance predictions, user behaviour prediction is often referred to as workload forecasting. For workload forecasting, established time series analysis techniques [3] are often used. For instance, Brown's quadratic exponential smoothing or general AutoRegressive - Moving Average (ARMA) models have been implemented in [8] and [5].

Concerning online performance prediction, in [11, 12], the authors describe a framework using analytic performance models in the design of self-configurable and self-managing computer systems. An general overview on performance models that can be evaluated efficiently, is provided in, e.g., [2]. Typically, these models are based on queuing networks and markov chains. A different approach is applied in [10], where the online performance prediction is based on a machine-learning approach.

In this chapter, we consider four different case studies in order to show how online prediction could help in this way to the resilience of systems. The first case study shows how detailed architecture-level performance models can be extracted and maintained automatically at run-time based on on- line monitoring data. Even though the current version of the extraction method is not 100% automated, and there are some prediction error yet, the case study demonstrated that the existing gap between low-level monitoring data and high-level performance models can be closed. In the second case, we augmented the Grid middleware with an online performance prediction mechanism that can be called at run-time to predict the Grid performance for a given resource allocation and load-balancing strategy, demonstrating the benefits of online performance prediction for run-time performance management. In the third example, we include an adaptive time slot scheduling based on a burstiness metric, that permits to control the monitoring frequency of the system depending on the burstiness levels detected by the algorithm. This means a considerable decrease of the overhead of the monitoring process, whose frequency can be adapted to the stress detected at the entry point of the system. This technique is used in the fourth case to build an admission control and load balancing algorithm that is based on throughput prediction for a Web system. These cases studies are just four individual examples, but they illustrate how on-line predictions increase the resilience of any kind of system performance problem. All of them are related one to the others, in several ways that coincide with three general questions to face off during their design: first, the necessity of gathering data from either monitoring or measurements in order to predict the future; second, the dynamicity of the on-line decisions based on partial temporal information and finally, the overhead of doing both procedures is the price to be paid in order to get the on-line predictions. The challenge in all cases is how to reduce overhead time as the QoS problem permits.

2 Automatic Model Extraction at Run-Time

As a proof-of-concept for automatic model extraction at run-time, we conducted a case study with a complex Java EE application. The case study shows how detailed architecture-level performance models can be extracted and maintained automatically at run-time based on online monitoring data [4]. The Java EE application we considered was a beta version of the new SPECjEnterprise2010 standard benchmark. We deployed the benchmark on Oracle WebLogic Server (WLS) and used the WebLogic Diagnostics Framework (WLDF) as a monitoring and instrumentation tool (Figure 1). The considered architecture-level performance model was the Palladio Component Model (PCM).

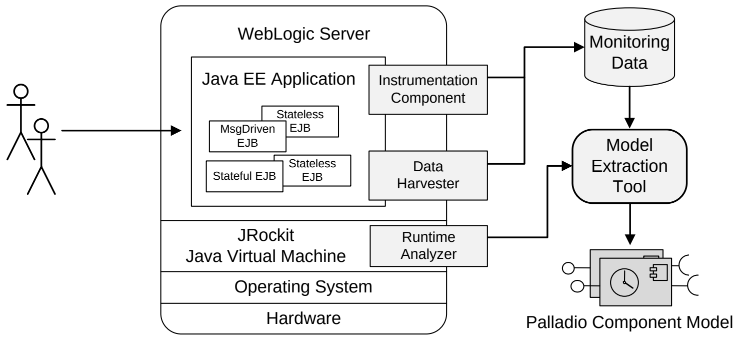

Figure 1. Model Extraction Tool Architecture

The PCM is a domain-specific modelling language for describing performance-relevant aspects of component-based software architectures [1]. In PCM, a component specification normally includes a definition of which interfaces the component provides and requires together with a set of *Resource Demanding Service Effect Specifications* (RDSEFFs). Each RDSEFF describes the performance-relevant internal behaviour of a provided component service in an abstract manner. The control flow and the resource consumption of the service can be modelled probabilistically as well as depending on the input parameters. Figure 2 shows a component service's RDSEFF in a notation similar to the notation of UML activity diagrams. The RDSEFF consists of an internal action abstracting component-internal resource demanding instructions, followed by a loop action containing a further internal action and an external call action to a required service. The loop iteration number of `LoopAct_985` is specified as a probability mass function (PMF). The PMF states that the loop iterates one time with a probability of 20% and ten times with a probability of 80%.

The extraction method for Java EE applications was implemented using the WLDF monitoring tool that is provided with Oracle WebLogic Server (WLS). The three main steps of the model extraction process are: i) the extraction of

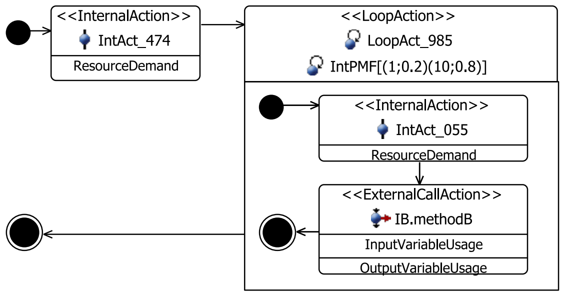

Figure 2. Example: Extracted RDSEFF

the application architecture, ii) the extraction of performance-relevant control flow and iii) the extraction of resource demands.

In the first step, the effective application architecture is extracted. The latter refers to the set of components and connections between components that are effectively used during operation. The components and connections are identified on the basis of trace data reflecting the observed call paths during execution. Based on the call paths, the effective connections among components can be determined, i.e., required interfaces of components can be bound to components providing the respective services. In the second extraction step, the tracing technique is applied to extract the performance-relevant control flow inside the components. We focus on monitoring the effective control flow and therefore extract probabilities of different call paths in contrast to extracting explicit parametric dependencies. Figure 2 shows an RDSEFF that has been extracted from trace data generated by WLDF. To estimate the resource demands of individual internal actions, we investigated two approaches: i) approximate resource demands with measured response times, ii) estimate resource demands based on measured utilization and throughput data. While the first approach is only applicable during phases of low resource utilization, i.e., <20%, the second approach can be applied during an observation period with medium to high load.

We applied the model extraction method to a beta version of the new SPECjEnterprise2010 benchmark. The benchmark workload is generated by an application that is modelled after a real-world business scenario. We deployed the benchmark in a system environment consisting of three machines. The Java EE application was deployed on an Oracle WebLogic Server (WLS) instance. As a database server (DBS), Oracle Database 11g was installed on the second machine. The benchmark driver was running on the third machine. The machines all have Intel Pentium Dual Core E2180 CPUs (2x2.0 GHz), 3 GB of RAM and are connected using a 1 GBit Ethernet.

To validate the extraction method, we compared predictions derived from the extracted PCM models with measurements on the real system. We considered two different models: i) *Model A* - PCM model in which resource demands were approximated with measured response times, ii) *Model B* - PCM model in which resource demands were estimated based on utilization and throughput data. We analysed the extracted models my means of simulation [1]. As performance metrics, we considered the average response times of business operations as well as the average utilization of the WLS CPU and the DBS CPU. We analysed scenarios under low load conditions, medium load conditions and high load conditions.

In the scenario we consider here, the workload consisted of the business operation ScheduleWorkOrder. Figure 3 shows the results. Predictions based on Model B perform slightly better than predictions based on Model A. For the highest considered throughput level, both models deliver no performance predictions. This is because the system as represented by the models is not able to sustain the injected load since the WLS CPU utilization is overestimated to be 100%. Both models overestimate the WLS CPU utilization while underestimating the DBS CPU utilization. The modelling prediction error for CPU utilization is mostly about 20%. The modelling prediction error for response times increases with the throughput level. The higher the CPU utilization, the bigger the impact of the overestimated WLS CPU demands on the predicted response times. We assume that the overestimation of the WLS CPU demands is due to the instrumentation overhead during resource demand extraction.

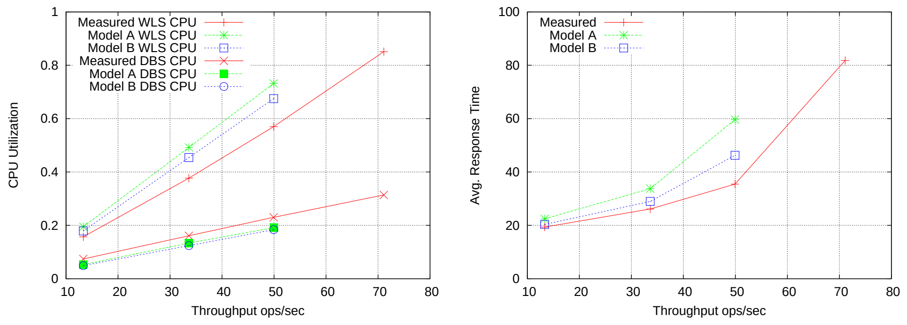

Figure 3. Validation of the ScheduleWorkOrder Performance Models

We considered a number of different scenarios, on the one hand, varying the operation mix and throughput level under which the PCM models were extracted, and on the other hand, varying the operation mix and throughput level for which performance predictions were made. The extracted models consisted of up to six components, eight RDSEFFs and 13 internal actions annotated with WLS CPU demand or DBS CPU demand estimations [4]. The results were similar to the ones presented here. The prediction error was between 20 and 30 percent. Even though the current version of the extraction method is not 100% automated, the case study demonstrated that the existing gap between low-level monitoring data and high-level performance models can be closed.

3 Autonomic QoS-Aware Grid Resource Managers

As a second proof-of-concept demonstrating the benefits of online performance prediction for run-time performance management, we conducted a case study of a SOA application running in a service-oriented Grid computing environment [14] [13]. The latter was implemented using the Globus Toolkit middleware which is based on open Web Services standards and can be seen as an incarnation of SOA. We augmented the Grid middleware with an online performance prediction mechanism that can be called at run-time to predict the Grid performance for a given resource allocation and load-balancing strategy. The online performance prediction mechanism was used as a basis for building a novel QoS-aware Grid resource manager architecture depicted in Figure 4. A *resource manager* is responsible for managing access to a set of Grid servers each offering some Grid services. The resource manager keeps track of the available Grid resources and mediates between clients and servers to make sure that SLAs are continuously satisfied. Before a Grid server can be used, it must register with the resource manager providing information on the services it offers, their resource requirements and the server capacity made available to the Grid. The Grid server must provide an architecture-level performance model that captures the information relevant to predicting the performance of the services it offers. For a client to be able to use a service, it must first send a *session request* to the resource manager. The session request specifies the type of service addressed, the frequency with which the client will send requests for the service, and the required average response time (SLA). The resource manager tries to find a distribution of the workload among the available servers that would provide the requested QoS. For each client session, a certain number of threads (from 0 to unlimited) is allocated on each Grid server offering the respective service. Incoming service requests are then load-balanced across the servers according to thread availability.

The resource manager considers different configurations in terms of thread allocation and for each of them it generates a predictive performance model (more specifically, a queueing Petri net model [9]) based on the architecture-level performance models of the involved services. The generated model reflects the current system environment in terms of available server resources and active client sessions. The model is analysed through simulation and used to predict the performance of the system in order to ensure that the client SLAs are satisfied. If no configuration can be found that satisfies the client SLAs, the session request is rejected or a counter offer with lower throughput or higher response time is sent back to the client.

We now present some experimental results that demonstrate the effectiveness of the above approach. Three sample services each with different behaviour and resource demands were run as part of the experiments. The services use the Grid to execute some business logic requiring a given amount of CPU time. The business logic includes calls to external (third-party) service providers which are not part of the Grid environment. Figure 5 shows the results from an experiment in which 99 session requests were sent to the resource manager over a period of 2 hours. The average session duration was 18 minutes in which 92

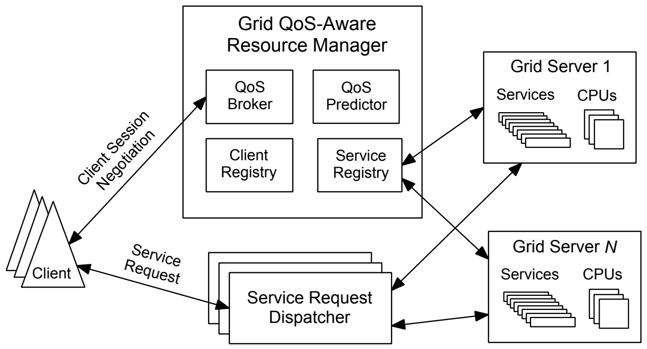

Figure 4. Grid QoS-aware Resource Manager Architecture

service requests were sent on average. We compare the behaviour of the system in two different configurations: i) with basic overload control and ii) with QoS control. In the first configuration, the resource manager simply load-balances the incoming requests over the Grid servers without considering SLAs, however, requests that arrive during periods in which both Grid servers are saturated are automatically rejected. In the second configuration, the resource manager uses its online performance prediction mechanism as described above to ensure that SLAs are satisfied. As we can see, without QoS control, the SLAs of the majority of accepted sessions were not fulfilled, whereas with QoS control, the response times of accepted sessions were much lower and all SLAs were fulfilled. The experiment was repeated for a number of different workload configurations varying the transaction mix, the average session length and the server utilization. The results were of similar quality as the ones presented here and they confirmed the effectiveness of our online performance prediction mechanism.

So far we have assumed that when a Grid server is registered with the resource manager, information on the service resource demands (i.e., CPU service times) is provided as part of the supplied architecture-level performance models. In case the resource demands are not known in advance, a simple method for estimating them on-the-fly based on monitoring data can be used. The method, described in detail in [13], is applicable for services with no internal parallelism. The method is conservative in that it starts with conservative estimates of the resource demands and refines them iteratively as requests are processed. We consider three different configurations in an experiment with 85 sessions over a period of 2 hours: i) Basic overload control, ii) QoS control with resource demands available in advance, iii) QoS control with resource demands estimated on-the-fly.

The experiment was conducted in a virtualised setup with 9 Grid servers. Table 1 presents a break down of the client sessions into: i) sessions for which the client SLA was observed, ii) sessions for which the client SLA was violated and iii) sessions that were rejected by the resource manager. Without QoS control, 96% of the requested sessions were admitted, however, the client SLAs were observed in only 22% of them. In contrast to this, in all configurations with QoS

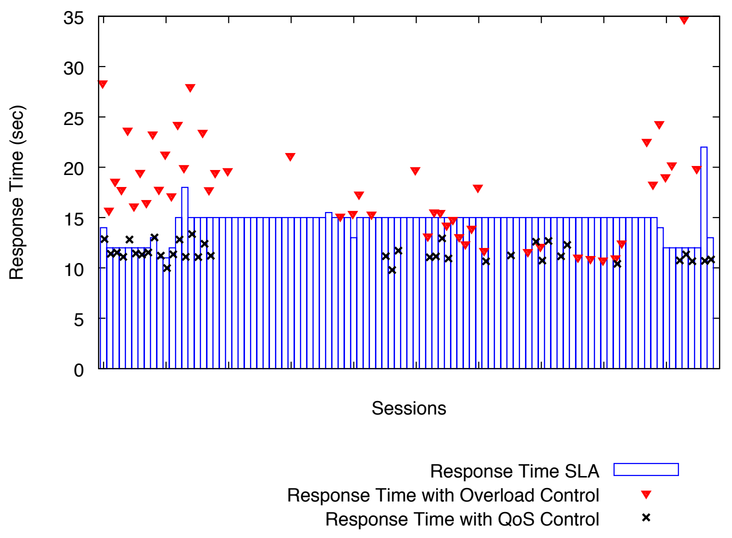

Figure 5. Response time results for 99 sessions over a period of 2 hours.

Table 1. Summary of session SLA compliance.

Configuration	SLA fulfilled	SLA violated	Rejected
1	19	63	3
2	46	2	37
3	34	0	51

control, the SLAs were observed for nearly 100% of the accepted sessions. Indeed, only 2 sessions had their SLAs violated and the violation was by a tiny margin. The price for estimating resource demands on-the-fly was that 14 sessions more were rejected which amounts to 16% of the total number of sessions.

Finally, we extended the resource manager architecture to support adding Grid servers on demand as well as dynamically reconfiguring the system after a server failure. Whenever the QoS requested by a client cannot be provided using the currently available server resources, the extended algorithm considers to launch an additional server to accommodate the new session. At the same time, each time a server failure is detected, the resource manager reconfigures all sessions that had threads allocated on the failed server. Existing sessions might have to be cancelled in case there are not enough resources available to provide adequate QoS. The extended algorithm was subjected to an extensive experimental evaluation the results of which are available in [13]. The results showed that adding servers on demand does not have a significant impact on the performance of the resource manager despite of the decreased flexibility in distributing the workload.

4 Adaptive Time Slot Scheduling

The advantages of predicting the performance of a system online can also be applied to generic distributed algorithms. As a third proof-of-concept we include an adaptive time slot scheduling based on a burstiness metric, that permits to control the monitoring frequency of the system depending on the burstiness levels detected by the algorithm. This means a considerable decrease of the overhead of the monitoring process, whose frequency can be adapted to the stress detected at the entry point of the system.

Considering a locally distributed cluster-based Web information system, a fundamental aim is the monitoring of some Web servers' parameters in an adaptive way in order to reduce the algorithm overhead. Some of the Web servers' parameters likely to be monitored are the arrival rate, the CPU/disk utilization, I/O performance, etc. The performance of the nodes that compound the Web system have to be monitored continuously in order to know their status and make the appropriate decisions in case of overload to avoid a possible congestion situation. This can be done in several ways: *(i)* each time a request arrives at the front-end of the Web system; *(ii)* at fixed times by using static time slot scheduling; or *(iii)* at non-fixed times by using dynamic time slot scheduling. The overhead introduced by option *(i)* is the biggest because each time a request arrives at the Web system, Web node parameters are monitored. While option *(ii)* introduces a constant overhead, option *(iii)* monitors the system at non-fixed intervals, hence, its overhead will depend on the frequency of those intervals. The drawback of defining monitoring in a constant duration interval schedule (option *(ii)*) is the choice of monitoring time interval. It is very difficult to set a duration interval that fits with all possible Internet arrival rates at the Web system due to its heavy tailed pattern.

We have considered six different approaches to define burstiness factors in order to compare their behaviour and detect their benefits or drawbacks under the same circumstances. All the burstiness factor values are defined in [0,1]. The precise definition of the burstiness factors can be found in [6]. Instead of defining them formally, let us describe them visually in Figure 6, where the arrival rate to the system is also shown.

Burstiness Factor 1 (BF1) smooths the arrival rate curve. Fig. 6a illustrates that it follows the arrival rate but does not accurately represent its quick variations. We consider that the burstiness factor should alert the system as quickly as possible of an increase in the arrival rate, and this factor increases or decreases along with the increasing or decreasing arrival rate trend but very slowly and delayed.

We propose the direct inclusion of the arrival rate value in the burstiness factor in the next proposal, as a way to modify it quantitatively. Fig. 6b shows that, in this case, BF2 also varies with the variations of the arrival rate. Nevertheless, there are some peaks in the arrival rate that are not followed by the factor. In the next proposal we introduce a penalisation when detecting a consecutive number of *bursty* slots.

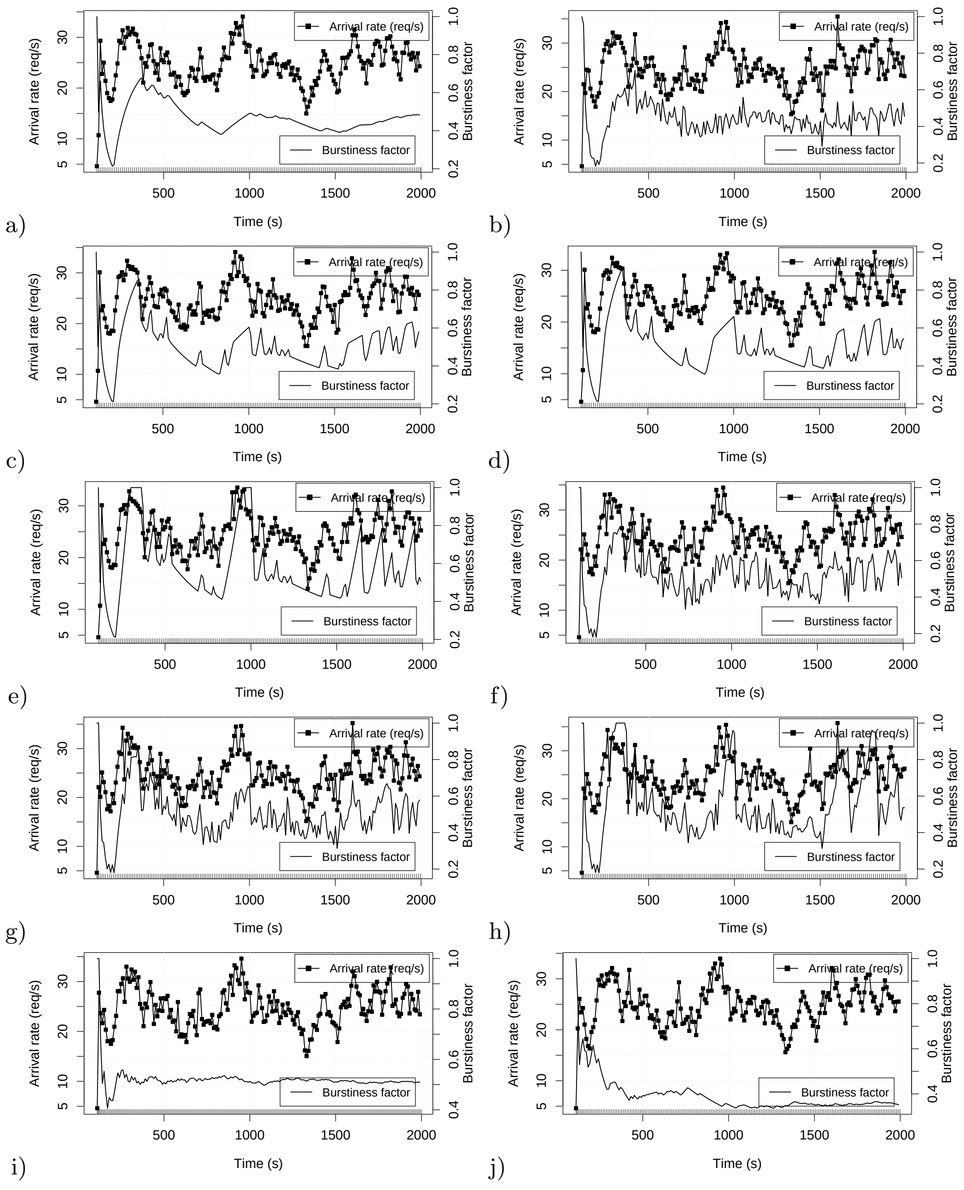
Figure 6. Arrival rate and burstiness factors: a) BF1; b) BF2; c) BF3 with $j = 3$; d) BF3 with $j = 4$; e) BF3 with $j = 10$; f) BF4 with $j = 3$; g) BF4 with $j = 4$; h) BF4 with $j = 10$; i) BF5; j) BF6.

Fig. 6c, 6d and 6e represent the results obtained with BF3 and a record of 3, 4 and 10 slots, respectively. It can be observed that as the number of slots considered increases, the burstiness factor penalisation also increases. We need to check if this penalisation leads to an increase in the system performance or otherwise, decreases its performance because of an overreaction to the arrival rate.

The BF4 values are shown in Fig. 6f, 6g and 6h, representing the results obtained with a maximum record of 3, 4 and 10 slots. We can observe that the resulting curves of BF4 are similar to the BF3 curves, but in this case the burstiness factor is also sensitive to changes in the arrival rate.

Fig. 6i shows the results obtained with this burstiness factor and the resulting curve can be observed as being even smoother than the one obtained from the original BF1.

In Fig. 6j, it can be observed that the BF6 curve does not accurately follow the arrival rate changes. The BF6 curve decreases in some points of Fig. 6j when the arrival rate curve increases. The main drawback of this burstiness factor is the fact that its calculation is made for each incoming HTTP request and then it needs a huge computational effort, which leads to a considerable overhead compared to the other proposals.

In order to define the adaptive time slot scheduling, we divide the total observation time T of the experiment in several slots of variable duration. While the experiment is simulated, the duration of the slot changes based on the value obtained by the burstiness factor. Hence, the duration of the slot $k+1$ is dependent on the burstiness of the two previous slots, $b(k)$ and $b(k-1)$, as follows:

$$\begin{aligned} d(k+1) &= \frac{d(k)}{1+b(k)+b(k-1)},\ if\ b(k) \geq b(k-1) \qquad (1)\\ d(k+1) &= \frac{d(k)}{1+b(k)-b(k-1)},\ if\ b(k) < b(k-1) \end{aligned}$$

Therefore, the number of slots defined during the simulation time is also variable. We can calculate the total number of slots that divide the observation time T during each slot. Considering the duration of the slot $k+1$, the frequency of slots is defined as:

$$e(k+1) = \tfrac{T}{d(k+1)}$$

As the duration of the following slot is defined by the value of the burstiness factor on the current slot, when a burstiness increase is detected, the following testing time is brought nearer in order to check the incoming arrival rate early enough and then tune again the algorithm parameters. If a decrease in burstiness is perceived, the duration of the following slot is enlarged to reduce the overhead. By controlling the burstiness in the arrival rate, and then the duration of testing slots, a sudden reduction in the future performance of the Web servers may be forecasted.

An example of adaptive time slot scheduling is depicted in Fig. 7. In the upper part of the figure the arrival rate and the burstiness factor curve are drawn following adaptive time slot scheduling. As the arrival rate increases from

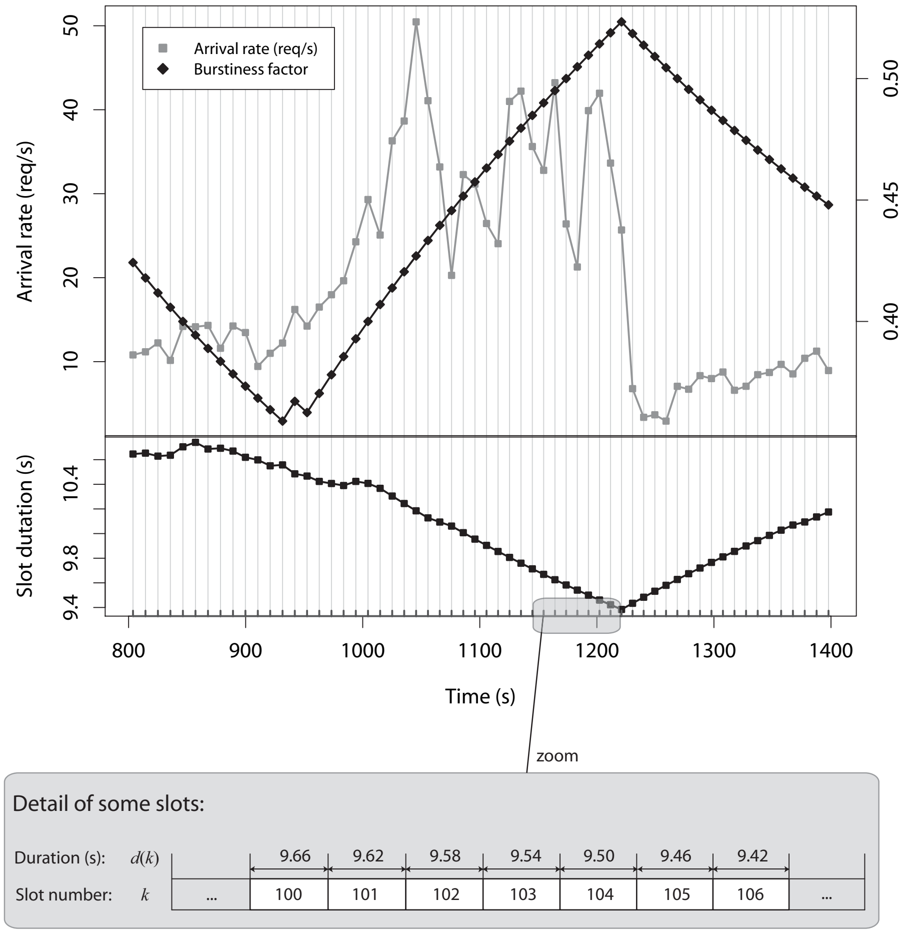

Figure 7. Arrival rate monitored following adaptive time slot scheduling and detail of some of the slots using the BF1.

time instant 910 seconds, the burstiness factor also increases. We have used BF1 to illustrate burstiness factor behaviour in this case. Below this figure, the slot duration is represented in another scale. It can be observed how the duration of the slots decreases when the arrival rate increases. Some slots have been zoomed in to detail the decrease of their durations.

The adaptive time slot scheduling has been implemented in an OPNET Modeler scenario and the complete simulation results can be found in [6].

5 Admission Control and Load Balancing Algorithm

In this section we want to describe an admission control and load balancing algorithm that is based on throughput prediction for a Web system as a fourth case

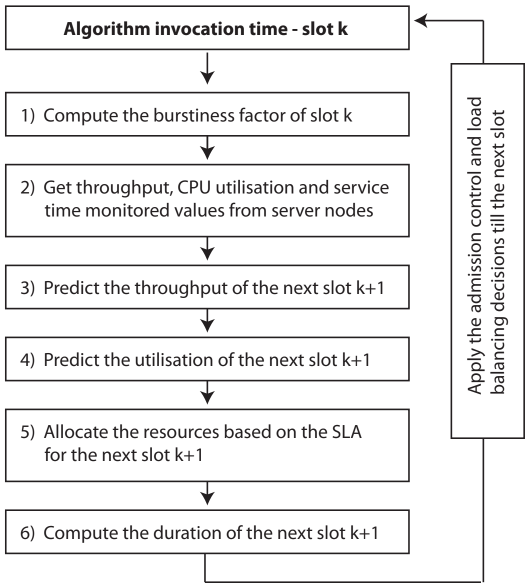
Algorithm invocation time - slot k
1) Compute the burstiness factor of slot k
2) Get throughput, CPU utilisation and service time monitored values from server nodes
3) Predict the throughput of the next slot k+1
4) Predict the utilisation of the next slot k+1
5) Allocate the resources based on the SLA for the next slot k+1
6) Compute the duration of the next slot k+1
Apply the admission control and load balancing decisions till the next slot

Figure 8. Adaptive admission control and load balancing algorithm overview.

study of online prediction. The invocation times of the algorithm are planned based on the adaptive time slot scheduling described in previous section.

This algorithm is adaptive because it is invoked adaptively depending on the arrival rate. Each time it is invoked, some computations need to be done in order to take the admission control and load balancing decisions that will remain till the next invocation. The period of time between invocations is considered a slot.

Figure 8 depicts the general steps that are taken by the algorithm. Once the burstiness factor has been computed and the monitored throughput, CPU utilisation and service time values obtained from the server nodes, the throughput that the nodes will get during the next slot is predicted. Five throughput predictors are defined in order to give us the trend of the system behaviour. These predictors permit the algorithm to take decisions about the distribution of the load in the Web system to maintain the performance of the system independently of the congestion level of the server nodes. Different classes of requests with different priorities are considered in this work. Depending on the priority of each request, we set a fraction of the utilisation of the whole Web system to be used by that request class. The SLA of the requests is defined in terms of CPU utilisation of the Web servers. Therefore, we consider a set of classes, $C = \{c_1, c_2, \ldots, c_r\}$, and define for them a normalised utilisation value in a decreasing order. Hence, the class of requests that represent c_1 have more priority than the class c_2, and so on. Finally, the resource allocation policy establishes how the utilisation of the server nodes is assigned to attend each class of requests that may arrive to the Web system.

The system architecture proposed is based on Web cluster-based network servers and includes a front-end Web switch. A layer-7 Web switch is normally described as a content-aware switch that can de-encapsulate the requests up to the application level and classify them on the basis of this information, but it can easily be the bottleneck of the Web system. This problem is easily solved by transferring the request distribution mechanism to the back-end nodes and replacing the content-aware Web switch with a content-blind Web switch.

The cluster of Web servers is locally connected to the Web switch in a two-tier organisation (Web server and App/DB server), as it is shown in Figure 9. We have considered 5 sets of Web and App/DB servers. Each Web server attends the requests that ask for static files, namely static requests and the App/DB server is accessed when the request asks for a Web page that needs to retrieve dynamic content (dynamic requests).

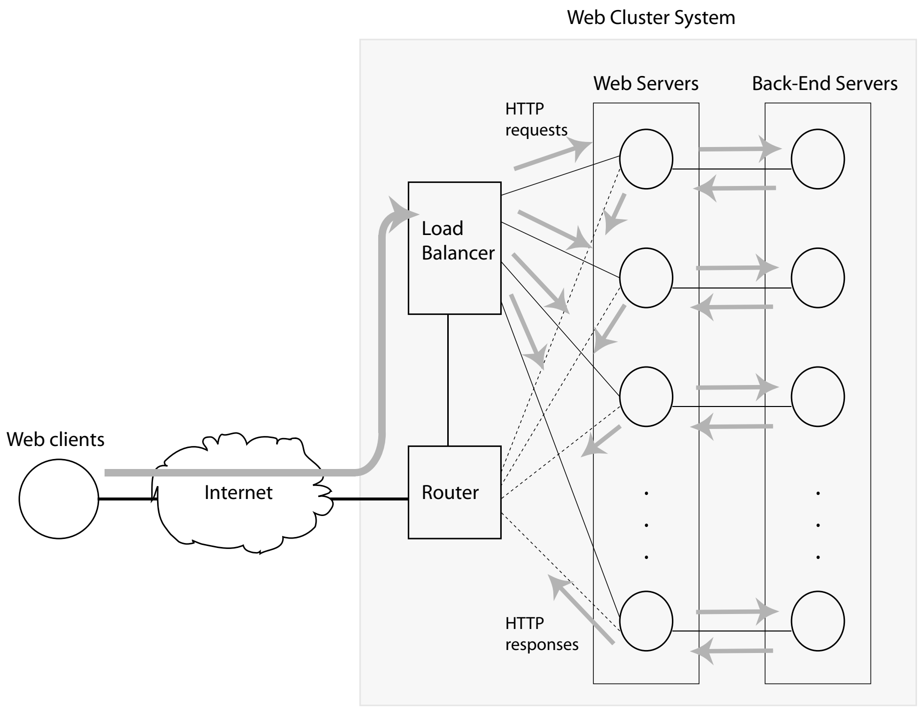

Figure 9. The Web architecture is made up of several mirrored Web servers and their corresponding database servers. The model architecture is one-way, which means that the incoming HTTP requests go through the front-end node but their HTTP responses use a different way to prevent a system bottleneck in this node.

The six throughput predictors (P1-P6) are defined and completely detailed in [7]. We have implemented our algorithm in the simulation tool OPNET Modeler which facilitates accurate simulation of the layers of the TCP/IP stack. We consider two different service classes, named c_1 and c_2, in all the simulations. Each service class contains two types of applications: one that asks for dynamic content and another that asks for static content. Static requests are attended by the Web servers while dynamic requests require access to the App/DB server.

As an admission control algorithm is going to be tested, we need to overload the system. The workload is generated in the Web system by 30, 40, 50, 60, 70, 80, 90 and 100 Web clients, as we are interested in stressing the system to test the algorithm with an increasingly high workload. So, the Web system starts rejecting requests when it is overloaded.

We configure two workloads in order to test the algorithm more accurately. Both are basically the same, the only difference is in the user think time. In Figure 10, we can observe that the arrival rate increases up to 350 Web requests per second for 100 clients during these 30 second periods.

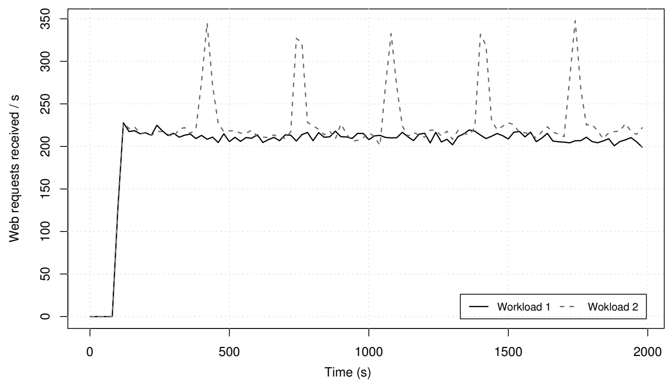

Figure 10. Workload 1 and Workload 2 generated by 100 clients

The response time of dynamic requests, represented in Figure 11, is more meaningful than the one obtained by static requests because the App/DB servers are more congested with the increase of traffic. If we analyse the case of *Workload 1* in Figure 11a, we can note some differences among the response time obtained by the predictors. Focusing on the last case, 100 clients, we can detect that the predictors P1, P2 and P3 obtain a higher response time than predictors P4 and P5. This is also depicted in Figure 11b, which represents the response time for *Workload 2*. We can also observe that the maximum response time for both workloads is around 2.5 seconds, that means that our algorithm achieves an extra goal, that is the limitation of the response time regardless of the amount of traffic arriving to the system. The predictor that shows a good response time and the most stable behaviour is P4, as P5 shows some variability in 70, 80, 90 and 100 clients for *Workload 1*. We can also observe that there is not any differentiation in the response times obtained by class-1 and class-2 traffic, as we do not distinguish different queues in the Web and App/DB servers in order to keep the approach simple.

The response time of dynamic Web pages obtained from the simulations leads us to the conclusion that P4 is the most suitable predictor for our admission control and load balancing algorithm. However, we would like to remark that the predictors P1, P2 and P3 do also obtain good performance results and that have an important advantage: they are easily obtained from the throughput of the two previous slots and that do not need a record of more previous slot throughput values as predictors P4 and P5, which are more complicated to compute (please, see [7] for more information).

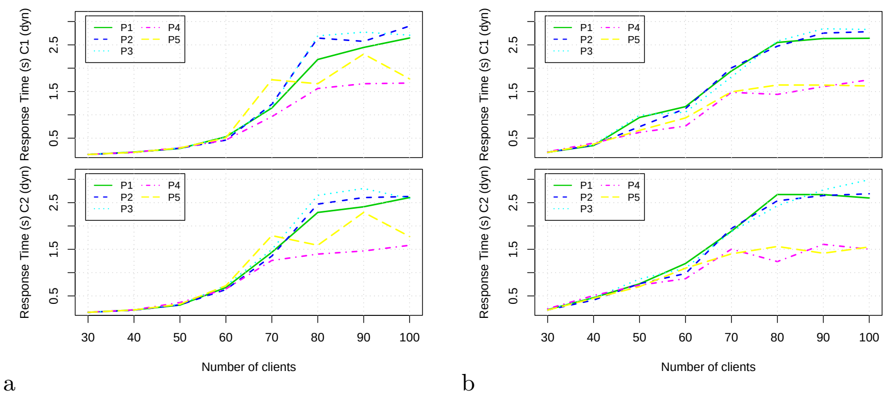

Figure 11. 95th percentile of the response time for dynamic requests: (a) Workload 1; (b) Workload 2

In order to show the benefits of the adaptive time slot scheduling (described in previous Section), the algorithm has been configured to be executed on a fixed time slot scheduling. The predictor chosen for these simulations is P3. The workload chosen for this comparison is *Workload 2*.

The 95th percentile of the App/DB server utilisation is represented in Figure 12. The results obtained when invoking the algorithm periodically are named as “P3_per” in the figure. Here we observe that the utilisation level of the App/DB servers is lower for P3_per in the first points of the x-axis of the graph. In the case of class-1 traffic, the servers seem to be less loaded for 30, 40 and 50 clients with P3_per. The case of 30 clients also reaches a lower utilisation level for class-2 traffic.

However, if we analyse the P3_per utilisation level of class-2 traffic after 40 clients, we can also observe that it is slightly greater that the rest of the simula-

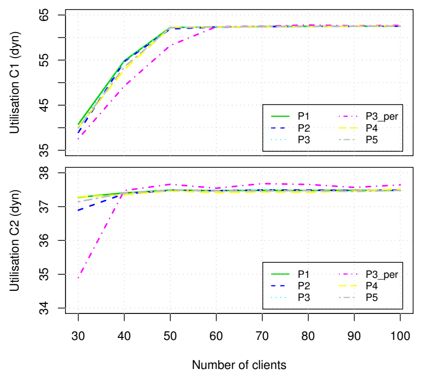

Figure 12. 95th percentile of the App/DB server utilisation for dynamic requests

tions. In fact, this indicates to us that the fixed time slot scheduling introduces some errors in the utilisation level reached for each traffic class. That also means that the algorithm is less accurate in its reservations and that the SLA is less guaranteed.

6 Conclusion

Although the use of online prediction methods and techniques are not yet generalised to all systems, it is clear that resilient systems should consider different strategies to ensure a certain QoS, despite failures, overwhelming services and other inconveniences that usually occur at run-time . In this chapter, we have tried to show through four practical examples how to use simple tools to bring interesting benefits, thanks to online predictions. However, it is much research in this direction, especially in finding common methodologies for building resilient systems considering the online prediction of the future and react accordingly by adjusting the prediction over time. These methodologies should consider not only the techniques presented here but other appropriate to each level of design abstraction and at each layer of the system during operation. We hope these four case studies illuminate the reader about these possibilities.

7 Acknowledgements

The work of Samuel Kounev was partially funded by the German Research Foundation (DFG) under grant No. KO 3445/6-1.
The work of Carlos Juiz was partially funded by the Spanish Ministry of Science and Technology under grant TIN2007-60440.

References

1. S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Syst. and Softw.*, 82:3–22, 2009.
2. G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. Wiley-Interscience, New York, NY, USA, 1998.
3. G. E. P. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control.* Holden-Day, Incorporated, 1990.
4. F. Brosig, S. Kounev, and K. Krogmann. Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In *Proc. of ROSSA-2009.* ACM, 2009.
5. Y. Chen, A. Das, W. Qin, A. Sivasubramaniam, Q. Wang, and N. Gautam. Managing server energy and operational costs in hosting centers. In *SIGMETRICS Int. Conf.*, 2005.
6. K. Gilly, S. Alcaraz, C. Juiz, and R. Puigjaner. Analysis of burstiness monitoring and detection in an adaptive web system. *Computer Networks*, 53:668–679, 2009.
7. K. Gilly, C. Juiz, N. Thomas, and R. Puigjaner. Adaptive admission control algorithm in a qos-aware web system. *Information Sciences*, In Press, 2011.
8. G. Jung, K. Joshi, M. Hiltunen, R. Schlichting, and C. Pu. A cost-sensitive adaptation engine for server consolidation of multitier applications. In *Int. Conf. Middleware*, 2009.
9. S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, July 2006.
10. J. Li, X. Ma, K. Singh, M. Schulz, B. de Supinski, and S. McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 89 –100, april 2009.
11. D. A. Menasce and M. N. Bennani. On the use of performance models to design self-managing computer systems. In *Proc. 2003 Computer Measurement Group Conf*, pages 7–12, 2003.
12. D. A. Menasce, M. N. Bennani, and H. Ruan. On the use of online analytic performance models in self-managing and self-organizing computer systems. In *In Self-* Properties in Complex Information Systems*, pages 128–142. Springer Verlag, 2005.
13. R. Nou, S. Kounev, F. Julia, and J. Torres. Autonomic QoS control in enterprise Grid environments using online simulation. *Journal of Systems and Software*, 82:486–502, 2009.
14. R. Nou, S. Kounev, and J. Torres. Building online performance models of grid middleware with fine-grained load-balancing: a globus toolkit case study. In *Proceedings of the 4th European performance engineering conference on Formal methods and stochastic models for performance evaluation*, EPEW'07, pages 125–140, Berlin, Heidelberg, 2007. Springer-Verlag.