UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Proactive Top-Down Approach to Dynamic Allocation of Resources in Data Centers

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Filippo Seracini

Committee in charge:

Professor William Griswold, Co-Chair
Professor Ingolf Krüger, Co-Chair
Professor Ryan Kastner
Professor Tajana Simunic Rosing
Professor Hal Sorenson

2014

The Dissertation of Filippo Seracini is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2014

DEDICATION

To my father
...for always believing in me,
for his tireless encouragement,
and for his support in every single step of my life.

To my mother
...for always having a smile for me,
and for the endless hours spent talking on Skype
making sure I would not feel alone or homesick.

To my uncle Massimo
...who showed me that no matter what life throws at you,
you should always smile and enjoy every second of it.

To my fiancè Valentina
...for her love and patience in these years,
and for pushing me to finish this Thesis.

EPIGRAPH

"e fare come gli arcieri prudenti,
a' quali parendo el loco dove disegnano ferire troppo lontano,
e conoscendo fino a quanto va la virtù del loro arco,
pongono la mira assai più alta che il loco destinato,
non per aggiugnere con la loro freccia a tanta altezza,
ma per potere, con lo aiuto di sì alta mira,
pervenire al disegno loro"

Niccolò Machiavelli
Il principe, Chpt. VI

"When the going gets tough, the tough get going"

Joseph P. Kennedy, Sr.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Many people have partaken in my journey to achieve this Ph.D. degree. Without them, this journey would have been much harder, if not impossible.

First and foremost, I would like to thank my father, for always being by my side and for his endless support throughout these seven years of graduate studies. My father has never stopped believing in me, even during the toughest moments. He endlessly pushed me to do my best and look ahead, confident that, eventually, the results would manifest themselves. He was right. Through his example, he taught me to never give up, no matter how hard things may appear. He has being supporting me and doing everything he could possibly do to allow me to have an amazing life. Words cannot say how grateful I am to what he has done and keeps doing for me. I would like to thank my mother, for being the best mother I could possibly hope for. Since I started my graduate studies, she has video called me almost every day, always showing up on the computer screen with her big smile that would make me feel like I was at home. She has encouraged me and cheered me up throughout the entire journey, and she has always been there for me every time I needed it. This Ph.D. is dedicated to my parents.

I would like to thank my uncle Massimo and my cousin Olivia, for being there for me, especially when I first arrived to San Diego. They have been my family in this part of the world and have always made sure that I felt at home. The example of how my uncle, who faced his cancer till the very last second, is something that forever changed the way I look at life. No matter how bad things got, he never stopped smiling. I have been trying to follow his example throughout these years.

I would like to thank my advisor, Professor Ingolf Krüger, for taking me into his team, for his advice and financial support that made this Ph.D. possible. I would like to thank Professor William Griswold, for his encouragement, moral support, advice, and the overall interest in my project. I thank Professor Tajana Simunic Rosing, for her invaluable

advice and guidance in the area of energy efficiency of data centers. I would like to thank Professor Luciano Baresi, Professor Sam Guinea and Professor Giuseppe Serazzi for hosting me in their teams as visiting Ph.D. student at the Politecnico di Milano, and for making me feel as one of their peers. The long discussions, their extensive support, and their advice on autonomic computing and queuing networks was instrumental for the research presented in this thesis. I would like to thank Massimiliano Menarini for being both my "personal advisor" and a great friend that guided me throughout my graduate studies. A thank you goes also to Xiang Zhang and Giovanni Quattrocchi for being both great friends and colleagues that spent many days, nights and weekends working with me on the projects presented in this dissertation. I thank William Bagnai, for his meticulous review and feedback on this work.

Finally, I would like to thank my fiancè Valentina, for her love and for being my North star, keeping me focused and encouraging me to complete this thesis.

Chapter 3, in part, is a reprint of the material as it may appear in Seracini, Filippo; Menarini, Massimiliano; Krüger, Ingolf; Baresi, Luciano; Guinea, Sam; Quattrocchi, Giovanni, "A Comprehensive Resource Management Solution for Web-based Systems", in Proceedings of the 11th International Conference on Autonomic Computing (ICAC) 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in full, is a reprint of the material as it appears in Menarini, Massimiliano; Seracini, Filippo; Zhang, Xiang; Rosing, Tajana; Krüger, Ingolf, "Green Web Services Improving Energy Efficiency in Data Centers via Workload Predictions", in Proceedings of the 2nd International Workshop on Green and Sustainable Software (GREENS), IEEE, May 2013,. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in full, is a reprint of the material as it appears in Seracini, Filippo; Zhang, Xiang; Rosing, Tajana; Krüger, Ingolf, "A Proactive Customer-Aware Resource

Allocation Approach for Data Centers" at the 12th IEEE International Symposium on Parallel and Distributed Processing with Applications. The dissertation author was the primary investigator and author of this material.

VITA

2007	Bachelor of Science, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Firenze, Florence, Italy
2007–2010	Research Assistant, Department of Computer Science and Engineering, University of California, San Diego, USA
2010	Master of Science, Computer Science and Engineering, University of California, San Diego, USA
2010	Certificate in Architecture-Based Enterprise System Engineering, Jacob School of Engineering and Rady School of Management, University of California, San Diego, USA
2010–2013	Research Assistant, Department of Computer Science and Engineering, University of California, San Diego, USA
2014 -	Program Manager, Microsoft, Redmond, USA
2014	Doctor of Philosophy, University of California, San Diego, USA

PUBLICATIONS

Filippo Seracini, Xiang Zhang, Tajana Simunic Rosing, and Ingolf Krüger,"A Proactive Customer-Aware Resource Allocation Approach for Data Centers", in Proceedings of the 12th International Symposium on Parallel and Distributed Processing with Applications (ISPA), August 2014.

Filippo Seracini, Massimiliano Menarini, Ingolf Krüger, Luciano Baresi; Sam Guinea; Giovanni Quattrocchi, "A Comprehensive Resource Management Solution forWeb-based Systems", in Proceedings of the 11th International Conference on Autonomic Computing (ICAC), June 2014.

Davide Cerotti, Marco Gribaudo, Ingolf Krüger, Pietro Piazzolla, Filippo Seracini and Giuseppe Serazzi, "Throughput maximization with multiclass workloads and resource constraints", in 21st International Conference on Analytical & Stochastic Modelling Techniques & Applications, June 2014, Springer Verlag LNCS.

Filippo Seracini, Xiang Zhang, Massimiliano Menarini, Tajana Simunic Rosing, and Ingolf Krüger,“Green Web Services Improving Energy Efficiency in Data Centers via Workload Predictions“, in Proceedings of the 2nd International Workshop on Green and Sustainable Software (GREENS), May 2013, IEEE pp. 8-15.

Ingolf Krüger, Massimiliano Menarini, Filippo Seracini, Massimilian Fuchs, and Jens Kohl,“Improving the Development Process for Automotive Diagnostics“, in International Conference on Software and System Process (ICSSP), June 2012, pp. 63-67, IEEE.

Filippo Seracini, Ingolf Krüger and Tajana Simunic Rosing, “A Top-Down Approach to Energy saving in Data Centers. Energy Aware Service Oriented Architectures“, poster at the 11th International Workshop on Environment and Alternative Energy organized by NASA, ESA (the European Space Agency) and C3P (the Portuguese Centro Para Prevenção da Poluição), November 2011.

ABSTRACT OF THE DISSERTATION

A Proactive Top-Down Approach to Dynamic Allocation of Resources in Data Centers

by

Filippo Seracini

Doctor of Philosophy in Computer Science

University of California, San Diego, 2014

Professor William Griswold, Co-Chair
Professor Ingolf Krüger, Co-Chair

Over the last couple of decades, data centers have seen a substantial increase in number, size, and use. This massive growth started with the emergence of the World Wide Web and accelerated with the advent of social media (e.g. Facebook, Twitter), content-sharing platforms (e.g. Instagram, Flickr), and cloud technologies (e.g., Amazon EC2, Microsoft Azure and OneDrive). Nowadays, large enterprises, service providers, and public cloud providers utilize anywhere from few thousands to hundred of thousands of servers in their data centers, which consume incredible amounts of electricity. However, studies have shown that the average utilization of these data centers is extremely low,

resulting in a waste of computing resources and energy. This is due to the fact that resources are typically allocated in such a way that, if a spike of requests occurs in a data center, the Internet applications that run inside it do not violate service level agreements (SLAs). As a result, resources are over-allocated, and the overall utilization of the data center remains very low.

This dissertation improves upon the state of the art of resource allocation techniques by taking an integrated, holistic, top-down approach, that proactively adapts the amount of resources allocated, based on incoming future workloads. This approach can be applied to workloads where different requests are correlated, such as service-oriented Internet applications. To enable proactive adaptation, this research introduces two contributions: 1) an approach to predict future workloads by leveraging correlations between requests sent to an application and 2) a technique to estimate, with acute accuracy, the impact that those workloads have on the performance of the currently allocated infrastructure. Thanks to these contributions, the work presented here can predict an incoming workload, calculate the required amount of computing resources needed to run it under a given SLA, and proactively adapt the infrastructure whenever needed.

The contributions presented in this dissertation are validated using a well-known benchmark and an implementation of a web service based on a public API; results are compared against existing solutions from scientific literature. Results show double-digit improvements for both energy savings and reduction of the amount of resources allocated, with no additional SLA violations.

Chapter 1

Introduction

1.1 The Problem with the Allocation of Resources in Data Centers

Over the last couple of decades, data centers have seen a substantial increase in number, size, and use. This massive growth started with the emergence of the World Wide Web and accelerated with the advent of social media (e.g. Facebook, Twitter), content-sharing platforms (e.g. Instagram, Flickr), and cloud technologies (e.g., Dropbox, Amazon EC2, Microsoft Azure and OneDrive). Nowadays, large enterprises, service providers, and cloud providers utilize anywhere from thousands to hundred of thousands of servers [75, 49, 84, 86, 56] in their data centers, which consume incredible amounts of electricity. As a result, operational costs and carbon emissions related to data centers have escalated considerably. Currently, the cost of power for a data center can easily account for more than 50% of a company's energy bill [53]. In addition to economic concerns, there is also a growing social and political interest to reduce the environmental impact of these data centers [19]. In 2007, the carbon footprint of the ICT industry accounted for more than 2% of the worldwide carbon footprint [48]. This number is increasing, since more and larger data centers are being built every year around the world [83, 85, 87]. Moreover, many studies [35, 99, 51, 36, 71, 39, 49] have revealed that the average server

utilization in these data centers is surprisingly low: the study in [99] estimates it to be as low as 18%. A low utilization directly translates to a waste of resources, namely energy, as well as increased management costs. Thus, considering the growth of data center based technologies, the rising costs of electricity, and stricter emission regulations, the optimization of computing resources has become a real and urgent necessity.

Until a few years ago, both the cost for electricity and the demand for computing resources were low. This paradigm has drastically changed. Nowadays, resource utilization and energy efficiency in data centers are indeed major concerns. The good news is that much can be done to improve the current situation. A conservative estimate demonstrates that an energy efficient data center could consume 25% less electricity, which translates to $4.5 million a year of savings for a mid-sized facility [52]. The Environmental Protection Agency (EPA) recently estimated that annual savings could range from US$25 to $75 per server [109]. If we multiply these values to account for tens of thousands of servers, as would exist in a medium size data center, or hundreds of thousands for large data centers, reduction in power usage and greenhouse gas emissions are substantial.

Virtualization solutions (e.g. Xen, Microsoft Hyper-V), smart planning, and allocation strategies can improve the utilization of resources in data centers, further contributing to superior energy efficiency. In fact, many studies have proven that it is more convenient to fully utilize a single server than to run two servers at half capacity, because servers' energy consumption is not linear with their utilization. For example, the servers that were used during this research work measure around 230W when idle, and about 350W when fully running; each server had 2 Xeon quad core 2.4 GHz CPUs, 16 GB memory, and a 500 GB disk. It is important to keep in mind that a reduction in energy used by servers has multiple effects on savings; not only do operators pay less to run them, but also less A/C is required, since a smaller amount of exhaust heat is produced.

According to Sharma et al. [95], the energy problem in data centers can be addressed at three complementary levels: cooling, power, and compute. The first level optimizes the efficiency of the cooling system within the data center itself. The second level focuses on improving the power infrastructure and power distribution in the data center. Finally, the compute level organizes the workload of a data center to be as energy efficient as possible. The focus of this Thesis is in optimizing the compute dimension by dynamically allocating resources in a data center.

1.2 Structure of a Typical Data Center

In this section, we provide an overview of what a common data center looks like, from the external structure to the management infrastructure.

As mentioned in [75, 49, 84, 86, 56], modern data centers can host an incredible amount of servers. These servers are hosted in large warehouses, such as the the one shown in Figure 1.1. Internally, data centers are organized into aisles of racks, such as the ones shown in Figure 1.2, where each rack can contain from few dozens to over a thousand of servers. As of 2014, densities of up to 1440 servers per rack are achievable with blade systems [17]. Each rack has one, or more for redundancy, power distribution units (PDUs) that provide electricity to the servers. On top of each rack, there are typically two switches to provide both intra- and inter-rack connectivity.

The internal hardware configuration of the servers varies significantly, depending on the kind of workloads executed. The software stack of these servers can also vary significantly. Here, we discuss only the operating system and management layer. Until a few years ago, it was common practice in the data center administrator community to install a single operating system on each server. That would pose a limitation on the number of tenants that could concurrently utilize the same physical server, because strong isolation between them was very difficult, if not impossible, to enforce. That was one of

Figure 1.1. Microsoft's data center in San Antonio, Texas. (470,000 square feet) [1]

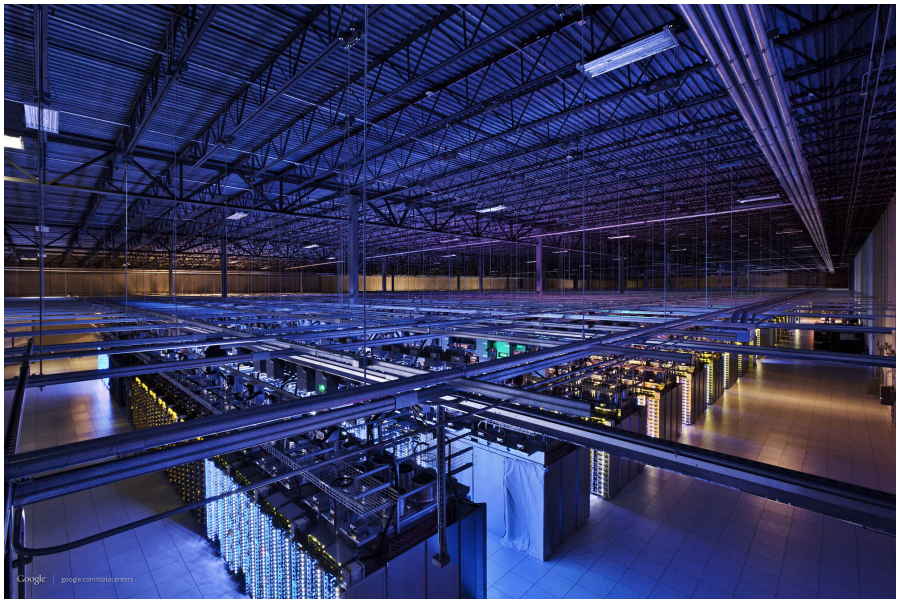

Figure 1.2. Aisles of racks, full of servers, inside one of Google's server farms in Council Bluffs, Iowa. (115,000 square feet) [8]

the reasons for the very low utilization in data centers, as, typically, each tenant would be assigned to each own set of servers, wasting the unused capacity.

In the last decade, several solutions to improve tenant isolation have been proposed, e.g. [33, 12, 16]. These solutions typically consist of a software layer called hypervisor, on top of which multiple operating systems can be installed and executed concurrently, in complete isolation. These operating systems running on top of the hypervisor are referred as virtual machines (VMs). By abstracting away the hardware layer, an hypervisor intercepts the software interrupts from the VMs above, and reroutes those interrupts in such a way that no VM can access any data of other VMs. This approach is called virtualization, and it is being widely adopted in the community. Multiple VMs are then allocated on a single physical host, until its capacity is fully utilized. In the next sections, we will discuss the implications of virtualization on capacity planning.

Virtualization is one of the main enablers of a completely disruptive business model: cloud computing. Indeed, thanks to virtualization, the number of tenants that can be concurrently hosted in a data center is not bounded by the number of physical servers installed. This allows one to significantly increase the ROI of data center hosters, as a much higher number of tenants can be served, thanks to a much higher resource utilization. With cloud computing, tenants, such as enterprises, can dynamically rent computing resources, on demand, based on their business needs, instead of having to build and manage their own data centers. By doing so, enterprises can significantly reduce their capital expenses, as well as the operational expenses associated with the personnel required to managed and maintain their data centers. Enterprises that cannot leverage public cloud offerings can still benefit from cloud technologies to improve their data centers utilization. Hence, cloud computing is a very appealing business model both for hosters, that offer cloud solutions, and enterprises, that either lease those cloud solutions or use them to manage their own data centers.

In the last decades, a number of different cloud technologies have been proposed. These technologies typically build on top of each other, composing the *cloud stack* [73], shown in Figure 1.3. The most commonly adopted and widely known cloud technologies are *Infrastructure as a Service (IaaS)* and *Software as a Service (SaaS)*. With IaaS, enterprise can dynamically lease VMs on demand to accommodate their business needs. The largest public IaaS providers are Amazon Elastic Cloud Computing (EC2), Microsoft Azure and Google Compute Engine. A IaaS solution is also commonly adopted by large enterprises to offer computing resources to their internal departments.

While IaaS solutions are mostly oriented towards enterprises and a very technical audience, SaaS solutions address also consumers' needs. Products like Microsoft OneDrive, Google Drive, Dropbox, Gmail, Office365, just to name a few, are all SaaS solutions, and many of them are offered both in a free and a premium version. With SaaS, the user is completely abstracted away from the underlying hardware, the locality of the data center and the underlying software stack. The user's focus is only on the functionality offered by the SaaS application.

1.3 SLA-Driven Resource Management

Service-oriented architectures (SOAs) are one of the most commonly used architectural approaches in Internet and cloud applications. Service companies such as Amazon (e.g. EC2, fulfillment, payment, etc.), social networks such as Facebook, or cloud providers such as Microsoft Azure, expose all their functionalities as web service based APIs. Complex distributed applications are then constantly built on top of these services to provide more complex services to end users. All these services are hosted on millions of servers in data centers all around the world.

A Service Level Agreement (SLA) is a contract between a service provider and a service consumer that regulates the quality of service (QoS), such as response time,

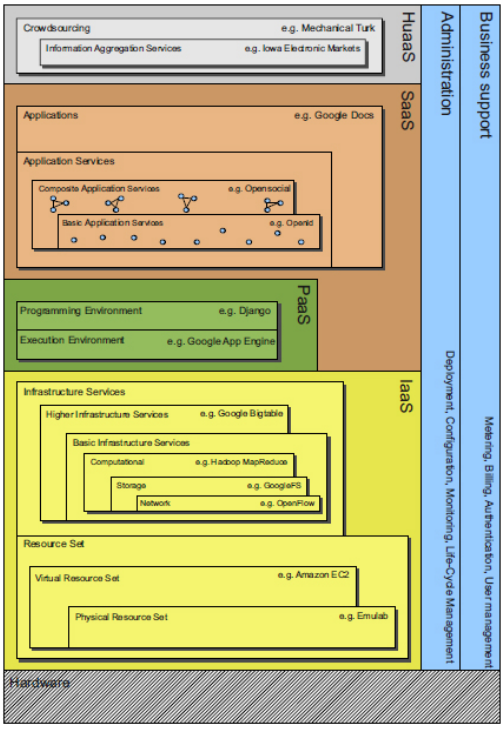

Figure 1.3. The cloud stack, with the different cloud solutions. [73]

and the penalties in case of violations. At any point in time, a multitude of clients is using these Internet applications, sending requests to multiple services. Individual clients expect these services to be always available, reliable and with a bounded, acceptable response time. This means that clients should receive the data and or the functionality they requested within the time frame defined in the SLA, independently of any external factor such as system load fluctuations or amount of resources allocated.

Each server in a data center incurs costs, such as the price of acquisition, electricity, cooling, maintenance, and management. Internet applications and services owners (service providers) have strong financial incentive to maximize their profits by increasing the amount of work, hence revenue, that they can get out of the resources allocated. However, such demands are often quite expensive. If not properly handled, a spike in demand can quickly saturate all allocated computing resources of a provider. This increases costs for the service provider, due to the penalties associated with SLA violations; in

addition, the consumer is inconvenienced if the service becomes no longer available or its performance is degraded. For instance, if the computing infrastructure of an e-commerce web site saturates, business could be lost because customers cannot access the web site and make purchases.

To make things more complicated, the volume of requests received by an Internet application or web service can significantly vary over time. Some of these variations follow recurring patterns. For instance, an e-commerce website will likely receive a high volume of requests during the months preceding the Christmas holidays. Similarly, a sports portal will receive a lot of visits during large sport events. These seasonal and historical patterns are fairly predictable; therefore, they can be accounted for when allocating resources for the Internet application.

However, traffic might occasionally increase over the average expected value, leaving the infrastructure underallocated. Such an unexpected variation of the volume of requests is indeed very hard to predict and plan for. For example, a news web site will likely receive a surge of requests immediately following a breaking story. If the resources allocated are not enough to satisfy the incoming requests, the news web site could quickly saturate and collapse, thereby losing readers that are unable to connect to it. The web site could also pay penalties because it did not show ads during a particular time of the day, thus violating its SLA with the advertisement company.

To avoid penalty charges and loss of business, providers commonly attempt to account for load spikes by overallocating resources. Providers use the maximum expected value as a reference for resource planning, even though the average request load is often significantly lower. As a result, data centers often have large amounts of servers sitting idle, waiting for a spike to eventually hit the data center, a spike that may never even occur. Figure 1.4 summarizes the current state of the art based on historical usage. As mentioned previously, studies show that the average resource utilization in data centers

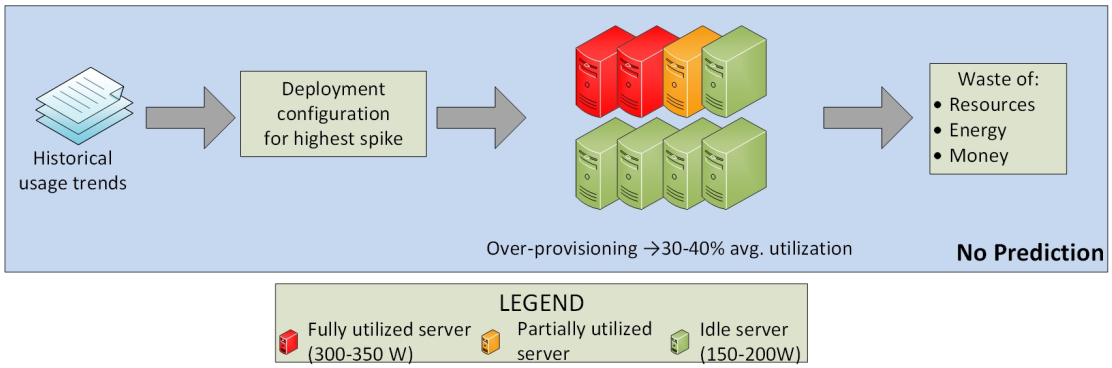

Figure 1.4. Common resource allocation strategy in data centers.

using this allocation strategy can be as low as 18%, which invariably wastes resources, energy, and money for the service provider. This over-provisioning approach is becoming unsustainable, especially when one considers the rising cost of electricity; electricity already accounts for almost 50% of the operational costs of a data center [19].

1.3.1 The Resource Domain Model for SLA

The models in Figure 1.5 and Figure 1.6, respectively, describe the relationship between a logical architecture of a SOA and its deployment, and how this relationship relates to the SLA. Figure 1.5 describes the services and their relationships within a SOA. The main entity of the logical architecture is the notion of *Service Specification*, which could be a simple functionality block such as a Web service (*Basic Service*), or it could be hierarchically decomposed into multiple services (*Composed Service*), that is, a service whose functionality is the result of aggregating and leveraging the functionality of other services.

Following the Rich Service architectural pattern [27], a service specification is further classified into *Infrastructure Service* and *Application Service*. An Infrastructure Service provides capability to address cross cutting concerns, such as monitoring, logging, or authenticating. An Application Service, on the other hand, implements the business

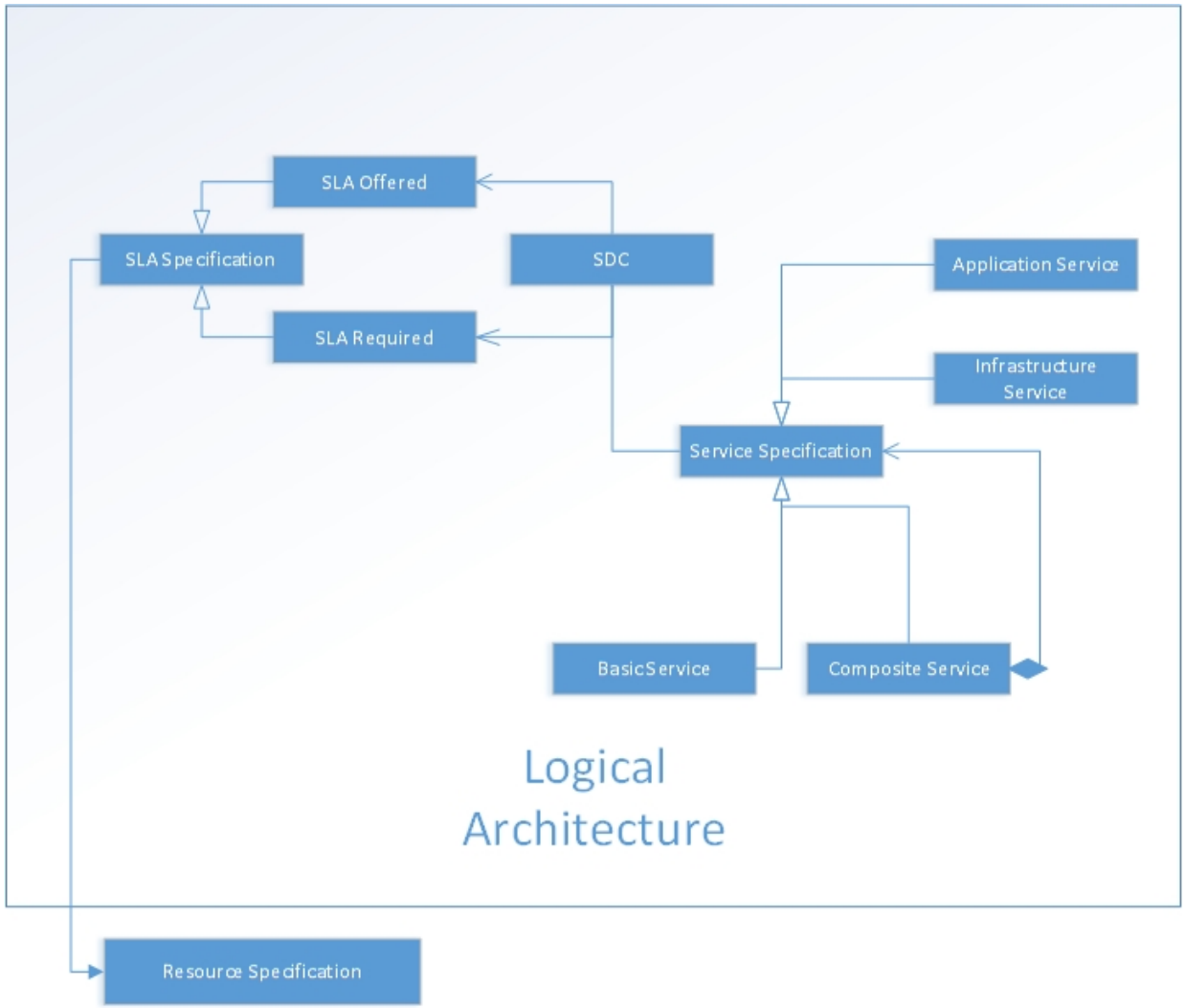

Figure 1.5. SLA-Resource domain model - The logical architecture

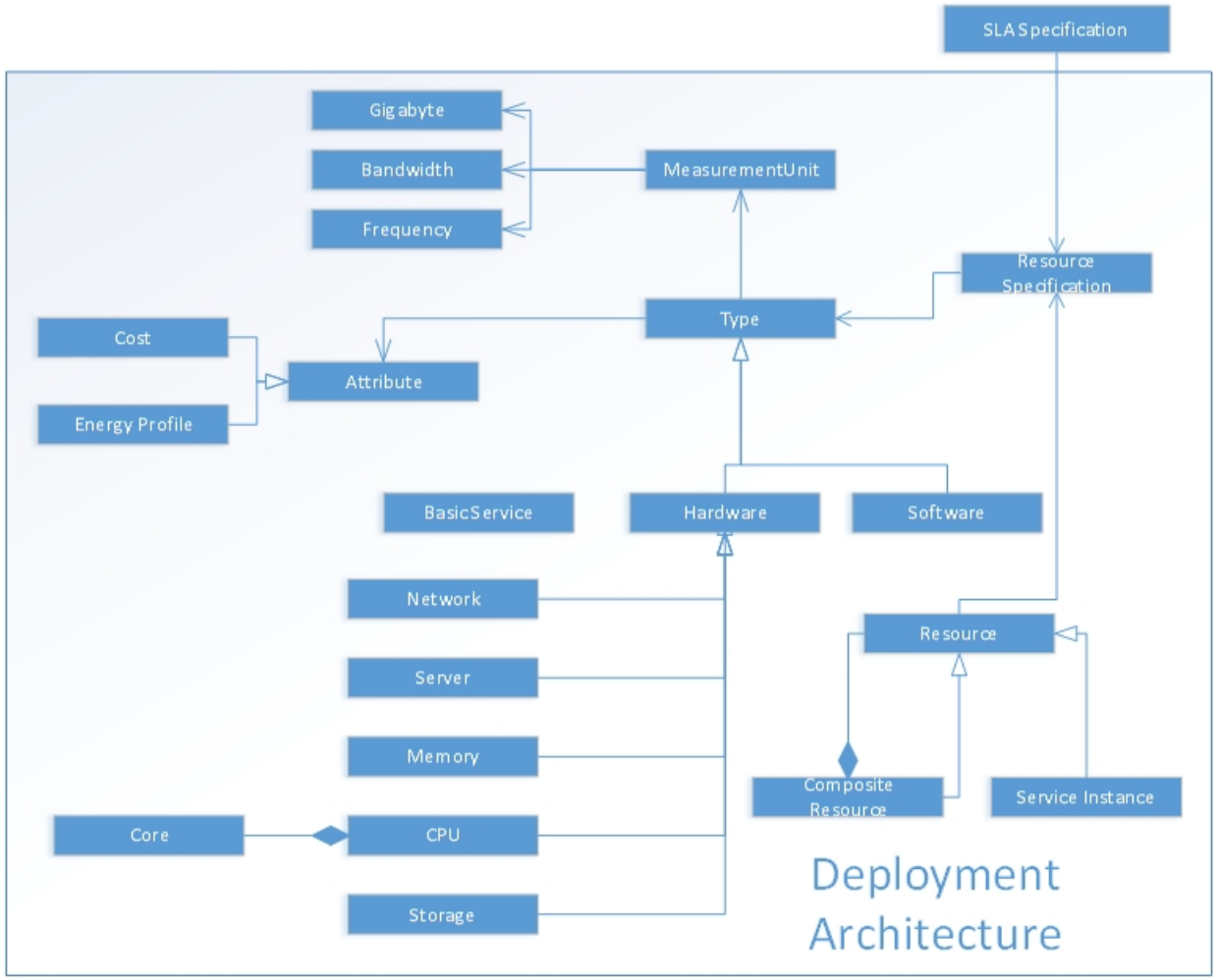

Figure 1.6. SLA-Resource domain model - The deployment architecture

logic of the application. Each *Service Specification* exposes its functionality through the *Service Data Connector (SDC)*, which hides the internal structure of the service and exports only the external interface. So, if service A wants to leverage service B's functionality, it will achieve that by using the interface exposes by service B's SDC.

Since the *SDC* is the services interface exposed to the external world, it is bound to the SLA that the service must hold to, the *SLA Offered*. Similarly, if the service needs to use other services, its *SDC* will take a dependency on their SLAs, the *SLA Required*. Both the *Offered* and the *Required* are types of *SLA Definitions*. So, when creating a service or a SOA, the developer should look for services that satisfy the *Resource Specifications* defined in its *SLA Required* specification. A *Resource Specification*, see Figure 1.6, has a certain quantity of resources of type *Type*. A *Type* can either be *Hardware* or *Software*. A *Type* has also a *Measurement Unit* and multiple *Attributes* like *Cost, Energy Profile*, etc.

1.3.2 Roles and implications for service clients and providers

A very commonly used *Resource Specification* in SLAs is Response Time. This determines the time–it can be defined as maximum, average, X percentile, etc.–that spans from when a service is invoked to when the response is sent out. Services can be offered that leverage other services; this is also reflected in the SLA specifications between services. Figure 1.5 correctly captures this relationship. Hence, if service A leverages service B, as is commonly done in Software as a Service (SaaS) cloud offerings, A will have an *SLA Specification* defined with B. In the model in Figure 1.5 and 1.6, response time would be a *Resource Specification*, with *Time* as *Type* and *Seconds* as *Measurement Unit*. In this case, in order for B to satisfy A, the *SLA Offered* by B will have to have a smaller value then that in the *SLA Required* by A.

This relationship between SLAs and services is recursive. Continuing with the example, B could be a SaaS offering that runs on top of an Infrastructure as a Service

(IaaS); let's call this service C. So, B has an SLA agreement with the service provider of C (the data center hoster) that guarantees, for instance, that B will always run on a set of servers with a quad core processor and will have a guaranteed 100 Mbit/s data transfer rate. These specs will guarantee B to be able to meet its *SLA Offered* to A. The higher the specs that B will require C to provide, the more the B service provider will have to pay. Similarly, the hoster will have to avoid over allocating the servers where service B runs by adding, for instance, too many additional network intensive applications. If the provider over allocated the servers, the available data transfer rate would drop below the value defined in the SLA agreement and the provider would incur penalties.

The service-oriented world is dominated by the SLAs that regulate them. This leaves every service provider to constantly try to solve the resource optimization problem. On one hand, providers want to maximize revenue, which is directly tied to the number of clients satisfied. On the other hand, they also strive to minimize costs, incurred by intermediate services used to deliver the final service. In the case of a hoster, the service offered would be the resources of its data center. Allocating too many clients for its infrastructure would drive up revenues, but it would break the SLAs, resulting in penalties. Moreover, if the SLAs are violated too often, clients would start leaving the hoster. On the contrary, allocating too many servers would keep the clients happy, but it would increase the provider's costs.

Variable and often unpredictable workloads, mixed with the fear of penalties and losing clients as a result of poor quality of service have forced service providers to over-plan and over-allocate resources for their services. This is particularly true with hardware service providers (hosters) or enterprise data centers, where the over-allocation is so exaggerated that the average utilization is as low as 18% [99].

1.3.3 Key Insight: The Application Layer

As discussed in the previous sections, service-oriented architectures are based on services. Each service offers a piece of functionality and may leverage other services. Relationships between services are regulated by SLAs that specify the QoS that each service is required to provide, such as response time, bandwidth, number of cores, etc. If a service violates the SLA, penalties and fines are issued to the service provider.

The main goal of this work is to improve the utilization of resources and the energy efficiency of data centers. So, we started looking at possible targets for optimizations. We immediately noticed that a lot of work had been done both at the hardware level (more power efficient processors, CPU power states, etc.) and at the operating system level, such as virtualization and virtual machine consolidation. But we also noticed a common aspect among all these techniques: they are all reactive, that is, they change/update the system only after an event has occurred. For a complete analysis of existing solutions, we refer the reader to the next chapters, where extensive related work discussions are applied to evaluate our contributions. For instance, the level of virtual machine consolidation would increase (reducing the required number of active servers) only after the servers had been sitting idle for some time. This translates to a waste of energy due to the slow adaptation. The opposite scenario is also very relevant. If the allocated resources are hit by a sudden increase of requests, the system could find itself under-provisioned and possibly incur SLA violations. Unfortunately, current solutions can only partially mitigate the resource allocation problem as they can only react after an event has occurred.

To solve the resource allocation problem, service providers need to dynamically adapt the amount of resources allocated **before events occur**. To achieve that, service providers must be able to **predict the incoming workload** and assess the impact that such a workload will have on the system currently allocated. Thus, the allocation

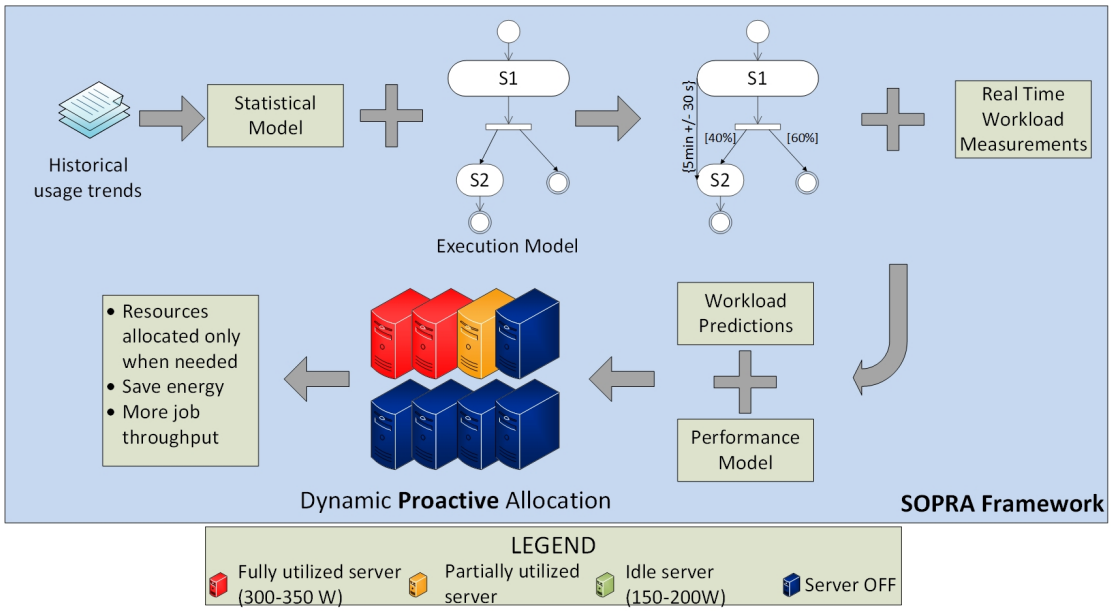

Figure 1.7. The SOPRA framework, a dynamic resource allocation strategy based on workloads predictions

procedure becomes a four-step process: 1) monitor the current usage, 2) predict the incoming workloads, 3) estimate the performance of the system under the predicted workload, and finally 4) proactively adapt the system in case a non-optimal allocation is detected.

A typical workload in a SOA consists of a sequence–it can be either fixed or varied–of service calls. The sequence of calls is commonly referred to as workflow. In this work, we use UML sequence diagrams to describe workflows. An example of a workflow is the sequence diagram shown in Figure 5.1 on page 94.

Our key insight is to look at the application layer for predicting the incoming workloads, as the application layer is the only one that contains all the information on the execution of a workflow. In fact, an Internet application can be seen as a workflow, where each service invocation is a step of the sequence. In an Internet application, service calls are correlated, and can be leveraged to predict future levels of workloads and proactively adapt the resources allocated. For instance, in an e-commerce web site, a customer is

likely to follow a sequence of requests such as searching for an item, reading the item description, adding the item to the cart, log in or register, and finally purchase the product. So, if we observe an increase in items being visualized, we can expect an increase in the number of requests to login and to purchase. This prediction, combined with a precise estimation of the impact that each request has on the system, allows one to proactively adapt the system.

Figure 1.7 describes our approach to proactive dynamic allocation of resources in a data center. Our approach, the SOPRA framework, leverages three models to achieve proactive adaptation: the Statistical, the Performance and the Execution Model. The Statistical Model contains statistical usage data extracted from historical usage trends, such as the percentage of items visited on the web site that are actually purchased. The Execution Model represents the workflow of the Internet application, like the one described above. Given a number of requests in input, Execution Model, combined with the Statistical Model, predicts the amount of requests that will hit the system in the near future. The Performance Model estimates the system behavior, including response time, given a workload and a system configuration as input. Once the workload prediction is computed, we pass it to the Performance Model to verify if whether the system is be able to satisfy the SLA and to see whether it is under under-allocated or over-allocated. With the response from the Performance Model, SOPRA proactively adapts the system configuration, if needed. With this proactive approach, the system is always correctly allocated, thereby reducing excessive energy and resource use. The system can also deliver more throughput, since either more requests or more types of loads can be funneled into the system if it is predicted to be over-allocated for the current load.

1.4 Main contribution and Dissertation outline

As summarized by Figure 1.4, service providers typically look at historical trends to decide the amount of resources to allocate for an Internet application. This leads to low utilization, which wastes resources, energy, and money for the service provider. Even though solutions currently exist to partially mitigate the resource allocation problem, there is still significant room for improvement.

The main contribution of this dissertation is a framework that proactively optimizes the resources allocated in a data center to deal with varying workloads for Internet applications. This framework allows us to: **i) predict future workloads**, **ii) estimate the impact of those workloads on the existing system**, **iii) put in place a proactive adaptation whenever needed**. *Predicting performance and, consequently, adapting the resources allocated is an effective way to contain costs, reduce energy consumption, while simultaneously satisfying the SLA*.

Figure 1.7 outlines the framework with the three models described in Section 1.3.3. Chapter 2 provides a background overview of possible alternatives for service providers to address the SLA –Resource problem. The chapter discusses strategies to dynamically adapt the amount of resources allocated in a data center. It also introduces a technique to predict the performance of an application, given a set of resources.

The research work in this dissertation is composed of three chapters that incrementally build upon the vision of a proactive dynamic adaptation. Chapters 3 introduces a highly accurate performance model based on queuing networks. We implemented a complete prototype that allocates resources in a reactive fashion based on the output of the performance model. To evaluate our solution, we implemented an existing work from scientific literature and tested both approaches using a well-known benchmark that simulates an auction web site.

Chapter 4 introduces the SOPRA framework, focusing mainly on the Execution model combined with the Statistical model. In this chapter, we shift from reactive to proactive dynamic resource allocation. To evaluate the approach, we implemented a SOPRA prototype and tested it using an Internet application based on a public API of one of the most widely used fulfillment center services. In this chapter, we also show the impressive energy savings that can be achieved with proactive adaptation.

Finally, in chapter 5 we further extend the SOPRA framework and its prediction capability. We show how the accuracy of the predictions can be further improved if the predictor is customer-aware, that is, it is able to distinguish different customer usage profiles. In fact, different customers can be associated to different Statistical models; hence their usage profiles are an important aspect that should be taken into consideration. To evaluate this approach, we extended the SOPRA framework and the implementation of the fulfillment center. The resource savings with this approach are significant.

With the work presented in chapters chapters 3 to 5, we prove that, with our proactive dynamic approach to resource allocation, for a given workload, we can achieve up to 52.49% reduction of the amount of energy used, and up to 84% fewer servers used. Our approach achieves that without introducing any additional SLA violation.

Chapter 2

Background

The previous part of the dissertation introduced the resource problem with modern data centers and its correlation with SLAs. This chapter dives into how resources are provisioned in a data center and gives an overview of a technique to estimate performance of large scale distributed systems.

2.1 Resource Allocation Strategies

This section describes different approaches to resource allocation in data centers. Section 2.1.1 discusses offline capacity planning, where resource allocation is decided before deployment. This is the de-facto standard in data centers and one of the main reasons for a very low utilization. Section 2.1.2 presents an overview of techniques that adapt the amount of resources allocated in a reactive fashion, that is, after an event like an SLA violation has occurred. Section 2.1.3 discusses techniques that take resource allocation a step further: the system adapts before a violation occurs, hence saving resources while avoiding penalties and degraded performance.

2.1.1 Offline Capacity Planning

As mentioned before, resource utilization in today's data centers is very low. The exact estimations on resource utilization vary among different studies, but all reported

figures are in the range between 18% and 40% [99, 35]. This means that, even with the most conservative estimates, more than half of all the resources of the data centers around the world are sitting idle most of the time. We mentioned already that this massive over-allocation has significant impact on costs, driven by electricity, purchase and maintenance of the servers, cooling, and overall infrastructure as more servers require more and larger data centers. This over-allocation is caused by a multitude of reasons [93], among which we mention:

- Dedicated servers for each application: In order to prevent resource contention and incompatibilities between applications and third party libraries, each application is often granted a set of dedicated servers and data center resources, e.g. switches.

- Planning for peaks: Application workloads can significantly vary over time and over different periods of the year. Studies show that peak workloads exceed the average by factors of 2 to 10 [26].

- Fault tolerance/disaster recovery: It is common practice to have a set of idle servers with the application installed as fail-over solution in case faults or disasters occur. The number of backup servers can be as large as the active servers, but since they are kept idle most of the time, they do not produce any useful result, and generate costs instead.

- Scaling: In order to be ready to address load fluctuations, lots of servers are kept idle, since adding or removing them is not an easy process, especially when dealing with the scaling of physical machines. With virtualization technologies and on-premise cloud solutions management [16, 11, 10, 14], this problem is being gradually mitigated. There are, however, many enterprises that have refrained from adopting virtualization technologies due to management reasons, strict performance requirements, and fear of adding complexity.

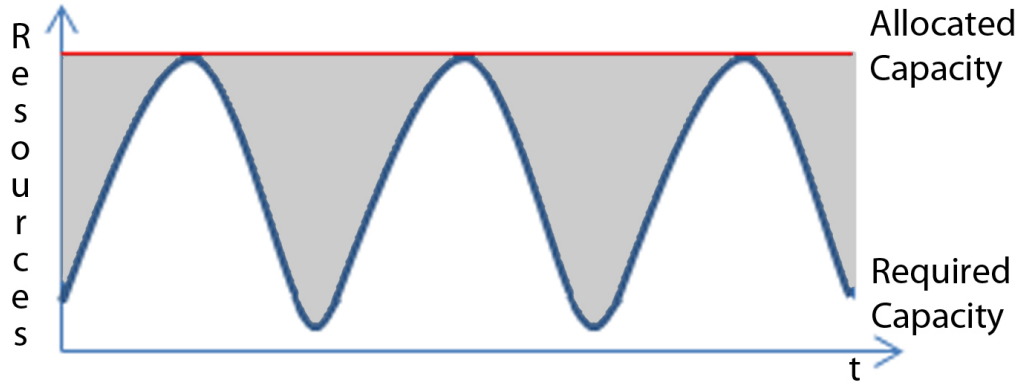

Figure 2.1. Effect of over-allocation: in gray, the unused allocated capacity. Based on figure from [26]

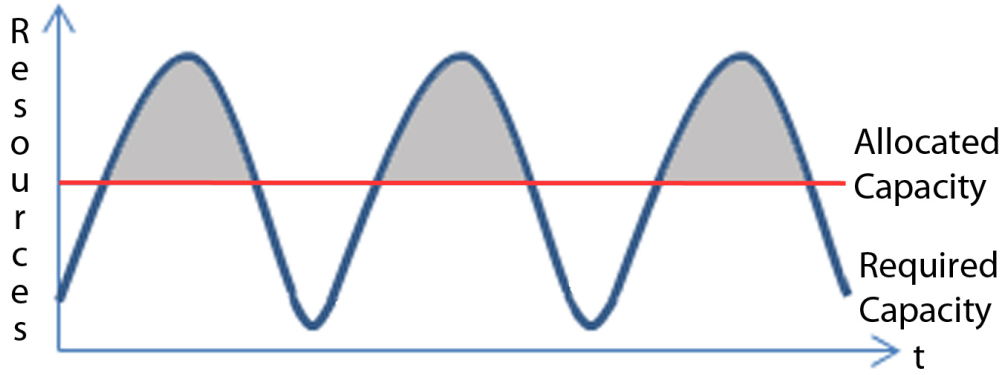

Figure 2.2. Effect of under-allocation: in gray, loss of business due to the inability to serve requests. Based on figure from [26]

Figure 2.1 shows a simplification of the effect of over-allocation. With offline capacity planning, the allocated capacity cannot be adapted, hence, in periods of low demand, much of the capacity is wasted (gray areas).

The opposite of over-allocation is under-allocation, which also causes problems. As shown in Figure 2.2, if the required capacity is larger than the allocated one (gray areas), the service provider will experience a loss of business and potential penalties due to the inability to serve customers' requests and violations to the SLA.

Since load fluctuations can be significant, unless the service provider has the ability to adapt the allocated capacity to follow the demand, it will always incur in either

over- or under-allocation.

Hence, when enterprises and data center hosters adopt a pure offline capacity planning approach, they end up over allocating resources, just in case any of these events occurs. However, most of the time these events do not occur, resulting in a massive number of servers sitting idle in data centers. Since any change to the physical servers allocated requires a human physical intervention, the offline capacity planning is also characterized by a strong inflexibility to later adaptations.

2.1.2 Planning Based on Reactive Approaches

This section gives an overview of the fundamentals of reactive approaches applied to resource provisioning. With a reactive approach, the allocated capacity is dynamically adapted when a trigger event occurs. A reactive system observes the environment, and changes itself to accommodate for any significant variations. Such event is typically the violation of some threshold, like for instance, response time, average CPU utilization, network bandwidth, etc.

Reactive systems can be fairly complex, but they are usually composed of the following blocks:

1. Monitor. The system observes the environment and its own behavior (output). What is being monitored is application and deployment specific.

2. Analyze. The monitored values are compared with thresholds to determine whether a violation has occurred. A typical example of monitored value in service-oriented applications is the response time. Let's define as A seconds the measured response time and let's call B the max response time allowed by the SLA. When A is larger B, a violation is detected and a notification is raised.

3. Plan. In this phase, based on the gravity of the violation and the available options,

a plan for adaptation is defined. A plan to react to a violated response time could require an increase in the allocated capacity by adding more servers. This step can either be manual, requiring human intervention, or autonomic. Historically, this was done by data center operators that would change the amount of resources allocated based on guidelines and their experience. Nowadays, thanks to virtualization technologies that add flexibility to data centers, the decision making process is automated and involves allocating more VMs to an application and/or reducing the level of consolidation by spreading the VMs on a larger number of physical servers. An example of the latter approach is Amazon AWS Auto Scaling[2] or Microsoft Azure [7]. Both the solutions can vary capacity up or down automatically according to conditions defined by the application admin. Moreover, it is also possible to define the intensity of the adaptation. So, for instance, the application admin could define a rule such as "if measure response time A is larger than B, add 10 virtual machines to the active pool".

4. Execute. During this phase, the plan is executed and the capacity is adapted. This step consists in executing a series of actions (if manual) or a set of scripts that adapt the system. An example of those could be a script that calls into the Amazon AWS or Microsoft Azure API to increase the number of VMs allocated.

The four steps described above represent the classic MAPE control loop [66]; Figure 2.3 shows a representation of the four steps of a MAPE loop, with the controlled application inside. Any reactive system can be decomposed into these four logical steps.

By following the MAPE steps, a reactive control loop has the functionality of adapting the allocated resources to load fluctuations, hence significantly improving over the offline capacity planning approach. However, a reactive approach has some limitations. First, such an approach reacts only once a monitoring event has occurred.

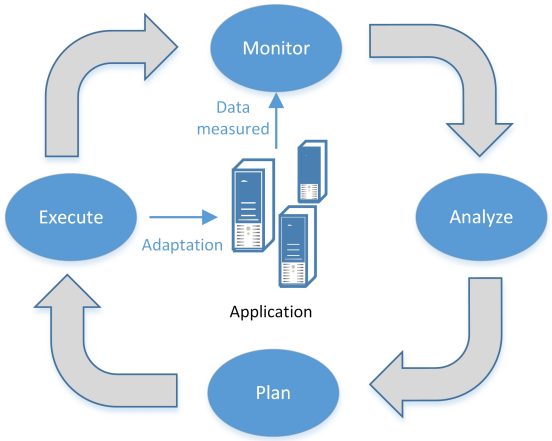

Figure 2.3. The steps of a MAPE control loop.

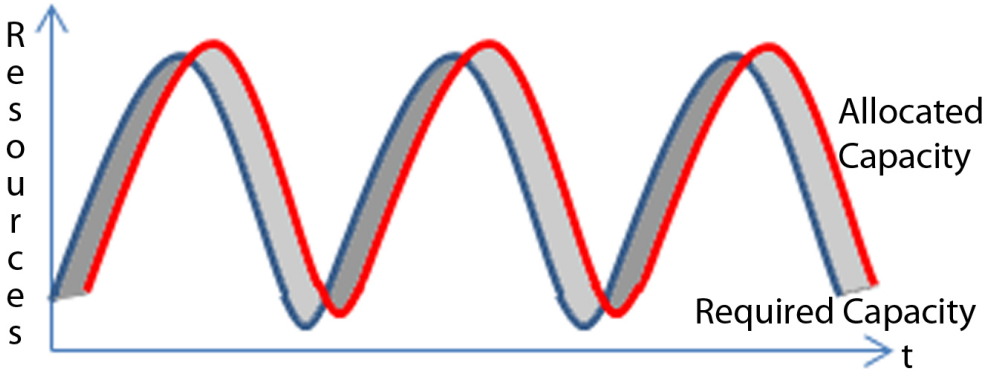

Figure 2.4. Reactive allocation. In dark gray the areas of under-allocation. Light gray are for over-allocation.

This event might verify the problem that initiated it only after a significant span of time has passed. This could create situations in which an adaptation is no longer possible or cannot solve the problem anymore. For instance, the SLA has already been violated, the provider has already been charged for the penalties or has already experienced a loss of business.

Second, the execution of adaptation activities could have an impact on the execution time of the running application, or it could take some time to complete, thus limiting its applicability. For instance, allocating new VMs takes few minutes, during which the system keeps violating the SLA. In repeated experiments done in our lab, we observed an average of 2 $\frac{1}{2}$ minutes for allocating a single VM instance of type "t1.micro" through the Amazon AWS API [3]. The measured time refers to only the instantiation of the VM, not including application setup, configuration, etc. That additional time is to be added to the overall reaction time, during which the system would still be in a violating condition.

Figure 2.4 shows the impact of a reactive approach on the allocated capacity. The light gray areas representing a waste of resources are significantly reduced compared to the offline planning shown in Figure 2.1. The allocated capacity (red line) follows the required capacity (blue line) much more closely. However, there are still areas where the two lines are not in sync, as the red line is constantly late, adding capacity after it is needed. This delay in allocating resources creates a situation where moments of under-allocation (dark gray areas) alternates with moment of over-allocation (light gray areas). In fact, when the demand decreases, a reactive system takes some time before reacting to de-allocating resources, hence creating a situation of over-allocation where resources are not fully utilized.

There are techniques that can be put in place to mitigate the limitations of reactive systems. The detection threshold could be lowered, so that enough buffer, both in terms of capacity and time, is left for reacting and adapting. The tradeoff with this kind of

mitigation is that it can potentially increase the distance between the allocated and the required capacity, resulting in an increased over-allocation.

Another mitigating technique is to increase the frequency to detect violations. This approach also has limited applicability. In the experiments described in Section 5.6, we show that there are cases in which a reactive approach could require an extremely high frequency in order to avoid SLA violations, making such approach unfeasible in practice, as the system would not be able to adapt that fast.

In summary, a reactive approach can significantly improve the allocation of resources compared to offline capacity planning, thereby reducing the amount of resources sitting idle without incurring in serious under-allocation issues, such as those shown in Figure 2.2. However, reactive approaches can only adapt once a violating event has occurred, which results in reduced applicability and effectiveness.

2.1.3 Planning Based on Proactive Approaches

Proactive approaches to resource allocation address some of the main limitations characterizing reactive based ones. The main idea behind a proactive approach is to adapt ***before*** a violating event occurs, striving to always keep the system properly allocated to satisfy the SLA.

The main requirement for proactive approaches is the ability to forecast the workload in order to be able to optimize the system allocation over the prediction window. The size of the prediction window depends on both the application and the type of traffic it is subject to. An application with a very uniform load behavior is more predictable, thus it is likely to have a larger prediction window than a hecticly patterned load. There are many available prediction techniques, which typically differ in the size of the prediction window, the accuracy, and the kind of data used for the predictions. Among the most common prediction techniques, we mention machine learning, such as

that done by Leitner et al. in [72]. The approach in [59] formulates service composition as a constrained model, where each constraint represents a QoS aspect. Predictors that leverage historical data [91] are another common approach. Finally, mathematical approaches are also used, such as the work in [28] and [93] that leverage an exponentially weighted moving average and a autoregressive moving average, respectively.

In Chapter 4 and 5, we introduce our **S**ervice **O**riented **PR**oactive **A**daptation framework (SOPRA). SOPRA uses correlations between service calls and statistical data to predict the incoming workloads and proactively adapt the system capacity. For a thorough discussion of existing predictive approaches from the literature and how they compare with SOPRA, we refer the reader to Section 4.6 and 5.2. In summary, proactive approaches address many of the limitations of reactive solutions as they adapt the system configuration before a violation occurs. All the approaches that predict the incoming volume of requests –instead of some QoS aspect as in [28]– must also determine the impact that such load will have on the currently allocated system. Hence, performance predictors must be used. One the most commonly used techniques for performance prediction are Queuing Networks (QNs). Section 2.2 gives an overview of the fundamentals of QNs. Finally, in Chapter 3 we present a QN model that predicts with high precision the performance for an Internet application.

2.2 Estimating System Performance

In the previous chapter, we discussed the pros and cons of the different approaches for allocating resources to Internet applications that are subject to variable loads. Among those approaches, we pointed out that proactive allocation is the most promising one, as it allows us to allocate resources right before they are needed, thus optimizing their usage without violating any SLA. However, many proactive techniques are based on a predictor that forecasts the incoming load. In fact, once the incoming load is known, there is still

an open question: how many resources need to be allocated in order to meet the SLA with that load?

When considering questions such as this, one must have a thorough understanding of the infrastructure in question, the application running on top of it, and all the target metrics that need to be addressed in order to comply with the SLA agreement. As discussed in [70], there are very different methodologies to answer that kind of question:

- *intuition and trend extrapolation*. The decision on how many resources would be based on the judgment of the operator/data center admin. It is easy to see how this approach can be feasible for only simple systems.

- *experimental evaluation*. This approach would require extensive offline profiling and stress testing in order to have a map for all the load points that the system can find itself in. Even though a partial performance profiling of a system is typically done before putting it in production, a full profiling would often be prohibitively expensive to do.

- *Modeling*. As defined in [70], a "model is an abstraction of a system; an attempt to distill, . . . , exactly those aspects that are essential to the systems behavior. Once a model has been *defined* through this abstraction process, it can be *parameterized* to reflect any of the alternatives under study, and then *evaluated* to determine its behavior under this alternative."

Clearly, modeling is the only viable approach for predicting the performance of a system under many load points. The benefit of modeling is that once the model is created, the systems behavior can be verified quantitatively under any alternative. Hence, by creating a model of the application, it is possible to assess how many resources the provider has to use in order to satisfy the demand, given the SLA.

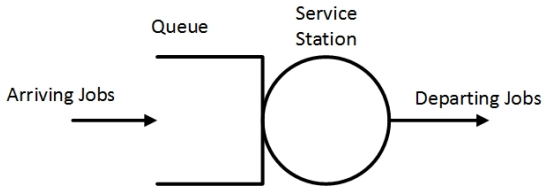

Figure 2.5. A single queue service station

A common approach to modeling computer systems is Queuing Networks. Section 2.2.1 presents an overview of the fundamentals of Queuing Networks. Chapter 3 shows the results of a research project where we leveraged QNs to implement a highly accurate performance predictor for an autonomic resource allocator of an Internet application.

2.2.1 Performance Modeling for Computer Systems

Modeling with Queuing Networks is a technique that represents a computer system as a *network of queues*. A network of queues model can then be solved analytically to gather performance relating information on the modeled system.

The basic unit of a queuing network model is the *queue service station*, like the one represented in Figure 2.5. Similarly to customers at a grocery store, *jobs* (the customers) arrive and wait in the *queue* until the queue service station (the cash register) is available. After being processed by the queue service station, the job departs and either moves to the next station or goes into the *sink*, as shown in Figure 2.6.

Queuing networks are a very powerful tool that can model a wide range of situations. First of all, there are two types of queuing networks: *open* and *closed* models. Open models are queuing networks like the one shown in Figure 2.6. Jobs enter the system via a *source*, go through the queue stations, and once they are completed, they leave the system through the *sink*. There is no correlation between the instant of time when a job leaves the system and the instant when a new job comes in, and there is an

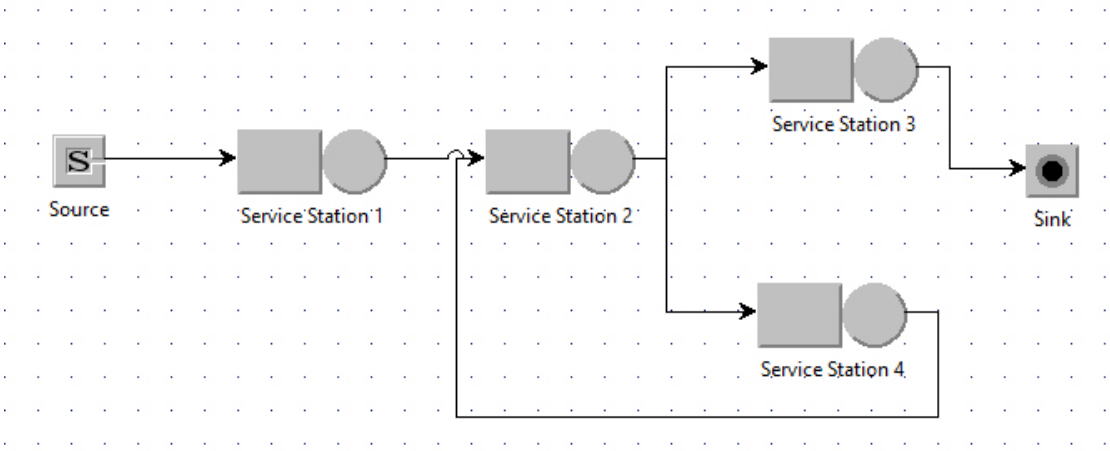

Figure 2.6. An open queuing network model, designed with the JMT tool [38]

.

infinite stream of arriving jobs. A queuing network model can be very complex and can have several parameters. However, the workload of an open model has these two parameters:

- *Workload Intensity*. Represented by λ, it defines the number of jobs arriving at the queue service station in a unit of time. For instance, if a service station receives $10 jobs/second$, it will have a $\lambda = 10$.

- *Arrival Distribution*. Defines the inter-arrival process of jobs, e.g., Exponential, Normal, Poisson, Pareto, etc.

Typical examples of open models are transaction workloads and interactive systems, where customers arrive, submit their requests (jobs) and leave once they get the response.

Instead, closed models like in Figure 2.7 are typically suited to represent batch workloads, where the number of jobs has a fixed population. The workload of closed models are characterized by the following parameters:

- *Number of jobs*. Specified by the parameter N, indicates the number of active jobs in the system. This number is constant since there is a strict correlation between jobs completed and jobs starting; when a job has completed the execution, it re-enters the system and restarts its execution.

- *Thinking time*. This is an optional parameter that is often used to represent the time a user spends thinking between issuing different requests. The thinking time can either be constant, or follow some statistical distribution. In the latter case, the thinking time will be associated with a set of parameters that describe its distribution.

So, in a closed model, there is a constant number of customers, each of them submitting one job at the time. Once a job is completed, a customer receives the response and then thinks for a certain amount of time before resubmitting another request. This loop continues indefinitely.

In both open and closed models, the time a job takes to compute at a service station is described by these two parameters:

- *Service Demand*. Represents the overall time that a single job spends at the service station during a complete execution.

- *Service Demand Distribution*. Specifies the statistical distribution of the service demand. The types of distributions are the same as for the Arrival Distribution, e.g., Exponential, Normal, Poisson, Pareto, etc.

Kendall's Notation. As mentioned above, in a QN model, each service station (often also referred to as queue) is described by two statistical distributions: one for the arrival process, and one for the service time. In 1953, D.G. Kendall proposed a notation to describes a service station; his notation has been widely adopted and it is named after

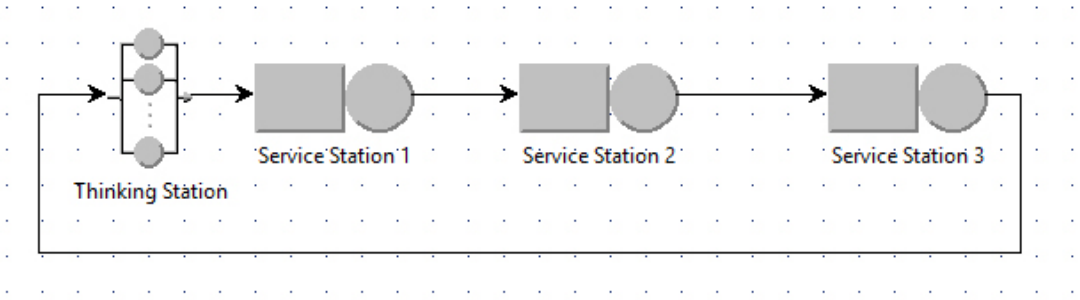

Figure 2.7. Example of closed model with a Thinking Station

him. In his initial formulation, the Kendall's notation denotes each queue with three factors, written as A/S/w:

- A. It denotes the time between jobs arrival in the queue. Each arrival distribution is associated with a letter:
  - M. The arrival distribution is Markovian, that is, it follows a Poisson process (no correlation between subsequent job arrivals).
  - G. The arrival process follows a General distribution (arbitrary distribution).
  - U. It denotes the Uniform distribution. Even though this distribution is uncommon for real life events, it is often used to study a system in ideal conditions.
- S. It stands for Service time distribution and it can have any of the statistical process listed above.
- w. Number of workers (or servers) at the service station. It indicates the number of jobs that can be executed in parallel. For example, a service station representing a CPU with hyper-threading could have $w = 2$.

So in Kendall's notation, a $M/M/1$ queue is a service station with Markovian

arrival process, Markovian service time and one server. To this basic Kendall's notation is often added the queuing policy, e.g. FIFO, LIFO, PS (processor sharing), etc. An example of this extended notation is $G/M/2-FIFO$, representing a General arrival distribution, with Markovian service time, two servers, and a First In - First Out queuing policy of the incoming jobs.

Given the set of input parameters, a model can be evaluated by solving the mathematical equations representing it. The solution yields performance results for each service station, as well as for the entire model. Among the performance parameters, there are *waiting time* (the total time spent by a job in a queue), *residence time* (the average time spent at a service station, including both queue and service time), *throughput* (number of jobs that went through a service station), *utilization* (proportion of time a service station is busy), and *system response time* (the overall execution time of a job in the system). Hence, queuing networks are a very powerful tool to evaluate the impact of a workload on a system. Performance analysis to estimate response times, utilization levels, throughput and bottlenecks of a workload, or of a set of workloads, can be easily done with queuing networks.

In a queuing network model of a computing system, the load is represented by λ and the arrival distribution, or, for closed models, by the number of customers N. The execution time of a single request is parametrized by the service demand and its distribution. Hence, it is evident that, with such a model, it becomes fairly easy to assess whether the current allocation is enough to satisfy the SLA for a given load, since the response time experienced by a customer is represented by the system response time. Moreover, with the throughput and the utilization measurements, the service provider can make informed decisions on the stress level of the entire system and can easily identify the bottleneck. As we will see in Chapter 3, this latter aspect is very important, as it allows for a greater allocation of resources where needed. For instance, if, in a multi-tier

application, the QN model identifies the application tier as the bottleneck, adding more resources to the database tier will not lead to any improved performance or any increase of the throughput. On the contrary, adding more resources to the application tier will directly benefit the system's performance, and the added capacity will be used efficiently.

Chapter 3

A Comprehensive Resource Management Solution for Web-based Systems

In this chapter, we build the first piece of our vision: the performance model. A performance model of the application allows us to correlate an input, i.e. the amount of incoming requests, with the output, i.e. the behavior (e.g. the response time) of the system, for any given set of allocated resources. To show the benefits of having a performance predicting capability, we introduce a reactive system that leverages a highly accurate performance predictor and a finely grained monitoring system to tailor the allocation of resources for an e-commerce web site. Compared with an approach from scientific literature, our approach yields up to a 42.5% reduction in the amount of resources allocated.

3.1 Introduction

Modern software systems are subject to continuous change, such as changes in the stakeholder requirements, evolutions in the software or resource components, and variations in the execution context [32]. System designers need refined runtime management techniques and tools to cope with these changes, so that the systems can

continue to provide the required functional and non-functional qualities.

In this paper we focus on complex Web-based systems. In these cases change often manifests itself as significant fluctuations in the systems' workload. Although they often follow historical and seasonal patterns, multiple spikes may occur without warning (e.g., flash crowds [20]). These situations can lead to frustrated customers and loss of online business. Consequently, service providers tend to over-allocate resources. As a result, the average server utilization in a typical data center is around 30-40% [35]; some studies even estimate utilization levels as low as 18% [99]. A better utilization of resources would lead to more efficient systems. Fortunately, the diffusion of dynamic resource allocation techniques provides the flexibility needed to build systems that can adapt their configurations based on the actual workloads and acquire resources on demand.

These systems must embed fine-grained *autonomic* capabilities that cover all their facets, from their hardware resources to their software. By introducing a MAPE (Monitor, Analyze, Plan, and Execute) control loop [66] that can *estimate* the application's load and performance, and help understand how many resources should be provisioned, we can compute and apply anticipatory or corrective actions on the fly.

Many different performance models have been proposed in the past. Unfortunately, in their load estimation, they tend to only take into account the volume of requests (e.g. [103, 105]). Regrettably, it has been shown that this is not enough to estimate an application's resource usage effectively [98, 100]. Some systems take a further step and claim to be workload-mix aware, meaning that they distinguish between different types of requests, based on their aggregated response times. Yet these approaches still do not consider the hardware resource usage that these requests imply, and the resource contention that can arise. Resource contention is one of the main causes of performance degradation in systems with high loads [47, 94, 67, 81]. Knowing the resource usage

profile of each of the different types of requests is vital in defining an effective allocation strategy.

Our approach provides a comprehensive, innovative solution for the *autonomic* management of complex Web-based applications. We exploit and suitably extend ECoWare [31], a monitoring and adaptation framework previously developed at Politecnico di Milano for service-based systems. Thanks to these extensions, we are now able to perform fine-grained measurements of a Web-based system's behavior by correlating the runtime data that we retrieve from its hardware and software. To achieve this, we included, in ECoWare, a completely new performance model that takes into account both the application's workload mix and the hardware infrastructure. It defines the resource footprint of the different application requests in terms of CPU instructions, cache, and main memory accesses. These metrics are then used to profile the resource usage of the requests and to increase the accuracy of the performance predictions.

To evaluate our approach, we used RUBiS [15], a well-known benchmark that simulates a common auction and shopping web site, and compared our results against a baseline approach and an existing solution from literature [98].

The rest of the paper is organized as follows. Section 3.2 provides a high-level overview of our autonomic solution. Section 3.3 describes the techniques and tools that we use to monitor applications and analyze their behaviors. Section 3.4 explains how we use a performance model and a planning algorithm to understand to what extent we need to change the way we provision resources. Section 3.5 illustrates how the autonomic solution can be applied to a real-world application, and provides a thorough empirical evaluation of our approach. Section 3.6 surveys the state of the art, and Section 3.7 concludes the paper.

3.2 Approach Overview

The overarching goal of our ECoWare project is to empower system designers and maintainers in the development of self-adaptive systems. Although, in this paper, we focus on complex Web-based systems, the approach does not rely on any specific implementation technology. Instead, ECoWare can be seen as a general-purpose monitoring and adaptation framework that exploits the common *MAPE (Monitoring – Analysis – Planning – Execution) control loop* approach. System designers can tailor ECoWare to their needs by choosing appropriate technology- and application-specific sensors and actuators, and by choosing appropriate analysis and planning techniques.

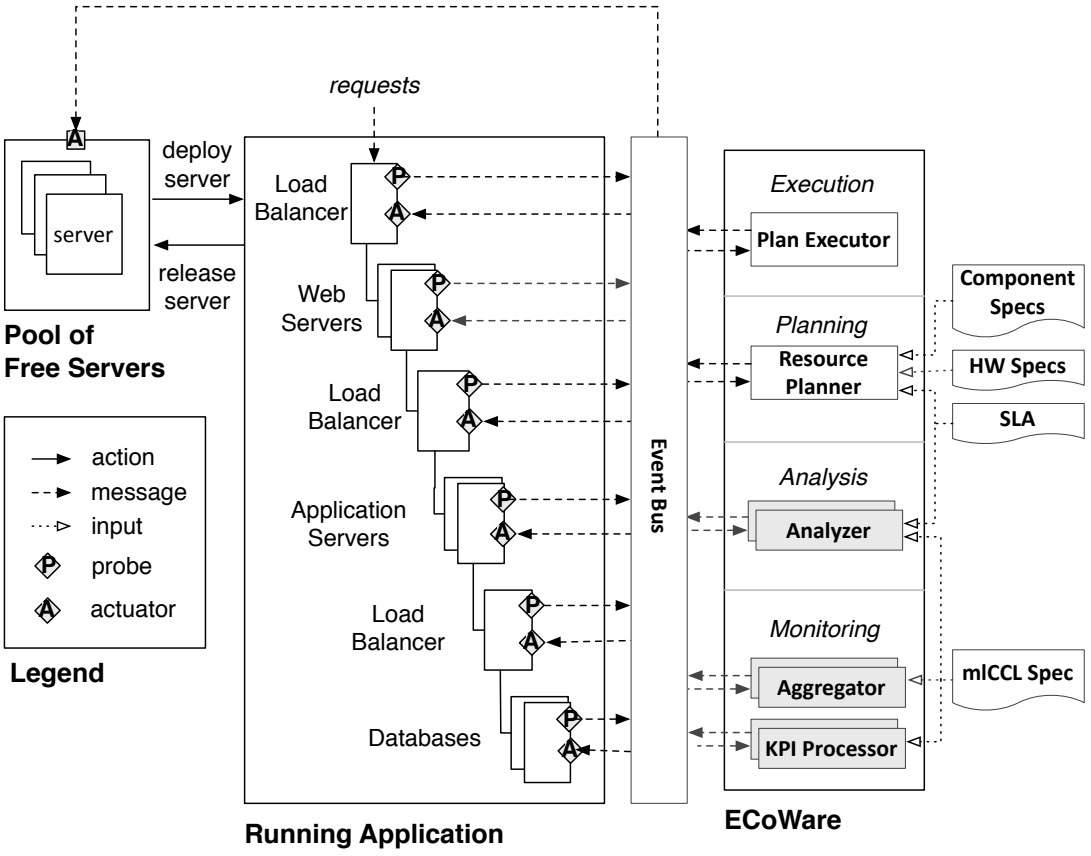

Figure 3.1. Example of an Internet application with ECoWare.

Figure 1 shows how ECoWare can be applied to a complex Web-based system.

The application uses multiple servers for each tier, and new servers can be added from a *Pool of Free Servers*. The application is on the left-hand side of the figure, and it interacts with ECoWare, which is on the right-hand side of the figure, through an Event Bus. Light gray components represent previous work, while white components are novel contributions of this paper.

As previously stated, ECoWare requires that technology- and application-specific probes and actuators- be deployed to the running application. The probes produce low-level runtime events and send them to the bus so that they can be consumed and manipulated by ECoWare; the actuators, on the other hand, wait on the bus for adaptation directives from ECoWare.

For the **Monitoring** step, ECoWare provides two kinds of components: *Key Performance Indicator (KPI) Processors* and *Aggregators*. The former use the low-level events produced by the probes to calculate various non-functional KPIs, such as average response times, arrival rates, and throughputs. The latter correlate multiple low-level events and/or KPIs to produce a holistic understanding of the application's behavior. Both produce new outputs that are sent back to the bus for further consideration. This allows us to compose processing components, using the pipe-and-filter style, to create complex processing.

For the **Analysis** step, ECoWare's *Analyzers* listen to the bus for monitoring events (e.g., average response time) and evaluate them against certain constraints. This allows them to determine whether a Service Level Agreement (SLA) has been violated, and whether we need to plan a new resource provisioning.

For the **Planning** step, ECoWare provides a *Resource Planner*. This component uses a workload-mix and hardware-aware performance model to understand what changes should be made to the resource provisioning.

Finally, for the **Execution** step, ECoWare provides the *Plan Executor*. This

component coordinates multiple actuators to accomplish the desired adaptations.

ECoWare is extremely flexible; indeed, it is able to support multiple kinds of management strategies, e.g., *reactive*, *proactive*, and *periodic*. We can even mix different strategies together to differentiate how we up-scale from how we down-scale our resources. For example, we could employ a reactive approach to up-scaling. The reaction window would depend on the nature of the SLA chosen, and it would allow us to timely adapt to sudden increases in the workload volume, or changes in the workload mix. On the other hand, we could adopt a periodic strategy for down-scaling. In this case, the frequency with which we check our resource usage would be determined by the application's deployment environment. This flexibility allows us to be fast to provision new resources, yet cautious when un-provisioning them; and it allows us to reduce instability.

The **Monitoring** and **Analysis** steps will be discussed in Section 3.3; the **Planning** and **Execution** steps will be discussed in Section 3.4. First, however, we will provide a brief overview of the running example we will exploit in the rest of the paper.

3.2.1 RUBiS

In order to evaluate our approach, we used the well-known RUBiS benchmark application [15]. RUBiS implements the core functionality of an auction site: browsing, selling, and bidding. It was developed using standard HTML, Java Servlet, and SQL technology. RUBiS uses an Apache server for its web tier, a JBoss Application Server for its business tier, and a MySQL Server for its backend data tier. In our deployment we decided to distinguish how the application serves static content (e.g., the home page, images, etc.) from how it serves dynamic content (e.g., the user's bidding history, the products available in a given geographical region, etc.). A single Apache server serves the static content, while up to three JBoss servers are used to serve the dynamic content. A content-aware load balancer (not originally a part of RUBiS) was added to route the

requests accordingly.

All the servers, as well as ECoWare, were deployed to different physical machines. For our experiments we used seven servers equipped with Intel Xeon processors running at 2.66 GHz. A processor is composed of two cores, each addressing a separate 6MB L2 cache; main memory is 32 GB on each machine. We deployed RUBiS' components (i.e., Apache Web Server, JBoss and MySQL database), the Event Bus and the monitoring and analysis parts of ECoWare on machines running Ubuntu 12.04 LTS; the Resource Planner, the Plan Executor, the load balancer, and the load generating clients were deployed on Microsoft Windows Server 2008 R2.

Although ECoWare can be used to manage many different kinds of SLAs (e.g., throughputs, costs, etc.), due to space constraints we will focus on how we use ECoWare to enable fine-grained resource provisioning for an Internet application like RUBiS, while satisfying an SLA of the kind "the X^{th} percentile response time should be within a fixed threshold over a period of time". The percentile response time is calculated by taking into account the response times seen by ECoWare over a two and a half minute window.

3.3 Monitoring and Analysis

The **Monitoring** and **Analysis** steps evaluate whether an application is complying with its SLA.

As previously stated, ECoWare provides two kinds of processing components for the **Monitoring** step: *KPI Processors* and *Aggregators*. KPI Processors and Aggregators are developed using Esper[1], a well-known tool for complex event processing (CEP). CEP allows us to analyze incoming streams of events against complex patterns that are defined using an Event Processing Language (EPL). In the case of Esper, the EPL is an SQL-like language through which we can express conditions, correlations, and aggregations over

[1]Esper – http://esper.codehaus.org

multiple incoming streams of events.

KPI processors can calculate well-known KPIs (e.g., average or X^{th} percentile response times, arrival rates, throughputs, etc.), as well as application-specific indicators. For each of the pre-defined indicators, ECoWare defines a basic Esper EPL template, which is completed at deployment time when we provide the indicator's configuration. Application-specific indicators, on the other hand, require full EPL declarations.

Aggregators compose multiple events, coming from different independent sources (i.e., probes or other processing components), to produce a new event that can be sent back to the Event Bus. Since different sources produce events at different times and rates, an aggregator needs to establish when it should operate. To do this, it relies on the definition of a "primary event". When the primary event appears on the Event Bus, the aggregation is triggered, and the "secondary events" (i.e., the other events that need to be collected) are gathered from their corresponding incoming streams. ECoWare's aggregators are also implemented using Esper EPL templates.

Analyzers evaluate incoming streams of events against data constraints that are defined using a modified version of the WSCoL (Web Service Constraint Language) assertion language [30], a language for defining pre- and post-conditions over distributed Web service interactions. Since an analyzer requires all the data to be available at the time of the analysis, analyzers are often used in conjunction with aggregator components.

In order to analyze the behavior of the RUBiS benchmark, we compared the X^{th} percentile response times of each request type with an SLA threshold. To do this, we had to develop technology-specific probes, and connect them to specific KPI processors, Aggregators, and Analyzers. More in detail, we needed to create a Servlet Probing Filter with nanosecond precision for use within JBoss. This probe generates an event every time a request is made to a servlet, and every time the corresponding response is sent out.

We also had to produce probes that could collect the data needed to configure

and use the performance model that we will discuss in Section 3.4.2. In particular, we needed to create a probe for the load balancer, to gather data on the inter-arrival times of the requests; a second probe for JBoss based on Hibernate, to calculate the amount of time that each servlet spends interacting with the MySQL database; and a probe for gathering information about the hardware resources being used.

We also wanted to identify the contribution that each servlet invocation has on the following three kinds of resources: CPU, cache, and system RAM. The operating system virtualizes the CPU core resources by means of OS threads. While Linux supports collecting information on a per-thread basis through the `perf` utility, it was not a viable option due to the overhead it introduces with respect to our desired granularity. Instead, we turned directly to the hardware itself, which provides per-core performance counters that can be programmed to count several hardware events. Since we wanted to measure each servlet call, and servlets are executed by threads, we extended the standard Linux kernel to provide per-thread information, directly within the OS's thread `sched` files[2].

Figure 3.2 shows how ECoWare was used to produce $\overline{R}$ and $\overline{\lambda}$. The former is the vector of X^{th} percentile response times per request type; while $\overline{\lambda}$ is the vector of arrival rates per request type. The reason why we distinguish between request types and produce these vectors, and do not aggregate all the response times and all the arrival rates, is that our approach needs to be mix-aware. We want to be able to take into account the fact that different request types use the system's resources differently.

The components placed above the event bus in Figure 3.2 calculate vector $\overline{R}$, while those placed below the event bus calculate vector $\overline{\lambda}$. The arrows show the event routings that are established within ECoWare's event bus to appropriately string together the KPI processing components in a mix-aware fashion.

[2]The modified kernel we created to run our experiments is available at https://sosa.ucsd.edu/confluence/x/tAD0.

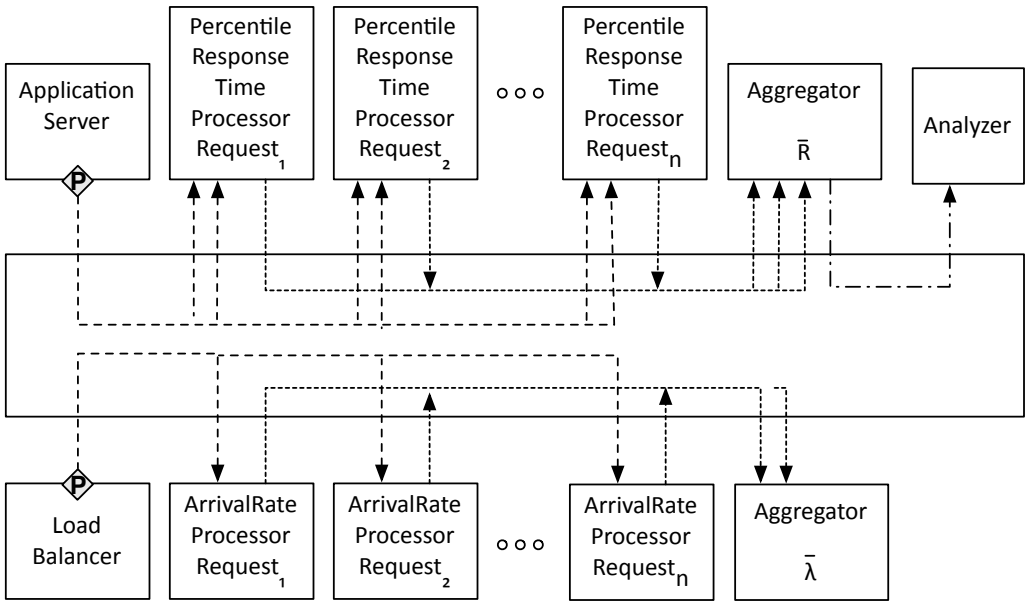

Figure 3.2. Monitoring and Analysis in the RUBiS.

To calculate $\overline{R}$ we configured the JBoss probes to send out a first event every time they received a new request, and a second event every time they produced the corresponding response. We then prepared one *Percentile Response Time Processor* for each of RUBiS' request types, and routed the event couples accordingly. These processors were configured to output a new percentile response time every two and a half minutes, while taking into account the event couples received in those two and a half minutes. The decision of this time-frame was made in conformity with the established SLA. The *Aggregator* component was then used to create vector $\overline{R}$, so that it could be sent to the *Analyzer* processor. Finally, the *Analyzer* was configured to produce a violation event whenever any value in $\overline{R}$ was greater than R_{SLA}.

As we can see in the lower half of Figure 3.2, we adopted a similar approach for $\overline{\lambda}$. In this case, we configured our load balancer probe to send out a new event every time a new request was received. Once again we had one *Arrival Rate Processor* per RUBiS request type, and we routed the events accordingly. These processors were configured to

calculate and output a new arrival rate every two and a half minutes, while taking into consideration the events received in those two and a half minutes. Once again this was done in conformity with the SLA. The *Aggregator* was then configured to take these arrival rates and produce $\overline{\lambda}$.

3.4 Planning and Execution

In this section we focus on the **Planning** and **Execution** steps of our MAPE control loop. These steps are implemented by the *Resource Planner* and the *Plan Executor* components respectively (see Figure 3.1).

The Resource Planner is responsible for identifying the correct amount of resources to add or remove from the running system and for communicating to the Plan Executor what actions it should perform. In order to do this, it implements the *Compute Configuration Algorithm*. Every decision that the algorithm makes is checked against a hardware- and workload-mix aware *Performance Model*. This allows us to avoid a trial-and-error approach in which we deploy (or remove) hardware and then wait for the subsequent iteration of the control loop to tell us if the problem was solved.

3.4.1 Compute Configuration Algorithm

Every time the Resource Planner is notified with a SLA violation, it invokes the algorithm illustrated in Algorithm 1. The algorithm takes the vector of arrival rates per request type ($\overline{\lambda}$), the response time value defined in the SLA (R_{SLA}), and the current system configuration ($SysConf_{Current}$), and determines a new system configuration ($SysConf_{New}$) that satisfies the SLA.

The algorithm takes a server S from the pool of free servers ($freeServerPool$) and adds it to the current system configuration to create $SysConf_{New}$. It then verifies the new system configuration by running *AnalyzePerf*, which solves a queuing network

Input: $\overline{\lambda} = (\lambda_1,\ \lambda_2,\ \ldots,\ \lambda_n)$, R_{SLA}, and $SysConf_{Current}$
Output: $SysConf_{New}$
if $freeServerPool = \emptyset$ **then**
  return $SysConf_{Current}$
else
  select server $S \in freeServerPool$
  $SysConf_{New} = SysConf_{Current} \bigcup S$
  $\overline{R_{New}} = AnalyzePerf(\overline{\lambda}, SysConf_{New})$
  if $\overline{R_{New}}$ *satisfies* R_{SLA} **then**
    return $SysConf_{New}$
  else
    return $ComputeConfiguration(\overline{\lambda},\ R_{SLA},\ SysConf_{New})$
  end
end

Algorithm 1: Compute Configuration Algorithm

(QN) model, i.e., the Performance Model, for $SysConf_{New}$ and the actual workload $\overline{\lambda}$. *AnalyzePerf* returns a vector of response times $\overline{R_{New}}$. If all the values of $\overline{R_{New}}$ are smaller than R_{SLA}, the algorithm terminates returning $SysConf_{New}$; otherwise, the algorithm recursively calls itself with parameters $\overline{\lambda}$, $SysConf_{New}$, and R_{SLA}. In the worst case scenario, i.e., in which it continuously fails to satisfy the SLA, the algorithm continues until all the available resources (i.e. servers) are provisioned.

As previously mentioned, the autonomic system will periodically check its provisioning to see whether resources can be safely removed. In this case, the Resource Planner invokes Algorithm 1 by passing a base configuration of one server as $SysConf_{Current}$. The algorithm recursively adds servers untill the queuing network (QN) solver validates the configuration. By doing so, ECoWare guarantees that the minimal set of servers is always allocated.

3.4.2 Performance Model

Algorithm 1 leverages a QN model to estimate the application's response times per request type, given a specific workload and a specific system configuration, allowing

ECoWare to compare the estimated values with those required by the SLA.

In a QN model (e.g., [40]), each component is represented as a queue, called a service station. A model can be closed, open, or mixed, depending on whether the volume of requests is constant, fluctuating, or mixed. Whenever the number of incoming requests is higher than the computing capacity of a service station, the requests are queued; the time spent in queue is called waiting time. Each type of service request is defined by an arrival rate process (i.e., how the arriving requests are distributed in a unit of time) and a service demand distribution (overall time that a single request spends at each of the service stations during a complete execution). This can be expressed using Kendall's notation in the form $A/S/C-POLICY$ [40]. A describes the arrival process, S describes the distribution of service time of a job, C describes the number of servers at the node, and $POLICY$ describes the queuing discipline used at that node (e.g., first come first served, processor sharing, etc.). In Kendall's notation, M stands for Markov or memoryless, meaning that arrivals occur according to a Poisson process; and G stands for general, meaning an arbitrary probability distribution.

As the arriving traffic of the different types of requests fluctuates over time, so does the mix of requests in execution. Consequently, resource contention changes over time. Figure 3.3 shows the performance model of our running RUBiS example. To simplify the presentation, the figure only shows one server (with two cores) for the application tier. The model correctly captures the following three aspects: the fact that the approach is workload mix-aware, the fact that the system is composed of multiple tiers, and the fact that the application tier relies on specific hardware resources.

The intensity of the arriving flow of requests in input to the queuing network, that is the workload-mix, is described by $\overline{\lambda}$, the vector of arrival rates per request type. Regarding the fact that we have multiple tiers, the model concentrates on the application tier (JBoss) and on the database tier (MySQL), and leaves out the web tier (Apache). The

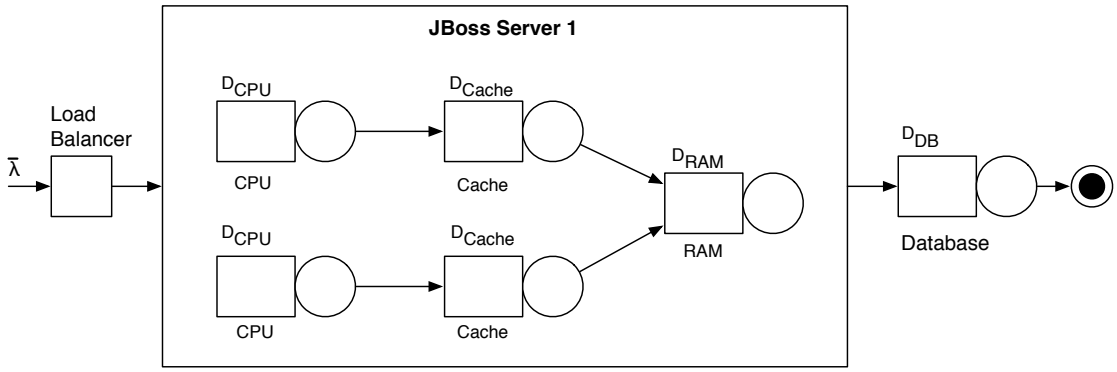

Figure 3.3. Performance Model in our experiments.

reason for this is that, in RUBiS, system performance is dominated by the application tier. The web tier only serves static content and has a negligible influence on the performance of the overall system. The database also shows a very moderate load, but we could not remove it entirely from the model since the JBoss servers query it. However, given its low utilization and resource contention, the role of the database was approximated by a single queue in the performance model.

In the application tier, i.e., in the dominating tier, we used a very fine-grained approach. We used multiple M/G/1-FCFS stations for the main hardware components of the application servers i.e., CPU, cache and main memory. (These were configured using data taken from our hardware probes.) This means that not only we consider the traffic of incoming requests and their mix, but we also consider the resource usage footprint of each request. On the other hand, we did not need this level of detail either for the database tier or for the incoming load balancer. Therefore, we used a single M/G/1-FCFS station in both cases, and configured them using information collected by the Hibernate and Load Balancer probes, respectively. The results of our experiments, which we will discuss in the following section, show that these simplifications are correct.

By modeling the service demands for the resources, our model can estimate the waiting time of a request at each service station. Whenever a resource becomes congested,

the waiting time at the corresponding service station increases. Since each type of request has a different response time and a different resource usage profile, different workload mixes (i.e., requests mixes) will use the hardware resources in a different way. As a result, the overall execution time is both mix- and volume-dependent.

Let us use a simple example to further exemplify our solution. Let us assume we have two types of requests, A and B, and that request type A is CPU-intensive and request type B is memory-intensive. If we mostly receive requests of type B, their total execution time, and the overall system's throughput, will suffer. The reason for this is that the cache will start being used by a high number of threads, and start becoming congested. When this happens, RAM will be accessed more frequently, resulting in longer waiting and execution times. A more balanced load between requests A and B may cause less contention, and the overall system's throughput would benefit.

Table 3.1 shows the increase in terms of response time, and number of cache and main memory accesses, for a memory intensive workload. Two different hardware architectures are compared: although they have the same CPU frequency and number of cores, HW_1 has twice the cache size per core than HW_2. The table clearly shows that the architecture with the larger cache size experiences much smaller performance degradation. Moreover, at high load levels (about 80% CPU utilization), the system with the smaller cache shows a 49% increase in time spent to access the main memory because of the increased cache conflicts. This demonstrates that reducing resource contention between running tasks is an effective way to improve a system's throughput, without sacrificing response times (this was also shown in [47, 42, 94]), and that explicitly modeling both the hardware and the resource usage profile of the requests can be an effective way to take resource contention into account, and to improve the accuracy of our performance predictions.

Table 3.1. Increase of response time (RT), cache accesses (CA), and memory accesses (MA) from low to high load levels for different hardware architectures.

HW Conf.	RT	CT	MA
HW_1	6%	$\sim 0\%$	19%
HW_2	26%	3%	49%

3.4.3 Plan Executor

Whenever the Resource Planner determines that the system configuration should be updated, the Plan Executor reconfigures the system accordingly. As shown in Figure 3.1, the Plan Executor may need to leverage multiple technology- and application-specific actuators to enable a reconfiguration. For example, it could depend on what kind of resource needs to be added (e.g. physical servers vs. virtual machines, virtual machines from a private cloud vs. a public one), what tiers can be modified (e.g. increasing the number of servers for the database tier can have some major intricacy), or what kind of control knob is available (e.g. changing the number of CPU cores or the load balancing strategy).

In this work we focused on allocating (or de-allocating) physical servers to the application server tier. This requires actuators that can move servers between the free servers pool and the application server tier, as well as actuators that can update the application's load balancer with the new list of servers being used. The load balancer was developed using Apache Camel, a Java-based implementation of the Enterprise Integration Patterns (EIP). It uses a Dynamic Router EIP component to capture an incoming request, understand whether the client is asking for static or dynamic content, and route the request according to the servers in use. In the case of requests for dynamic content, the requests are dispatched with a round robin policy. The load balancer's actuator listens to the Event Bus for configuration changes and updates the load balancer's configuration accordingly.

3.5 Experimental Evaluation

For the evaluation of this work, we focused on assessing ECoWare's frugality in allocating resources while satisfying the SLA; our experiments show up to 42.5% improvement compared to the state of the art [98]. Thanks to ECoWare's great flexibility, we were able to set up different experiments, featuring different allocation strategies, with minimum change to the infrastructure. We adopted the queuing network model already illustrated in Figure 3.3. To create and solve the model, we used the Java Modeling Tool JMT [38].

3.5.1 Setup of the Performance Model

During our initial evaluation of RUBiS, we loaded the system to measure the servlets' resource usage profiles. These were then used to calibrate the queuing network performance model. Since the servlets' execution times are mostly determined by the queries they send to the database, and by the amount of data they get back, we focused our study on two servlets with fixed queries: `ViewUserInfo` and `ViewBidHistory`. We measured the system at different load levels and compared the results with the output of the queuing network model: the average error for the response time was 8%. Since the servlets' average execution times were very close, and often below 1ms, we decided to increase the load for servlet `ViewBidHistory` to make it more representative of a real-case scenario —in the RUBiS' original database, the average number of bids per item is only 0.21. Hence, we increased the number of bids, obtaining a servlet baseline execution time of 88ms. We also added a routine to `ViewUserInfo` that validates the content of the comments that are left about a given user (as typically done in many social web sites). By doing so, `ViewUserInfo` required 160ms to compute on average. The larger number of bids and the text validation routine allowed us to increase the execution

times by one to two orders of magnitude with respect to the original servlets.

3.5.2 Experiments

In this section, we evaluate the accuracy of ECoWare's provisioning strategy with multiple non-stationary workloads. Since provisioning decisions are applied independently at each tier, we focused on the application tier and over-provisioned the others to ensure that they did not represent bottlenecks.

To validate our approach, we compared it with a baseline solution that uses an M/M/K-FCFS queue as its performance predictor, and with the approach presented by Singh et al. in [98], where they use a closed formula approximation of a G/G/1 queue to predict server capacity for a given workload mix[3]. For this evaluation, we followed a very similar approach to the one used in [98], that is: i) we defined the SLA in terms of the $95th$ percentile of response time and we set a threshold of 0.5 seconds; ii) we took a public web benchmark (in our case RUBiS); iii) selected a few servlets; iv) created two different workloads, and v) reconfigured the load balancer to distribute requests evenly among the available servers.

The first workload was designed to follow the structure of the workload used in [98]: it has a varying mix, yet a constant volume of requests. The second workload has both a varying workload mix and a varying volume of requests. Both workloads were executed several times and exhibited consistent results. For the sake of simplicity, we show here a randomly selected execution.

Varying-mix Workload with Constant Volume

The first experiment shows how provisioning decisions are affected by variations in the workload mix of the two types of requests, given a consistent volume of requests.

[3] In order to perform our comparisons we implemented Singh et al.'s approach according to the formulae presented in their paper.

Workload One. The workload was 110 minutes long and had a constant total volume of 15 requests per second ($\lambda = 15 req/s$), with a mix of two types of requests that changes every 10 minutes. The value for λ was chosen small enough to not saturate all the available resources; the 10 minute interval was taken from the experimental setup of [98]. The requests' arrival rate follows an exponential distribution. Figure 3.4 shows the mix of RUBiS' requests that are sent by the clients. The workload starts with 100% of requests of type `ViewBidHistory`. For the following 50 minutes, every 10 minutes, a 20% decrease of `ViewBidHistory` requests is matched by a 20% increase of `ViewUserInfo` requests. This causes the average response time to increase, as `ViewUserInfo` is a heavier type of request. At the beginning of the second hour, `ViewUserInfo` starts decreasing by 20% at each step while `ViewBidHistory` climbs back up.

Server Provisioning and SLA. To construct a fair comparison between all three approaches (i.e., ours, the baseline, and Singh et al's.) the provisioning logic was invoked every 150 seconds (i.e., every two and a half minutes) with data on service time, response time, arrival rate and inter-arrival rate. This time frame is consistent with the chosen SLA, and with how we configured ECoWare's monitoring and analysis steps. Figure 3.5 shows how the provisioning strategies behave in terms of allocated servers for workload one, while Figure 3.6 shows the average response time every 150 seconds w.r.t. the 0.5 second SLA line. From Figure 3.6 we can observe that the baseline approach tends to under-allocate, resulting in many SLA violations. The figure also shows that the baseline solution exhibits a lot of instability. In order to mitigate this phenomenon, we enforce that de-allocation cannot occur unless three consecutive de-allocation requests arrive; in other words, the provisioning logic must consider the infrastructure as over-allocated for at least 7.5 minutes. This rule was enforced with all three solutions.

The provisioning strategy from [98] does not incur in any SLA violation, but this comes at the cost of a higher number of servers allocated. The almost-flat line of

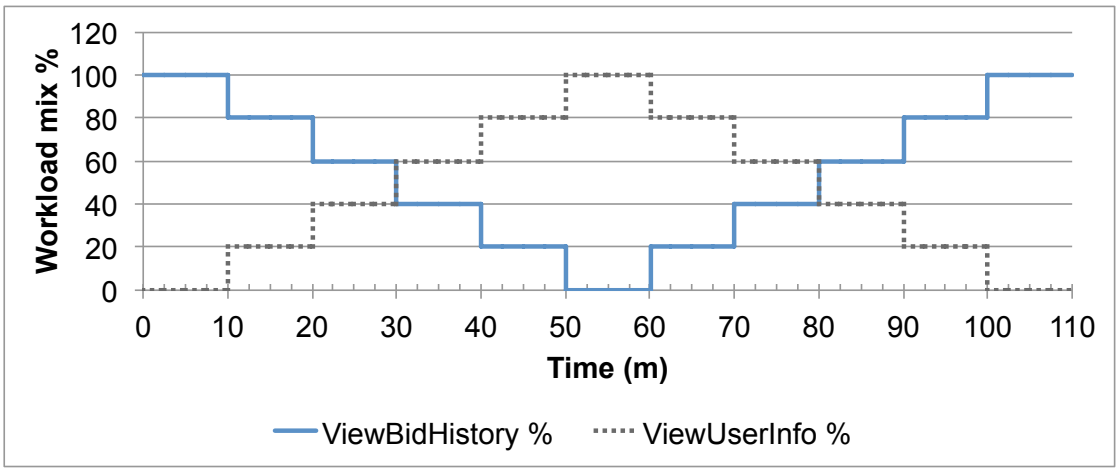

Figure 3.4. Requests mix for workload one.

the response time across the entire workload further confirms over-allocation. Between $t = 52.5$ and $t = 62.5$ minutes, four servers are requested by the provisioning algorithm. Since we only had three application servers at our disposal for the experiment, the plan executor was only able to allocate a total of three during that time interval. This does not introduce any bias in the experiment, since the provisioning algorithm from [98] is memory-less, i.e., it does not use any information on previous decisions nor on the current state of the infrastructure. Moreover, this can also happen in a real scenario whenever the provisioning request is higher than the availability of the data center.

ECoWare, on the other hand, experiences two minor violations, which are promptly handled by increasing the number of servers. The decrease-increase step between $t = 30$ and $t = 32.5$ minutes is expected. In fact, ECoWare correctly de-allocates one server at $t = 30$ minutes after seeing three consecutive de-allocation requests. However, ECoWare cannot be aware that, at $t = 30$ minutes, the workload is going to start changing, wich would require more servers; instead those violations are expected with a reactive approach. The monitoring infrastructure reports the change at the next interval, which occurs at $t = 32.5$ min, and ECoWare's provisioning logic correctly allocates an additional server. ECoWare also shows a correct symmetrical de-allocation behavior

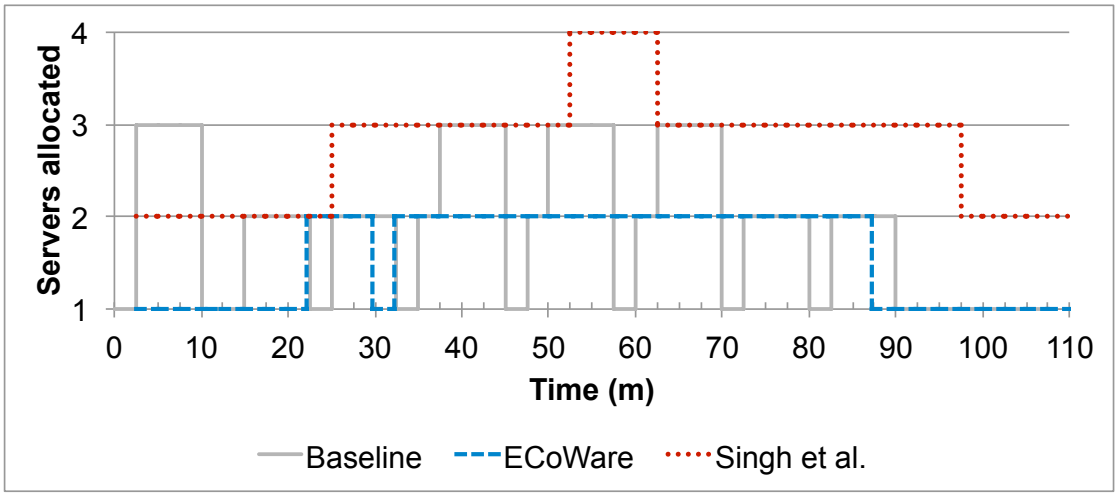

Figure 3.5. Number of servers allocated by the three solutions for workload one.

when the workload decreases after $t = 87.5$ min.

Result. To quantitatively assess the difference in server provisioning, we multiply the number of servers by the time they were allocated, a measurement unit that we call server-minute. The baseline approach shows a value of 200 server-minutes but with a high number of SLA violations. The approach in [98] reaches 300 server-minutes, without any SLA violations. Finally, ECoWare uses 172.5 server-minutes with only two minor SLA violations, which are consistent with the expected behavior of a reactive approach that is not over-provisioned. Hence, ECoWare used 13.75% fewer server-minutes than the baseline and 42.5% less than [98].

Random Varying Workload Mix and Volume

In order to evaluate ECoWare in a closer-to-real situation, where the workload mix can change in intensity at any time, we randomized both the intensity of the changes in the mix and the time length of the steps.

Workload Two. Figure 3.7 depicts workload two, featuring its workload mixes and the different time lengths of the steps. The maximum total number of requests per second at each step was limited to 22.5 in order to generate a workload that could be

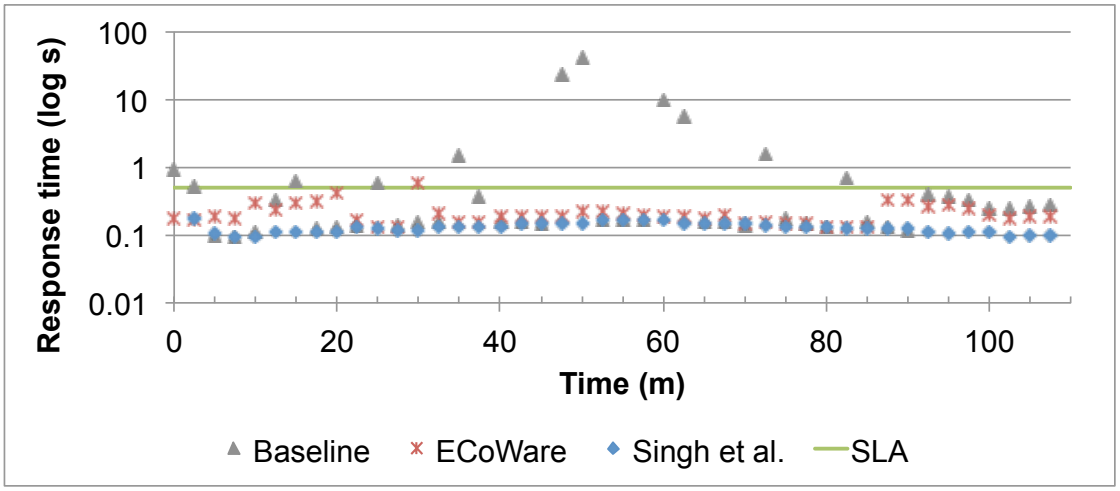

Figure 3.6. 95% Percentile response time for workload one.

handled by the amount of servers at our disposal. The same randomly generated workload was used for all three solutions.

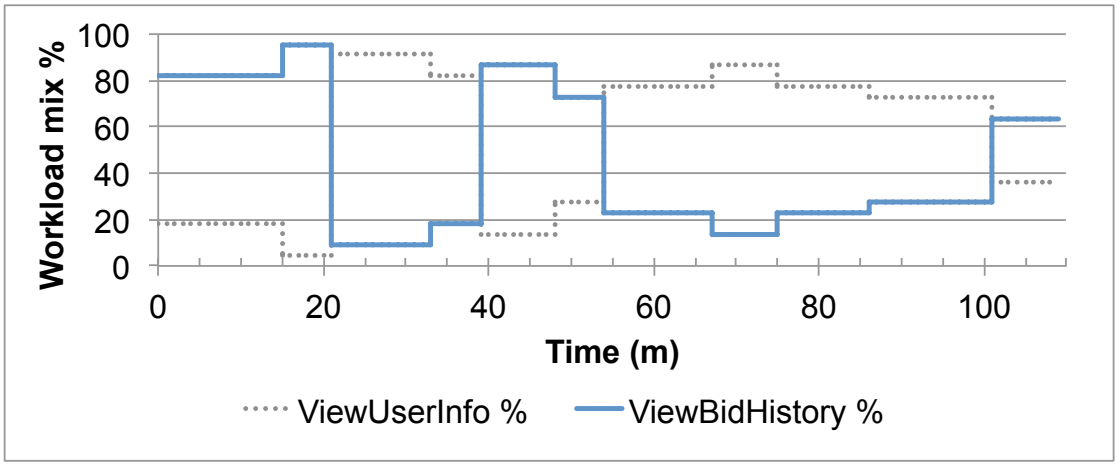

Figure 3.7. Requests mix for workload two.

Server Provisioning and SLA. Similarly to the previous experiment, the baseline approach shows a strong instability throughout the entire workload. Figure 3.8 describes the provisioning decisions for workload two. The baseline approach allocates two or more servers every time the system experiences a transitory phase. These phases typically happen right after a steep variation in the workload and last for 1 or 2 minutes. After

the transient phase is over, the baseline starts reducing the number of allocated servers, often incurring in SLA violations —see Figure 3.9. So the baseline approach confirms its tendency to under-allocate, and consequently experiences several SLA violations.

The solution from [98] exceeds the SLA limit twice at the beginning of the workload. This is due to the very high load that was selected by the random generator at the very beginning. Since we always begin with one allocated server, which obviously cannot satisfy the demand at the workload's first step, the two SLA violations are unavoidable and cannot be attributed to an erroneous provisioning. Indeed, the provisioning strategy completes the workload with no further violations, confirming its soundness. In terms of allocated servers, the solution from [98] appears to be more generous, requesting up to four servers for extended periods of time. Even though only three servers are actually allocated, even when four are requested, the 95% percentile of the response time remains steadily well below the limit. Again, an almost flat line for the response time throughout the workload shows that the servers are running at a very low utilization.

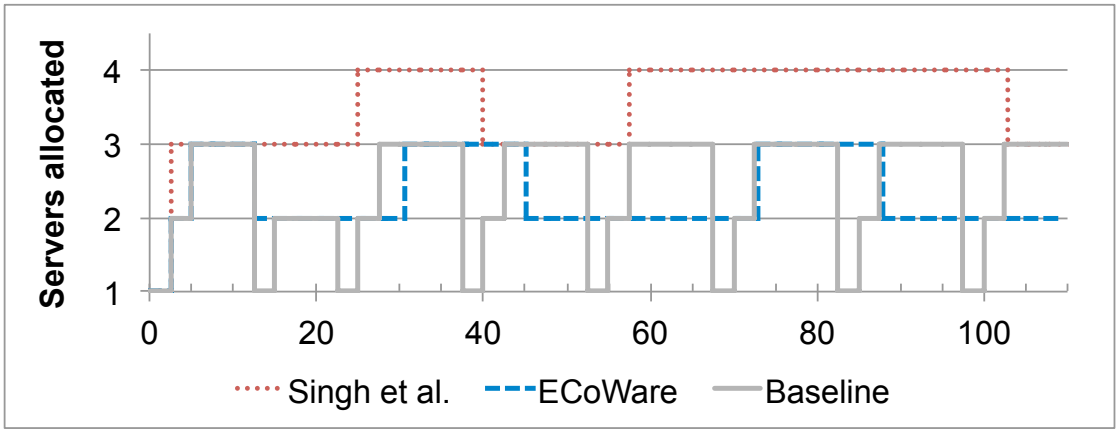

Figure 3.8. Number of servers allocated by the three solutions for workload two.

ECoWare experiences the same SLA violations at the beginning of the experiment as it does in the previous two approaches. Similarly to [98], it does not incur any further violation for the remainder of the workload. ECoWare correctly adapts its provisioning

decisions throughout the experiment, allocating up to three servers in three different moments. ECoWare's diagram in Figure 3.9 shows wide fluctuations in the response time, proving that the servers have a much higher utilization, thus causing a longer execution and waiting time for the requests. However, ECoWare is able to maintain those fluctuations under the SLA limit, thus improving the overall utilization of the allocated servers.

Result. For this experiment, the baseline approach scored 302.5 server-minutes, but incurred a significant amount of violations due to its tendency to under-allocate. The provisioning strategy from [98] used 385 server-minutes, with a perfect response time, with the exception of the two initial violations that could not be avoided. Finally, ECoWare allocated only 255 server-minutes, with no SLA violations, with the exception of the usual two at the beginning, which, again, were unavoidable. ECoWare also showed a higher fluctuation in the response time, which insinuates a higher utilization of the servers. Overall, ECoWare used 16% fewer server-minutes than the baseline approach with less violations and 33% less than [98] with the same amount of violations.

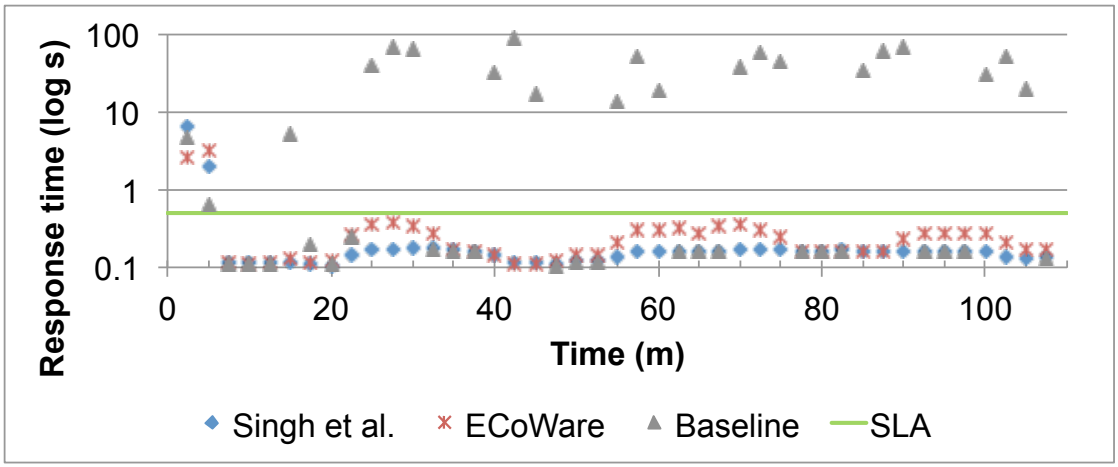

Figure 3.9. 95% Percentile response time for workload two.

3.5.3 Threats to Validity

A possible objection to our evaluation is that we only focus on two servlets. We do not consider this to be an over-simplification; on the contrary, many related works adopt a similar approach (e.g. [98] uses three servlets). Furthermore, modern applications often have many different kinds of request types, and it is often difficult to make a qualitative distinction between them.For this reason, it is quite common to use K-means clustering, as done for instance in [98], to group requests into fewer clusters with similar service demands. In our work, we have taken `ViewBidHistory` and `ViewUserInfo` as two representatives of two clusters: the cluster of memory-intensive requests and the cluster of CPU-intensive requests. Clustering also allows us to mitigate a limit of queuing networks, i.e., that they do not scale well when the number of classes of jobs significantly increases.

Table 3.2. Average solving time for the QN model for different numbers of classes.

# of classes	2	3	5	7	10
Solving time (s)	3	5	9	15	18

To evaluate the scalability of our approach, we benchmarked the QN solver with up to 10 classes (see Table 3.2); we chose 10 since we believe it be a realistic top limit for the number of clustered request types. The table shows that even with a higher number of classes (such as 10, in our case), the QN solver would still terminate within a reasonable time frame; hence, our approach could be applied without any modification. A different setup (different hardware or a different QN solver) could generate even better results. We also set a 20 second time limit to its execution, meaning that the solver is preempted after that period of time. This allows our model to have a bounded computational time, at the cost of a possible loss of accuracy. However, that does not impact our results in any noticeable way, since it takes the solver longer to terminate when the infrastructure

is close to saturation. When this happens, the response time will violate any given SLA, since it will grow unboundedly. Therefore, ECoWare's provisioning strategy saves time, without losing any accuracy, by preempting the solver, adding one server and re-running the solver with the new configuration.

3.6 Related Work

In this section, we focus on means for achieving dynamic resource provisioning in Web applications. Most existing approaches are mix un-aware; they only consider the volume of arriving requests when determining their provisioning. In their QN model, Almeida et al. [20] use the peak session arrival rate. Villela et al. [105] leverage QNs to model application servers and to solve an optimization problem for profit maximization; however, they only consider the aggregate number of requests generated by the clients. Kusic et al. [68] present an optimization framework for resource allocation, expressed as a sequential decision making problem under uncertainty, and solved using a limited look ahead control scheme. The approach distinguishes between different classes of clients with different SLA limits but does not consider the resource usage profile. He et al. [54] focus on dynamic allocation for interactive systems (e.g. web search engines), where the quality of the results depends also on the allocated time; in this case they do distinguish between interactive and batch jobs. In [112] Zhou et al. present a two-tier resource management framework that focuses on optimizing resource allocation while provisioning proportionality fairness to clients. They consider different classes of clients, associated with different SLA agreements. Ghanbari et al. [50] address capacity autoscaling in clouds as a stochastic, model predictive control problem where both performance and costs are taken into consideration. SPIRE [78] is an autonomous system for service provisioning driven by a utility function: optimize the average earned revenue over time, while satisfying the SLA. SPIRE also assumes a uniform service time for

all the requests. As additional formulation of the dynamic allocation problem, we also mention the decentralized control theory approach of [107] and the machine learning technique proposed in [69].

As we have seen in our study, different types of requests can have both very different service times and resource usage profiles. Our approach is mix aware. It takes into account the resource usage profile of the requests and its impact on the response time, given a hardware configuration. Among mix-aware approaches, we mention Sharma et al. [97]. They use a network of M/G/1-PS queues and an approximate model to compute response time distribution. Their solution takes hardware configuration into consideration. However, their approach differs from ours, as they focus on dealing with a multitude of hardware configurations in terms of service rates. Instead, our approach focuses on modeling the hardware to achieve a higher accuracy in the performance predictions. We plan to extend our approach to also consider different hardware configurations as well. Works in [94, 81] also focus on mitigating performance degradation caused by shared resource contention. Finally, Krebs et al. [67] provides metrics to quantify performance isolation in the context of shared resources.

Layered QNs [34, 92, 81, 82, 74, 62] and layered Petri Networks [88] have also been extensively used for modeling hardware-software systems with very promising results. We plan to extend our solution to these techniques in the future.

When performing dynamic resource management, much effort has been placed on optimizing energy consumptions. Although this is not the main focus of our work, we could adapt our request usage profiles to include energy footprints, creating a bridge between the application's QoS and its footprints. Our experiments already show promise, since they allow us to allocate fewer resources than what others achieve. Oliveira et al. [46] equip an application with a loop that determines the minimum amount of resources needed to provide the best possible quality of service, and a second loop that

manages the physical resources in the infrastructure. However, they do not currently synchronize these two loops. Ardagna et al. in [25] allocate resources in a multi-tier virtualized system with the goal of maximizing SLA revenue while minimizing energy costs.

3.7 Conclusions and Future Work

We demonstrated that an allocation strategy that takes into account the resource usage profiles of the different requests helps increase the accuracy of our performance predictions, which, in turn, allows us to improve the utilization of our infrastructure. In the future, we will continue to evaluate ECoWare in the context of cloud technology, and evaluate the advantages that we could derive from the use of layered queuing networks for our models.

3.8 Acknowledgments

We would like to thank Giuseppe Serazzi and Marco Gribaudo for the great discussions on queuing networks. A thank you goes also to William Griswold for the useful feedback.

This chapter, in part, is a reprint of the material as it may appear in Seracini, Filippo; Menarini, Massimiliano; Krüger, Ingolf; Baresi, Luciano; Guinea, Sam; Quattrocchi, Giovanni, “A Comprehensive Resource Management Solution for Web-based Systems”, in Proceedings of the 11th International Conference on Autonomic Computing (ICAC) 2014. The dissertation author was the primary investigator and author of this paper.

This work is based on an earlier work: Seracini, Filippo; Menarini, Massimiliano; Krüger, Ingolf; Baresi, Luciano; Guinea, Sam; Quattrocchi, Giovanni, “A Comprehensive

Resource Management Solution for Web-based Systems", in Proceedings of the 11th International Conference on Autonomic Computing (ICAC) 2014.

Chapter 4

Green Web Services: Improving Energy Efficiency in Data Centers via Workload Predictions

After improving over the state of the art for reactive systems with the highly accurate performance model and monitoring infrastructure presented in chapter 3, we now overcome the limitations of reactive systems themselves by adopting a proactive approach to resource allocation.

In this chapter, we introduce the remaining two pieces of our vision by presenting the SOPRA framework. SOPRA is able to proactively adapt the amount of resources allocated, thanks to an Execution model combined with a Statistical and a Performance model. We show how predicting workloads yields to significant energy savings and improvements in the allocation of resources for an Internet application, when compared to reactive solutions.

4.1 Introduction

Thanks to the widespread adoption of the cloud computing paradigm, more and more IT resources for people and enterprises have been externalized to public cloud providers. This has led to a significant growth of data centers worldwide, both in size and

number. Because of smaller form factors, the number of servers deployed in a data center has also dramatically increased, and so has both the total amount of energy required to power data centers and their carbon footprint. With the cost of electricity expected to rise up to 15% in the next five years [18] and already accounting for almost 50% of data centers' operational costs [65], in addition to the fact that carbon emission regulations are becoming stricter [101], the reduction of power consumption has become a real and urgent necessity.

Web services are an important type of workload deployed in data centers. The resources needed by web services are related to the volume of requests that they have to satisfy. This volume can experience a manifold increase without warning, an event called a flash crowd [20], creating significant performance problems that can lead to frustrated customers and a loss of online business. Although web services often follow historical patterns, such as high levels of demand during holiday seasons for e-commerce related web services, they still have multiple spikes and lows during a regular day. The quality of service is assured to the consumer by the service provider and is defined in a contract called Service Level Agreement (SLA). An SLA regulates multiple aspects of the quality of service for a web service, e.g. response time, and the penalties in case of violations. In order to accommodate for the spikes of demand and avoid the penalties associated with SLA violations, web service providers tend to over allocate computing resources. However, this leads to poor resource utilization - in a typical data center the average server utilization is around 30-40% [35] - and higher electricity bills. Capacity planning and optimization of computing resources are among the main areas that can reduce energy bills in data centers [20, 95]. Therefore, a system that autonomously allocates resources according to variations of demand could significantly reduce the electricity bill, hence the operational costs and carbon footprint.

Self-adaptive systems can either be reactive or proactive, depending on when

the adaptation is triggered. In the case of reactive systems, adaptation takes place after monitored events have occurred. For the purpose of minimizing energy consumption in data centers, reactive systems have drawbacks. One drawback is that reconfiguration can start only after the monitored event triggers it. For example, if we monitor the response time of web service calls, a sudden slowdown will cause a reconfiguration. However, in this situation either the SLA is already violated, or the resources have been over allocated for some time and energy has already been wasted. A second drawback is that the execution of the adaptation could impact the execution time of the running application, worsening the violation that triggered the reconfiguration. A third drawback is that, if the adaptation time is long (e.g. turning machines ON/OFF), the violation of the SLA could protract for a relatively long time, limiting the applicability of these reconfiguration strategies in conjunction with reactive adaptation. Proactive systems do not have these drawbacks, since the adaptation takes place before reaching an execution point where a problem may occur [28]. On the other hand, a wrong prediction, that underestimates the computation needs of a web service application, will produce severe violations of the SLA.

The proactive approach presented in this paper reduces the amount of energy consumed by a multi-tier web service system by switching off unused computing resources. Our framework SOPRA – *S*ervice *O*riented *PR*oactive self-*A*daptive framework, leverages information from the application layer, in this case web services, to predict the future workload. SOPRA uses the predictions to proactively allocate resources right before they are needed, i.e. spikes of requests, and to de-allocate them when they are not, i.e. when a low volume of requests is expected. SOPRA satisfies the business needs of the service provider by fulfilling the SLA even with fluctuating demand. Furthermore, by using resources only when needed, SOPRA decreases the total amount of energy used. This reduces CO2 emissions and electricity bills.

SOPRA forecasts the capacity needed by predicting the number of requests that will be received. To this end, it uses information on the web application workflows. Each workflow captures the possible sequence of web service operations that are called to perform certain tasks. To obtain a reliable prediction, SOPRA combines workflows with statistical data on the observed call patterns. The effectiveness and accuracy of these predictions depend on the kind of workflow and usage patterns. In this paper, we use a fulfillment center web service for an e-commerce web site as use case to demonstrate how our predictive proactive approach can contribute to save energy in the case of a service-oriented architecture deployed in a data center. In our experiments, we obtained a substantial reduction in energy consumption. Compared to an over allocation strategy, that allocates resources for the worst case scenario, our approach saved 52.49%. Compared to an optimal static prediction strategy, that allocates a fixed number of resources over our experiment interval, we still saved 13.95%.

We also experimented with the effects that different SLA have on achievable energy savings. SLA can greatly affect the number of resources that must be allocated to support a given workload. In our experiments, using a more demanding SLA led to a substantial reduction in the energy saving achieved by SOPRA, which went from 52.49% to 28.29%. Finally, we investigated how prediction errors affect the performance of our technique. In particular, we identified tradeoffs between saving energy, tolerating timing errors in the prediction window, and constraints imposed by the SLA. The main contributions of this paper are i) a proactive approach, based on workload predictions, that improves the allocation of resources to web services and reduces the energy consumption of data centers and ii) the identification of workload properties that can improve the accuracy of the predictions, and iii) SLA properties supporting adaptation strategies that can lead to larger energy savings.

This work is organized as follows: Section II introduces the case study that will

be used for the experiments. Section III gives an overview of the SOPRA framework. In Section IV, we describe the experiments, while in Section V we discuss the results and the properties that workflows and SLAs should have to achieve better results. Section VI presents previous work in this area. Finally, Section VII summarizes the conclusions and future work.

4.2 Case Study

For this work, we took a simple workflow composed of two web service calls, *S1* and *S2*. For our experiments, we created a fulfillment center (FC) web service based on the Amazon fulfillment center API [4] that exposes two operations to e-commerce web sites: *getPreview* and *createOrder*. They correspond to *S1* and *S2* respectively. *S1/getPreview* is invoked when the e-commerce web site gathers information about an item. This request is sent to the FC every time a customer clicks on an item on the e-commerce web site. *S2/createOrder* is invoked to create a fulfillment order. This request is sent every time the customer proceeds with a purchase on the web site. For the FC, the create order request entails much heavier processing, since it has to check for availability, update the inventory, calculate the packaging, the shipping costs, and finally, schedule the pick-up with the delivery company.

4.2.1 Leveraging the Application Workflow

By observing this simple common life example, represented by a customer purchasing an item on a web site, and combining it with user statistical data, extracted from the history log, we can extract some important pieces of information, summarized in Figure 4.1.

1. Before S2 is invoked, one or more invocations of S1 take place.

2. On average, there is a delay between a *S1* and a *S2*. For this work, we assume the usage statistical data report a value of five minutes $+/-30s$.

3. The number of invocations of *S2* is strictly related with that of *S1*. For this work, we assume that the usage statistical data return a ratio of 40%.

These three simple pieces of information can be determined by looking at the historical data for that workflow. They enable SOPRA to make predictions on the volume of future demand and determine the time window needed to adapt the allocation of resources. In fact, if SOPRA observes a spike in the volume of *S1* requests, after about five minutes it will probably receive about the 40% of that volume that *S2* requests. Hence, if the resources currently allocated are not sufficient to satisfy that demand, SOPRA has five minutes to allocate more by, for instance, turning ON more servers.

It is important to mention that our approach can be applied to any workflow where statistical data on the correlation between steps are available. For this work, we used a simplified implementation of a fulfillment center; the workflow represented the series of web service calls an online customer generates during a purchase on an e-commerce web site. However, the approach is applicable to any other workflow with similar characteristics.

4.2.2 The Service Level Agreement - SLA

From this discussion, we can already observe that the accuracy of the statistical data and the adaptation window are key elements needed to make the predictions effective. Furthermore, different lengths of adaptation windows could enable different adaptation strategies. For instance, strategies could include turning servers ON/OFF, stand-by, hibernation, HW low power states, changes in virtual machine consolidation; all these strategies are characterized by different energy saving effectiveness and adaptation time

Table 4.1. SLAs for the case study for web service S2

Name	Average Response Time over 5s
SLA1	3200 ms
SLA2	2500 ms

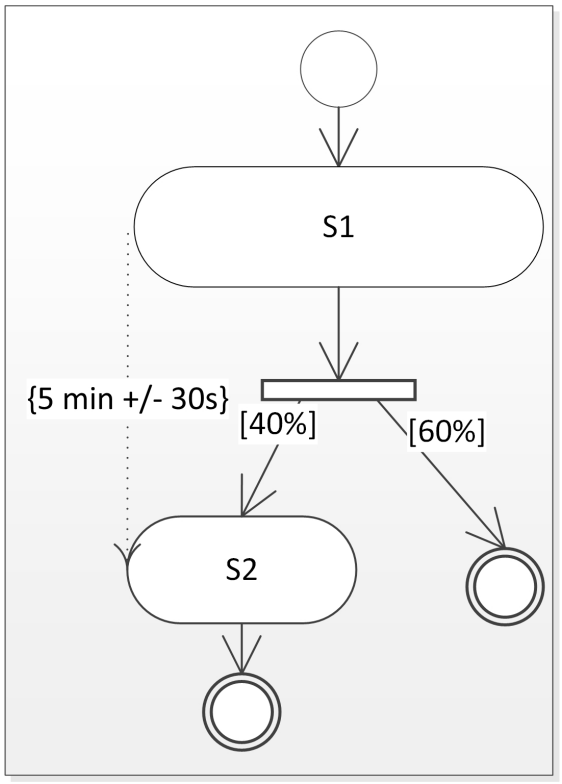

Figure 4.1. Case study workflow

–see Table 4.2. As a result, a flexible SLA can have a big impact on the kind of adaptations that can be put in place; hence, it can directly influence the amount of energy that can be saved. For our experiments, we assumed having two SLAs, as shown in Table 4.1.

SLA1 is more flexible and allows higher average response time over 5 seconds. *SLA2* instead is significantly more stringent. The implications of the flexibility of the SLA and the accuracy of the predictions will be discussed in Section 4.6.

4.3 The Sopra Framework Overview

In this section we give an overview of the SOPRA framework. Our approach to proactive adaptation follows these steps: i) monitor incoming requests to determine the current amount of workload, ii) use these data in conjunction with statistical usage data to predict what the future workload will be, iii) combine the prediction with resource usage data to determine the amount of resources needed, iv) update the system configuration (i.e. resources allocated) and the web service composition. The SOPRA proactive framework continuously loops through these steps and is composed of six components: i) *System Monitor*, ii) *Predictor*, iii) *System Configurator*, iv) *Performance Model*, v) *Execution Model* and vi) *Load Balancer*.

The *System Monitor* is responsible for monitoring and recording the number of incoming requests for each public web service operation (i.e. *S1* and *S2*). The *Predictor* takes the monitored data from System Monitor and combines them with the statistical usage data to predict the future workload. In our example, the Predictor calculates how many *S2* requests the system should expect in five minutes. The *Execution Model* is a mapping from a public web service operation to the set of private operations that compose it. The *Performance Model* maps each web service operation to the number of requests that the server can handle for that operation in a fixed amount of time without violating any aspect of the SLA. The values of the performance model are the result of stress-tests we ran on our servers to assess the amount of resources consumed by a single invocation of each web service operation, both public and private, where the latter are the web services exposed within the FC for its internal functioning. To simplify the discussion, in this paper, the resource usage contribution of private operations will be factored in that of the public operations. Our approach could also be applied to private operations as well. The *System Configurator* combines the workload prediction and the Performance

and Execution Models to determine the amount of resources that will be needed at time $t + t_{prediction}$, where $t_{prediction}$ indicates the time frame of the prediction window.

Every $t_{adaptation}$, the System Configurator retrieves the configuration computed for that time point and updates the infrastructure with the appropriate amount of resources. The value of $t_{adaptation}$ determines the frequency that the system adapts its configuration. The higher the frequency (i.e. smaller $t_{adaptation}$), the more precisely can SOPRA adapt to fluctuating volume of demand. Fig. 2 presents the algorithm that computes the configuration that the system will adopt at time $t + t_{adaptation}$. Step 4 is where the configuration is actually computed; it sums up all the contributions in terms of resource utilization for each private operation composing the public web service operation. Since we measured the resource utilization of each private operation, we can determine the total amount of physical machines that the system will need to satisfy the volume of demand predicted.

Input: workload prediction (*pred*), execution model (*exMod*), performance mode (*perfMod*)
Output: Map of required number of servers for each public web service operation (*pubWsOps*)

```
1 reqServers ← ∅ ;
2 foreach pubOp_j ∈ pubWsOps do
3 |   foreach privOp_i ∈ exMod(pubOp_j) do
4 |   |   reqServers(pubOp_j) ← reqServers(pubOp_j) + (1 /
  |   |   perfMod(privOp_i) * pred(privOp_i, pubOp_j) ;
5 |   end
6 end
```

Algorithm 2: Compute Configuration algorithm

4.4 The Experiments

4.4.1 Environment Setup

To assess the energy saving capability of our approach, we created a running prototype. The prototype uses information of the correlation between web services and the statistical data we described earlier to manage the resources allocated to our implementation of the FC, which we implemented as SOAP web services deployed over Mule Enterprise Service Bus (ESB) 3.3.1 Community Edition [13]. The setup consists of eight physical machines, each equipped with two Intel(R) Xeon(R) CPU (2.66 GHz, 4 cores) and 32 GB memory, running 64-bit Microsoft Windows Server 2008 R2. In order to avoid cross effects between adaption strategies, in the BIOS we turned off the DFVS of the processors and the thermal adaptation of the cooling fans. The FC is implemented as a three tier architecture, with a physical machine allocated to the presentation tier and one to the database; the number of machines allocated to the processing tier varies with the workload. Since only the processing tier has a variable need for resources over time —the other web services would not be able to saturate their hosting machines, in our experiment, SOPRA optimizes only the number of machines assigned to that tier.

The customer workload is generated on a separated machine by using Apache JMeter [6]. Finally, the SOPRA framework is deployed on the same machine as the presentation tier, since the latter was not able to fully utilize the server; we did not experience any resource contention issue or significant performance degradation. In our setup, we set the adaptation interval $t_{adaptation}$ to 30s and $t_{prediction}$ to 300s. Note that, due to differences in hardware configuration between the servers (mostly motherboard, types of hard drives and number of cooling fans), they present different energy profiles. SOPRA takes this into consideration and assigns the servers in increasing baseline power consumption order. For the power measurements, we used an Avocent PM 2000, a

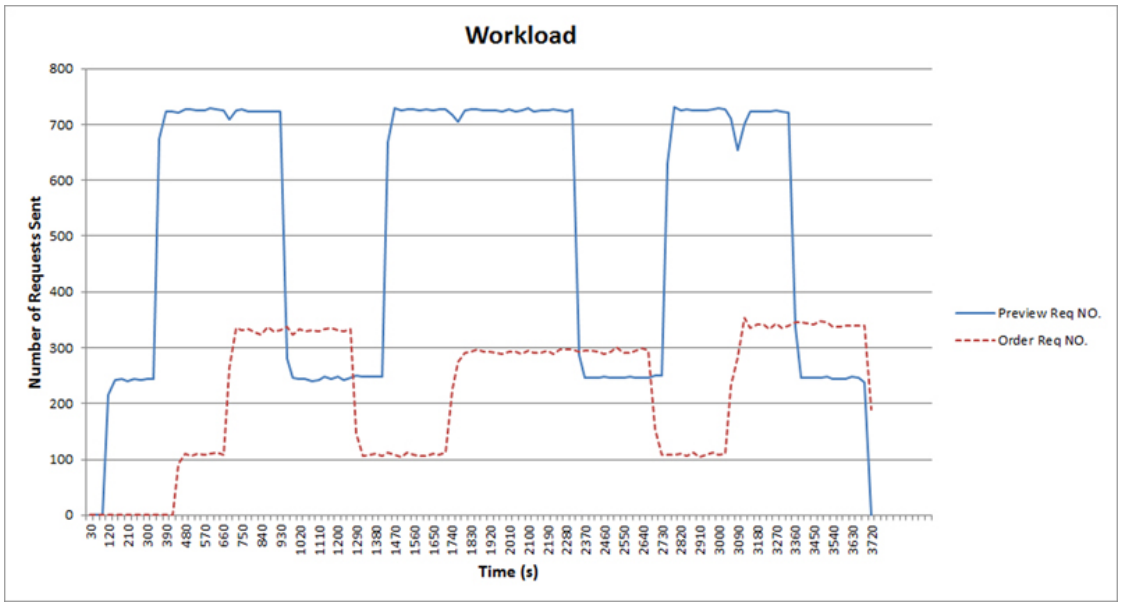

Figure 4.2. Workload used for the experiments

networked PDU with metering capability at each outlet.

4.4.2 Workload

As mentioned earlier, the contribution of this work is to improve workload predictions by statistically correlating different web service operations and allocating resources based on those predictions. In our case study, our solution allows the infrastructure to optimize the resources for the create order operation, based on the volume and timing of the preview requests. However, we are not improving on the predictions for the preview requests. In the literature there are many examples of usage history analysis also applied to capacity planning and workload predictions in data centers (e.g. [104, 58, 37]); these approaches could be applied to improve the predictions for preview requests. Table 4.3 shows the statistical assumptions we used for our model. We then created one, hour long workflow with three different spikes that follow the statistical data; in other words, by running a statistical analysis on the workload, we would get similar statistical data. The workload profile is depicted in Fig. 4.2.

Table 4.2. Possible adaptation strategies

Energy Saving Technique	Saving	Time Granularity
Turning servers on and off	Very High	Minutes
Sleep mode	High	Seconds
CPU Power throttling	Low	Tenths of seconds

Table 4.3. Statistical Values of the Workflow

Property	Average Value	Variation
Delay between preview and create order requests	$300s$	$+/-30s$
Ratio between volume of preview and create order requests	40%	4.5%

The second step of our approach was to decide what adaptation strategy to use. Many adaptations exist that feature different energy saving profiles and time granularity. In Table 4.2 we list few of them. In general, the larger the time granularity is, the higher is the potential energy saving.

Each machine takes up to two minutes and 20 seconds to boot up and up to 20 seconds to turn off; hence, a workflow that shows an interval between the two steps shorter than the power cycling time of the machines could not benefit of this adaptation strategy. For this study, the expected interval between *S1* and *S2* is $300s$. While this is a limitation of the adaptation strategy itself, it does not affect the validity of our approach, as other adaptation strategies with faster reaction time could be used instead. For workflows with shorter prediction windows, we could use the same prediction approach but leverage a quicker adaptation technique.

We envision that all adaptation strategies could be used jointly to maximize the energy savings in case of complex workflows with prediction windows of different length. A system that selects the adaptation strategy based on the prediction window of that particular workflow is left for future work.

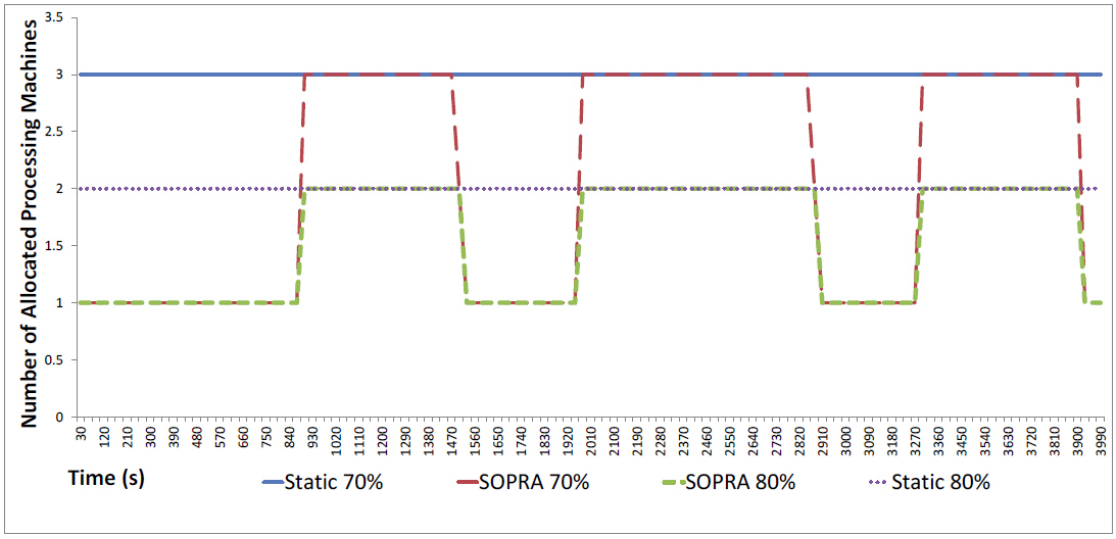

Figure 4.3. Number of allocated machines to the Processing tier

4.5 Results and Discussion

For our experiments, we executed the workload with 5 different settings: 1) over provisioning, 2) static prediction with 70% load, 3) static prediction with 80% load, 4) SOPRA with 70% load, and 5) SOPRA with 80% load. The first setting is often used in data centers. It can be very inefficient from the resource and energy point of view if the average workload is low and presents large flash crowds. Because of the flash crowds, data center management is required to leave enough resources to handle spikes, resulting in an average utilization level around $30-40\%$ [35]. Settings 2) and 4) set the maximum utilization level for the machines at 70%. In our experiment, it is the value required in order to satisfy the response time defined by *SLA2*.

When the incoming workload is expected to load a machine over that threshold, an additional server is added to the pool of processing tier servers. Finally, in 3) and 5), we use the 80% threshold, because it is the utilization target suggested by the industry [9]. In our experiments, 80% is required to provide the average response time of *SLA1*. We compared our work with a static prediction, because that is a common approach in the

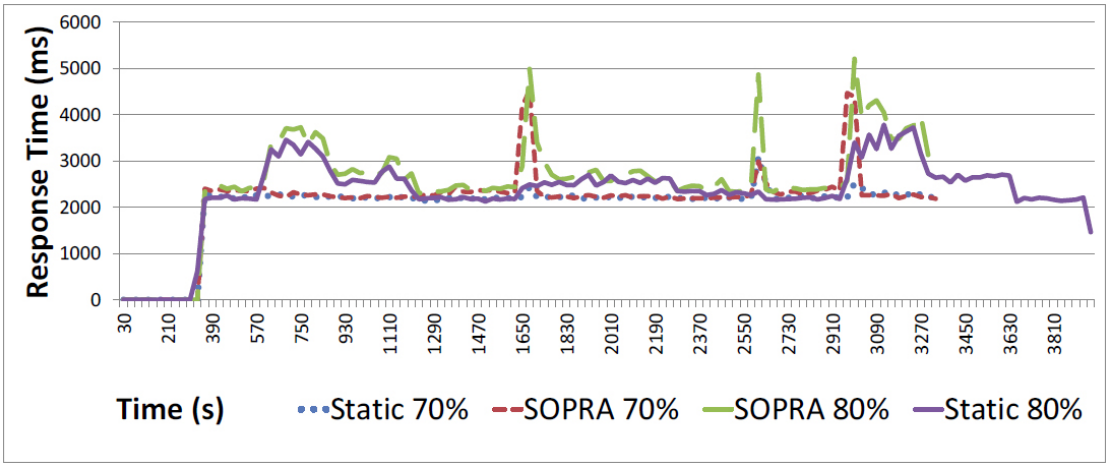

Figure 4.4. Five seconds average response time for web service S2

literature (e.g. [58, 37]). We did not compare with a reactive solution, because of the limitations identified in Section 4.1.

Fig. 4.4 shows the amount of servers allocated to the processing tier over time for the five settings. The most interesting part is the difference between the 70% and 80% line: the latter setting never allocates the third machine. This, however, comes with the cost of a smaller number of requests being addressed for the 80% setting during the spikes; this is clearly visible in third spike in Fig. 4.5. The reason for this is in the constant number of clients that is being generated by JMeter. Since the processing servers are taking longer to answer, a smaller number of requests per unit of time are sent by the clients. In fact, the clients have to wait for a longer time before receiving an answer, thus sending another request. The same behavior can be observed in the response time, which is also affected by the amount of resources allocated. The 80% configuration clearly underperforms the other two settings, especially when the prediction is less precise. We can observe in Fig. 4.3 that, since the second spike of create order requests (see Fig. 4.2) arrives about 30*s* earlier than expected, the system is saturated because the additional machines have not been allocated yet. This is the reason for that spike for both the 70% and 80% settings. However, since the 70% has more capacity left, it can buffer more

Figure 4.5. Number of S2 requests handled

requests before saturating; hence the response time does not present a peak as high as that of 80%. The latter also has higher average response time during the spikes, which is expected since it uses only two machines. In both the second and third spike, the workload does not follows the prediction as create order requests keep arriving, even after the 5 minutes window. Since SOPRA is purely based on predictions, five minutes after the end of the spike of preview requests, it starts de-allocating resources; hence, we observe an increase in response time at the end of both the second and third spike.

Finally, in Fig. 4.6, we can see the effectiveness of the three different strategies when it comes to energy saving. Table 4.4 shows the energy measured for the both the SLAs under the five different approaches. The baseline is the over allocation approach, where we considered a constant number of six servers allocated the whole time. The static prediction would be the result of a perfect predictor; it allocates two servers for SLA1 and three servers for SLA2. Table 4.5, instead, shows the energy savings w.r.t. the over allocation approach. As expected, SOPRA, with the 80% configuration, saves the largest amount of energy, since it allocates at most two machines.

The SOPRAs energy savings are 52.49% and 28.29% for *SLA1* and *SLA2*, respectively. This shows how different SLAs can have very different effects on the system

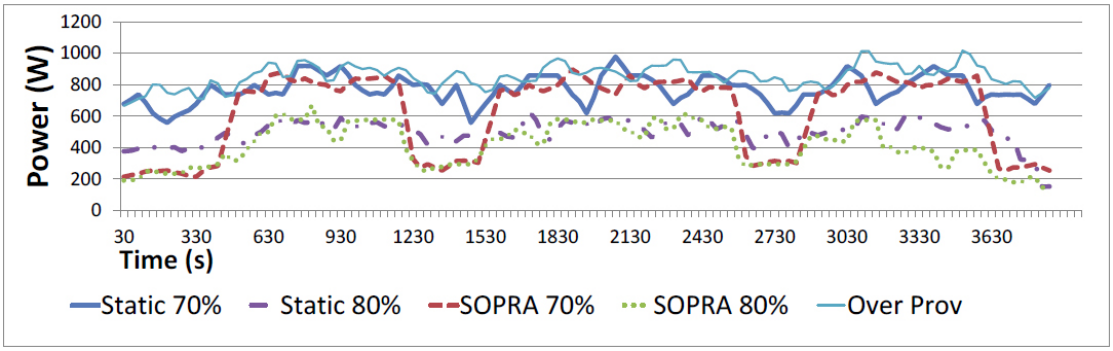

Figure 4.6. Power Profile

Table 4.4. Measurements of Energy Consumption

SLA	**No Prediction Over Provisioning**	**Static Prediction**	**SOPRA**
SLA1 $(3200ms/5s)$	$0.947kWh$	$0.523kWh$	$0.450kWh$
SLA2 $(2500ms/5s)$	$0.947kWh$	$0.758kWh$	$0.679kWh$

configuration. In fact, a more flexible SLA, that tolerates higher response time, enables more aggressive configurations that allow more resources, thus energy, to be saved. Instead, a very strict SLA that cannot tolerate any violation or only very small delays in the response time will likely preclude those kinds of aggressive saving strategies and require a higher number of physical machines to always be allocated.

The accuracy of the statistical data also has a direct impact on both performance and energy savings. If a prediction has a low confidence level, because the correlation between the steps composing the workflow is weak, then the performance of the system decreases; the system could either adapt too late, resulting in higher response time and lower request served rates, or too early, resulting in wasted resources and energy.

Again, a stricter SLA would force the system to be more conservative and anticipate the allocation of resources. Instead, a flexible SLA could tolerate the inaccuracy of the predictions, fostering energy and resource savings. Supporting the precision of

Table 4.5. Energy Savings

SLA	No Prediction Over Provisioning	Static Prediction	SOPRA
SLA1 ($3200ms/5s$)	0.00%	44.78%	52.49%
SLA2 ($2500ms/5s$)	0.00%	19.95%	28.29%

the prediction is an aspect that could be taken into consideration by developers and data center management. A workflow that has a high confidence level could be provided at a better price than one that exposes very little predictability.

As shown in Table 4.2, there are many possible adaptation strategies. As proof of concept and because of HW limitations of our machines, for this work we decided to only turn servers ON/OFF. Even though this strategy provides great saving results, it could be further improved by combining it with other strategies. The ideal situation would be to have energy consumption proportional to the load. With current HW technology, this is not possible. In fact, when idle, our servers consume 150-200W and 300-350W when fully loaded. Thus, because of the high energy consumption of servers when idle, the ideal situation would be to have few servers running fully loaded and the others off. When turning servers ON/OFF is not an option, the other strategies could be applied. Using a combination of adaption strategy could help to reduce energy consumption. For instance, adapting the clock frequency of the processors would help our energy profile to more closely follow the workload curve. Given that the energy consumption does not have a linear relationship with the load, the analysis of the tradeoffs between the different power profiles is more complex. We leave an analysis of the performance of our prediction using different adaption strategies as future work.

4.6 Related Work

The problem of data centers' resource optimization is well known in literature and many approaches have been presented thus far. Our solution can be seen as a particular application of proactive adaptation of service composition combined with resource allocation policies that aim at reducing power consumption within a data center. The authors in [37] use machine learning techniques to create predictions to adaptively allocate resources in a data center in order to save energy and fulfill the SLA. With their work, we share the concept of the performance model, where we map workload levels to amount of resource needed. The main difference is that our solution generates predictions continuously in real time, while their approach is based on historical patterns and performed in scheduling rounds.

The work in [58] creates a resource model of multi-tier web applications through offline analysis of the logs. This model is then used in a machine learning analysis to predict the amount of resources that will be needed to satisfy the SLA for a given workload. With this work, we share the concept of performance model and extracting a probabilistic model out of the web services, but our approach performs adaptation at run time instead of offline capacity planning. The work in [64] also puts servers to sleep when the incoming network traffic is low; servers are turned ON only when the length of a request queue goes above a threshold. The approach in [64] leverages SLAs that do not have a fixed deadline, but instead suffer a penalty proportional to the response time. Our solution deals with stricter SLAs that do have fixed deadlines; hence, SOPRA cannot suspend the requests while waiting for the request queue to go above the threshold. The authors in [57] present a power saving solution that combines dynamic voltage frequency scaling with low power states in web server farms. Their approach achieves a max 17% in reduction of power consumption compared with a non-adaptive strategy; however,

the QoS is slightly decreased because of the time servers take to wake up from the low power state. Our solution solves that limitation thanks to the use of predictions that allow SOPRA to turn servers ON right before the additional capacity is needed. Also, compared with a not-adaptive approach, our solution shows higher energy savings.

The field of self-adaptive systems is also populated with wide variety of approaches; we will only give a short survey here. In [28] the authors present a framework for proactive adaptation of service composition based on changes in service operation time. With this work, our approach shares the concept of the creating a model of the execution to predict the future behavior of the system. However, since we extract information from the workflows, our prediction window is larger. Moreover, our optimization focus is different, because not only do we try to avoid SLA violations, but we also enforce an aggressive resource management to save energy. PAWS [23] is a very flexible adaptation framework for web service composition. The main difference is that PAWS is a reactive system, hence some adaptation strategies are harder to apply; energy is also not taken into account. The approach presented in [60] formulates service composition as a constrained model, where each constraint represents a QoS aspect. In the paper, the authors introduce the concept of cumulative QoS metric, and energy could be one of that. Thus, it would be interesting to apply this constraint model to our framework in order to achieve further optimizations. For example, one could attempt to select different implementations of of the same service depending on some QoS parameter or class of service.

The fact that useful information regarding the application layer is not visible at lower levels of the SW/HW stack, where energy optimization typically takes place, is also advocated in [46]. In this paper, the authors propose a mathematical model to quantify the energy consumption of the hardware resource associated with the execution of a service; they use this model to make informed decision on the structure of the workflow. They also define Green Performance Indicators as metrics to assess the energy

impact of an application. However, this solution is limited to a reactive system. In the solution described in [90], the authors dynamically adapt the QoS of an Internet Java EE server by upgrading or downgrading the service quality, in their case a video streaming service. This concept could be applied to our scenario by having different classes of servers or different amount of resources allocated to the same service, depending on the QoS defined in the SLAs; currently SOPRA only supports instantiating more servers with the same code base. VGreen [47] presents a dynamic runtime resource utilization profiling for virtual machines like Xen. The main focus of this work is to save energy by speeding up computation on machines hosting multiple VMs thanks to the reduction of hardware resource contention within the same physical machine. This approach could be an extension to SOPRA when trying to optimize different workloads concurrently.

Turning machines ON and OFF has been dividing researchers and data center managers for quite some time. In fact, the authors of [44] claim that repeated ON/OFF cycles can increase the wear-and-tear of server components. Besides, data center managers have also to take into account that a server might fail during a power-cycle, hence increasing the operational costs. On the contrary, in [89] and [43] the authors advocate that power-cycling is an effective way to reduce consumption, without mentioning any negative effect on the reliability. This is an aspect that requires more studies and evaluation, in particular related with the business model of the data center. This evaluation is outside of the scope of this paper. In our case, due to hardware limitations of our servers, we decided to turn off the machines. However, our approach could be easily extended to leverage additional adaptation strategies. Furthermore, the adaptation strategy is a function of the prediction time window; the further ahead the system can predict, the more choices of adaptation strategies the system can have.

Finally, instead of reducing the total amount of energy, a data center management could aim at increase the amount of revenues for a given amount of energy. This could

be achieved by allocating unused resources to low priority jobs during the prediction windows. We leave this approach for future work.

4.7 Conclusions

In this paper, we presented an approach to optimize resource utilization in a data center in order to reduce the amount of energy consumed. We presented SOPRA, a proactive self-adaptive framework that leverages correlations between web services, combined with statistical information extracted from the application layer, to make predictions on the incoming volume of requests. Compared with an over allocation strategy, our system can save up to 52.49% of energy. We also showed that the SLA has a direct impact on the adaptation strategies that can be applied. A stricter SLA forces the system to allocate more resource, hence consuming more energy.

The only adaptation strategy used in this paper is to turn servers ON and OFF. The application of SOPRA on a real context, in order to assess the accuracy of the predictability, is ongoing work. Future work is to support more adaptation strategies so that the system can dynamically choose which is most efficient, given the potential saving of energy, the SLA restrictions, and the type of workload. We also want to improve the amount of revenue for a given amount of energy consumed. This can be achieved by smart scheduling low priority workloads during the prediction time window when the system is not fully loaded. Finally, we leave for future work the optimization of resources when more than one workload is currently being executed.

4.8 Acknowledgments

This work in this chapter was partly funded by NSF Project GreenLight grant 0821155 and Cisco Systems, Inc.

This chapter, in full, is a reprint of the material as it appears in Menarini, Massimiliano; Seracini, Filippo; Zhang, Xiang; Rosing, Tajana; Krüger, Ingolf, "Green Web Services Improving Energy Efficiency in Data Centers via Workload Predictions", in Proceedings of the 2nd International Workshop on Green and Sustainable Software (GREENS), IEEE, May 2013,. The dissertation author was the primary investigator and author of this paper.

Chapter 5

A Proactive Customer-Aware Resource Allocation Approach for Data Centers

In the previous chapter, we introduced the SOPRA framework and we showed the significant benefits that a proactive approach has when compared to both reactive approaches and static allocation solutions.

In this chapter, we extend the SOPRA framework by making it customer-aware, that is, to make it able to distinguish different customer usage profiles and leverage them in order to increase the accuracy of the workload predictions. By doing so, workload predictions become more accurate, yielding to further improvements in resource allocation.

5.1 Introduction

A common problem in data center management is resource allocation and provisioning in the presence of load spikes that occur frequently with Internet applications. Resource over-provisioning leads to low average server utilization [35] and high recurring utility costs. In this paper, we introduce a framework to dynamically allocate resources which is capable of up to an 84% reduction in the quantity of physical servers allocated, as compared with an over-allocation approach. Moreover, this is accomplished without

violating any Service Level Agreement (SLA).

An SLA is a contract between the service provider and the service consumer that regulates the quality of service, such as response time, and the penalties in case of violations. If not properly handled, a spike in demand can quickly saturate the computing resources of a provider. This can lead to increased costs for the service provider due to the penalties associated with SLA violations; it can also lead to issues for the service consumer if the service becomes no longer available or its performance is degraded. Most providers adopt a strategy of resource over-allocation to accommodate load spikes. Considering the rising cost of electricity, which already accounts for almost 50% of the operational costs of data centers [65], it is readily apparent that over-allocation of resources is becoming unsustainable. According to Sharma et al. [96], the optimization of computing resources is one of the main areas that can help reduce operational costs in data centers. Therefore, a *self-adaptive* system, that can adjust the amount of resources allocated to accommodate the variations of demand, could significantly reduce operational costs.

Self-adaptive systems can either be *reactive* or *proactive*, depending on when the adaptation is triggered. In the case of reactive systems, adaptation takes place after monitored events have occurred. Such systems have significant drawbacks. First, the monitoring event can be thrown some time after the problem that ultimately triggered the event has happened. This could create situations in which an adaptation is no longer possible or cannot solve the problem anymore, e.g. the SLA has already been violated, and the provider has already been charged for the penalties. Second, the execution of adaptation activities could have an impact on the execution time of the running application, or it might simply take some time to complete, thus limiting its applicability. Proactive systems, on the other hand, do not have these drawbacks because, as defined in Aschoff et al. [28], *proactive adaptation of service composition* is the “detection of

the need for changes and implementation of changes in a composition, before reaching an execution point in the composition where a problem may occur". In our approach, we extend this concept a step further and apply it not only to service composition, but also to infrastructure configuration. We proactively determine the need for changes in a composition, in our case a Service Oriented Architecture (SOA) deployed over a single data center, and we dynamically change both the composition and the infrastructure configuration before a problem occurs or the resources are poorly utilized.

To achieve that, we start from the observation that a lot of information about the execution is available at the application layer where web services are executed. Part of this information is inevitably lost as we go down the software/hardware stack as each layer abstracts away information. In our previous work, we introduced *SOPRA - Service Oriented PRoactive self-Adaptive framework* [80]. SOPRA leveraged the information at the application layer to make predictions of the incoming workload and on how it will affect the performance of the infrastructure; that information was aggregated across all the customers. However, different customers can have very different usage patterns, which can lead to very different loads on the infrastructure running the Internet application. In this work, we go beyond our previous work and extend SOPRA to become *customer-aware*. SOPRA can now distinguish among different customers of an Internet application to make tailored workload predictions by leveraging their usage patterns. These predictions are then used to adapt the allocation of resources in a proactive fashion, so that the infrastructure is ready for the incoming workload, and the number of SLA violations is reduced. We also show how, by extending the interfaces of the web services constituting the Internet application, information on customer usage patterns can be shared between the clients and the service provider in order to enable customer-aware workload predictions. From the point of view of the data center management, our solution reduces operational costs by i) reducing SLA violations (with the associated penalties)

and ii) improving resource allocation. To evaluate our approach, we implemented a prototype of an Internet application based on the Amazon Fulfillment Center API [4].

To assess the reduction of SLA violations and the improvement in resource allocation, we compared our framework with two reactive systems and a system without adaptation. In this paper, we use the term framework for the SOPRA implementation and the term infrastructure for the resources allocated to the Internet application. Finally, we use the term resources to refer to the physical servers of the data center. The main contributions of this paper are i) a *customer-aware* resource management infrastructure that leverages information from the application layer to predict future levels of workloads and proactively allocate resources. SLA violations are reduced and resource allocation is improved. ii) An extension to service interfaces to carry along the customer type information that is used by the framework to make customer-aware workload predictions.

The remainder of this paper is structured as follows. In Section 5.2,, we discuss how our approach compares to related works. In Section 5.3, we describe the use case for this work. In Section 5.4, we show how SOPRA predicts incoming workloads and we describe how service interfaces can be extended to expose useful information from the application layer. In Section 5.5, we present an overview of the SOPRA framework. In Section 5.6, we evaluate the framework. Finally, in Section 5.7 we discuss our final remarks and future work.

5.2 Related Work

Our solution can be seen as a particular application of proactive adaptation of service composition, combined with resource allocation policies that aims at reducing SLA violations and improving resource utilization. In [28], the authors describe ProAdapt, a framework for proactive adaptation of service composition based on changes in service operation time. We share the concept of creating a model of the execution to predict the

future behavior of the system. However, SOPRA predicts future workloads and can adapt before a variation in service operation response time is observed. On the other hand, SOPRA does not cover all the adaptation triggers of ProAdapt.

In [45], the authors present a proactive approach that uses predictions to trigger self-healing of the infrastructure. These predictions focus on network data transmission, while SOPRA predicts the volume of requests, thanks to additional information from the application layer. Therefore, the two approaches complement each other.

Leitner et al. present PREvent [72], a framework that uses machine learning techniques to make predictions and trigger adaptation in case SLA aspects are not going to be met. An important difference with our approach is the time frame for the prediction and adaptation which, in the case of PREvent, is significantly larger.

The authors in [91] propose predicting resource availabilities by leveraging past data, using predefined predictors such as linear recent history predictor. An important difference is that SOPRA leverages statistical data of the relationships among workflows, while [91] focuses on resource usages in the past time frame.

The authors of [77] present a rule-based approach that dynamically reconfigures the resources allocated to virtual machines (VMs). Their solution autonomically sets and adapts the threat thresholds of the rule-based system, in order to accommodate for variability in the workloads. Moreover, their system does not require any previous knowledge of the workloads. SOPRA differs from their approach as it proactively adapts before violations occur by predicting future levels of workloads. Instead, the solution in [77] reallocates resources when the threat thresholds are violated. The granularity of the adaptation is also different, as we allocate physical servers instead of reconfiguring VMs. However, our system requires knowledge of the workloads in order to create the performance model.

Both the PROSA [55] and the SHIWS [102] use online testing to validate the

current service composition and trigger adaptation if violations are detected. Tests are run when external events happen, such as a change of service provider or service context. These tests mainly verify the compatibility of the newly attached service with the rest of the composition. The online testing could be an interesting extension to our framework in case we add more adaptation strategies such as adding external service providers.

The approach presented in [59] formulates service composition as a constrained model, where each constraint represents a Quality of Service (QoS) aspect. This model performs runtime prediction of SLA violation/conformance for service orchestration based on monitoring information and the constraint model. SOPRA instead uses information from the application layer combined with statistical data to generate its predictions.

The work in [106] proposes a cloud workflow management system that dynamically provisions resources to process large scientific data sets. The authors of [77] present a rule-based approach that dynamically reconfigures the resources allocated to virtual machines (VMs). Their solution autonomically sets and adapts the threat thresholds of the rule-based system, in order to accommodate for variability in the workloads. Moreover, their system does not require any previous knowledge of the workloads. The solution in[111] strives to optimize resources by migrating VMs based on requests' response time. PAWS [24] is a very flexible adaptation framework for web service composition. We share the concept of extending the service interface with additional information; PAWS adds QoS data to select the most appropriate service, while we add the customer type to predict the future workload. Finally, Sandpiper [110] uses a hot spot detection algorithm to decide when to dynamically reallocate resources to VMs in a data center. Sandpiper plans reactions *after* violations have occurred and focuses mainly on hardware metrics. All these works are reactive systems, hence they cannot adapt before a violation occurs. Our work instead utilizes application-level information –the correlation between service invocations– to predict future workloads and to proactively allocate resources

The MOSES service adaptation broker [41] presents an interesting analysis of the tradeoffs between Java as a Web service and an ESB-based solution for service adaptation. With MOSES, we share the concept of per flow adaptation because we look at the aggregated volume of requests instead of at the single request.

The work in [79] presents a compositional approach that maps an abstract BPEL process to the real implementation of services. The composition is expressed as a constraint satisfaction optimization problem. We share the focus of the optimization, data center management, associated with this approach. The main difference with our work is that the composition is performed only at design-time.

SCENE[29] is a reactive framework that uses BPEL to define the process and a set of rules to discipline how the composition evolves at runtime via re-binding and re-configuration of the services. The main difference with our approach is in the optimization focus as we optimize resource utilization via workload predictions while SCENE makes sure that the set of rules are met.

In terms of auto-scaling techniques at the VM level, Autoflex [21] is a service-agnostic framework used to predicate future workload and auto-scale the infrastructure by dynamically selecting a predictor at runtime based on historical data. The approach in [61] relies on queuing theory to model the relationship between the number of VMs and the response time such that an appropriate number of VM instances can be predicated.

The work in [63] proposes an adaptive power management algorithm that dynamically changes the amount of power bought based on current workloads and renewable power available. SOPRA's predictive capability could be leveraged to further improve data center power purchasing strategies, such as the one in [63].

5.3 Motivating Example

To validate our approach, we used a common real life example of an online purchase. Each online customer probably has his or her own approach to online shopping, but common features can still be observed among them. As proof of concept, for our experiment we used two different groups of customers' usage patterns -our approach is independent from the number of groups. The users belonging to the first groups already know the e-commerce web site that they want to use for their purchase, and, after some browsing on its catalog, they typically purchase the items. Instead, the users of the second group do not know in advance what web site to go to, so they start shopping on a search engine. Upon receiving a request from a user, the search engine queries many different e-commerce web sites and returns a list of products that match the requested item. Typically, few requests are done to find the right product, and then the user proceeds to the selected e-commerce web site to finalize the purchase.

There is a interesting difference between the two usage patterns. In the first group, users look at few items on the web site, do a few searches on the internal catalog, and finally purchase one or more products. In the case of the second group of users, the search engine generates a multitude of search queries to retrieve as many products as possible that have similar characteristics to the product specified by the customer. Many e-commerce web sites receive those requests, but only one will eventually sell the product. So, the usage pattern of the first group shows a high correlation between the number of products seen and that of products purchased. On the contrary, in the usage pattern of the second group, the correlation between items observed and those purchased is significantly lower. These usage patterns can be used to predict what the future workload will look like. In fact, if we observe a high volume of traffic of first group's users searching through the catalog, we can expect a fair amount of purchases arriving in the near future. Instead,

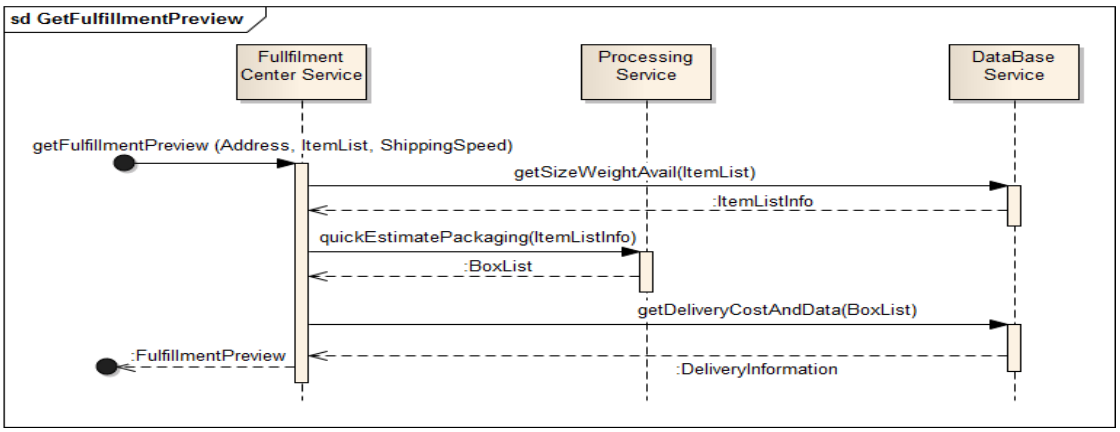

Figure 5.1. UML Sequence Diagram of getFulfillmentPreview

a high volume of view product requests from the second group of users will subsequently generate only a small amount of purchase requests.

5.3.1 Use Case Description

Based on the motivating example, as a use case for this work, we implemented a system that exposes web services based on the Amazon Fulfillment Center (FC) API [4]. The FC is the facility used by both the Amazon.com website and other e-commerce web sites to manage their inventory and handle deliveries of purchased products. All the functionalities of the Amazon Fulfillment Center are exposed as web services. For our experiment, we focused on two methods of the API: *getFulfillmentPreview* and *createFulfillmentOrder*. We decided to use the FC for our experiment because it is broadly adopted, well documented, and generic enough that the same approach we used for the FC can be applied to many other similar situations (e.g. online plane ticketing, hotel booking).

Fig. 5.1 shows the UML sequence diagram that describes a three-tier implementation for the *getFulfillmentPreview* request of the FC. Every time a user from group one clicks on a particular product to get more details, or every time the search engine used by group two's users queries the web site for product details, the e-commerce web site sends

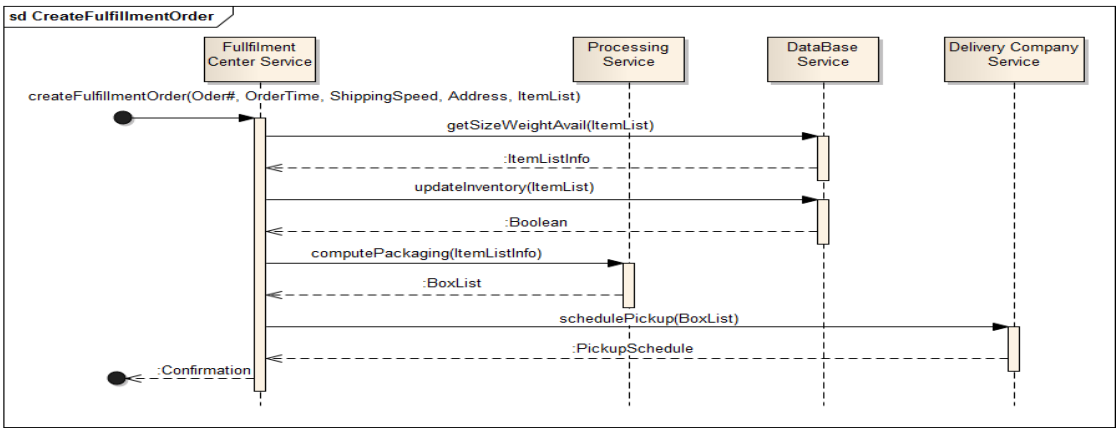

Figure 5.2. UML Sequence Diagram of createFulfillmentOrder

a *getFulfillmentPreview* request to the FC. The Fulfillment Center Service generates the response by first invoking the Database Service (*getSizeWeightAvail*) to get size, weight and availability of the list of items. Then, it invokes the Processing Service to get an estimate on the packaging for those items (*quickEstimatePackaging*). Finally, it queries the Database Service again to get information on the delivery.

Fig. 5.2 shows the sequence diagram that is executed when an item is purchased. Upon receiving a *createFulfillmenteOrder* request, the Fulfillment Center Service queries the Database again to get information about the items. It then sends a request to the Database to update the inventory (*updateInventory*) and to the Processing Service to compute the exact packaging required (*computePackaging*). Finally, it contacts the Delivery Service to schedule the pickup (*schedulePickup*).

5.4 Leveraging the Application Layer Information

5.4.1 The Key Insight

As of now, the two interactions shown in Fig. 5.1 and Fig. 5.2 are independent. This means that the resource manager of a data center cannot infer any useful information about the correlation between them; *getFulfillmentPreview* and *createFulfillmentOrder*

are just two separated operations exposed by the Fulfillment Center Service. But, as we previously observed, there is in fact a correlation between them:

- Before *createFulfillmentOrder* is invoked, one or more invocations of *getFulfillmentPreview* take place.
- The number of invocations of *createFulfillmentOrder* is strictly fewer than that of *getFulfillmentPreview*.
- The *getFulfillmentPreview* requests generated by group one's users directly on the e-commerce web page have higher chances to generate invocations of *createFulfillmentOrder* in the near future than those generated by group two's users, i.e. through search engines.

5.4.2 Leveraging Statistical Data

When it comes to resource management and optimization, the basic information described above, combined with some usage statistical data, can be used to make informed decisions about what the best infrastructure configuration would be. The statistical data on how different requests correlate with each other is application and usage dependent. Even though exact numbers are hardly ever disclosed to the public, for business reasons, it is well known that e-commerce web sites record a considerable amount of data to profile their users and the way they navigate their web sites (e.g. [22, 108]). Data analysis techniques to extract usage patterns from application logs are out of the scope of this paper; we mention [5] as an example of an available commercial solution. For this work, observations were collected from consenting lab members and used to form the statistical model:

- On average, we observed a two minute delay between when the item is initially displayed (*getFulfillmentPreview*) and when the purchase is finalized (*createFulfill-*

mentOrder). Such delay was caused by reading the item description, going through the login and filling out forms with address, credit card information, etc.

- Users shopping on Amazon.com browsed 2.5 items on average before proceeding with the purchase. In the statistical model, this translates into 40% of the *getFulfillmentPreview* generating a *createFulfillmentOrder* request.

- The majority of users shopping on the Google Shopping portal would select an item from the first page, which defaults to 20 items. In the statistical model, this translates into a 5% correlation between *getFulfillmentPreview* and *createFulfillmentOrder*.

5.4.3 Enriching Service Interfaces

Unlike the Fulfillment Center (FC), which cannot distinguish the kind of customer that is generating the request, the e-commerce web site has full knowledge of the incoming workload. Thus, in order to inform the FC about which user group is creating the load, we extended the service interface to carry the source type. When invoking a service operation (*op*), the service consumer (in our example the e-commerce web site) passes the source type, along with the other parameters, to inform the provider (the FC) that it is invoking *op* on behalf of that particular customer group. With this simple extension to the service interface combined with some statistical data, such as that discussed in Section 5.4.2, our framework is able to distinguish between the workflows of the two user groups and can make customer-aware, tailored predictions of the workload that the data center will receive in the near future. Therefore, if the e-commerce site shares this information, the FC can proactively adapt its configuration and provide a better service to the e-commerce web site. The SOPRA framework is described in detail in Section 5.5.

Applying this kind of information to the real Amazon Fulfillment Center API

Table 5.1. Execution Model for Fulfillment Center

Public Operation	**Private Operations**
getFulfillmentPreview(Address, ItemList, . . .)	getSizeWeighAvail(ItemList) quickEstimatePackaging(ItemListInfo) getDeliveryCostAndData(BoxList)
createFulfillmentOrder(Order#, OrderTime, . . .)	getSizeWeighAvail(ItemList) updateInventory(ItemList) computePackaging(ItemListInfo) schedulePickup(BoxList)

would not have a disruptive effect on the existing e-commerce web sites currently using it. In fact, the new interface could be offered as a new web service, leaving the choice of migrating to the clients (the e-commerce web sites). Both parties have an economical interest to adopt this approach and to use accurate data: the provider can provide a better and cheaper service to the user if its data is accurate. This translates into the provider being able to be more competitive on the market and the user saving some money. The backup plan is defaulting to a customer unaware allocation strategy, which is more expensive for the user.

5.5 SOPRA Middleware Implementation Overview

As briefly discussed in the previous sections, our approach to proactive adaptation uses the following steps: i) monitor incoming requests to determine their source types and volume, ii) use this data to predict what the future workload will be, iii) combine the prediction with resource usage data to determine the amount of resources needed, iv) update the system configuration and web service composition. In order to achieve these steps, the SOPRA proactive framework is composed of six main components: i) *System Monitor*, ii) *Predictor*, iii) *System Configurator*, iv) *Performance Model*, v) *Execution Model* and vi) *Load Balancer*.

The *System Monitor* is responsible for monitoring the number of incoming re-

quests for each public web service operation (i.e. *getFulfillmentPreview* and *createFulfillmentOrder*). Note that two requests with different *SourceType* are distinguished.

The *Predictor* takes the monitored data from System Monitor and combines it with the statistical data (see Section 5.4.2) to predict the future workload. In our example, the Predictor calculates how many *createFulfillmentOrder* requests the system should expect in two minutes; this value factors in requests from both groups' *SourceType*.

The *Execution Model* is a mapping from a public web service operation to the set of private operations comprising it. In our example, the execution model is shown in Table 5.1.

The *Performance Model* maps each web service operation to the number of requests that the server can handle for that operation within the period t_{sampl}, without violating any aspect of the SLA. In order to create this model, we stress-tested the servers of our data center with each operation exposed by each web service independently. These metrics allow us to understand the resources consumed by a single invocation of each web service operation.

Input: workload prediction (*pred*), execution model (*exMod*), performance model (*perfMod*)
Output: Map of required number of servers for each public web service operation (*pubWsOps*)

```
1 reqServers ← ∅ ;
2 foreach pubOp_j ∈ pubWsOps do
3 |   foreach privOp_i ∈ exMod(pubOp_j) do
4 |   |   reqServers(pubOp_j) ← reqServers(pubOp_j) + (1 /
  |   |   perfMod(privOp_i) * pred(privOp_i, pubOp_j) ;
5 |   end
6 end
```

Algorithm 3: Compute Configuration algorithm

The *System Configurator* combines the workload prediction with the *Performance* and *Execution Models* to determine the amount of resources that will be needed at time $t + t_{prediction}$ – see Table 5.2 for variables explanations.

Table 5.2. Variables of the Adaptation Algorithm

Variable Name	Meaning
$t_{prediction}$	Prediction window for future workload
t_{sampl}	Frequency of sampling
$t_{adaptation}$	Adaptation interval

Fig. 3 shows the algorithm that computes the configuration. We deem "public" operations to be those exposed by the Fulfillment Center web service (i.e. *getFulfillment-Preview* and *createFulfillmentOrder*), whereas those exposed by the internal web services (i.e. Processing Service, Database Service, Delivery Company Service) are deemed "private". The algorithm takes the future workload prediction (*pred*), the execution model (*exMod*), and the performance model ($perfMod$) as inputs. *reqServers* maps public operations $pubOp_j$ to the number of servers needed by the web service that exposes $pubOp_j$ in order to satisfy the SLA. Since we deploy only one web service per physical machine, the number of servers is equivalent to the number of service instances. Then, after initializing *reqServers* (step 4), for each public web service operation $pubOp_j$ (step 5), it loops through all the private operations $privOps_i$ comprising that operation (step 6), and, for each of the operations, computes the number of service instances required to respond to the amount of requests defined in the prediction (step7). Finally, the algorithm returns the number of servers (*reqServers*) that will then be used to update the infrastructure configuration.

Every $t_{adaptation}$, the System Configurator retrieves the configuration computed for that time point and updates the infrastructure with the appropriate amount of resources. The value of $t_{adaptation}$ determines how often the system adapts its configuration. The higher the frequency, the more sensitive to the spikes SOPRA becomes. The $t_{adaptation}$ value should be defined after a thorough analysis of the type of workload the infrastructure has to support. Among the aspects to consider during this analysis we mention i) the

tolerance of the SLA w.r.t. response time (if the average response time is very close to the SLA value, the infrastructure cannot tolerate spikes of demand without prompt adaptation), ii) The historical trend of the workload (if a workload tends not to have frequent spikes, a lower adaptation frequency can be tolerated), and iii) The accuracy of the statistical data (if a workload has a fixed and known interval between operations (e.g. between *getFulfillmentPreview* and *createFulfillmentOrder*), then $t_{adaptation}$ can be tailored to that interval). A complete description of this analysis is out of the scope of this paper.

5.6 Evaluation

5.6.1 Experimental Setup

To evaluate the effectiveness of SOPRA's ability to proactively adapt a data center's configuration, we created a running prototype of our approach. This prototype uses the information on the correlation between web services and the statistical data we described earlier to manage the resources allocated for our implementation of the FC. The setup consists of seven physical machines, each equipped with two Intel(R) Xeon(R) CPUat 2.66 GHz running Microsoft Windows Server 2008 R2. Three of these machines are allocated to the Processing Service, while the other two are allocated to Delivery Service and Database Service respectively. One server runs the Fulfillment Center Service and the SOPRA Framework prototype, including the *System Monitor*, *Load Balancer*, *Predictor*, *System Configurator*, and the *Performance and Execution Models*. In our setup, we set the adaptation interval $t_{adaptation}$ and sampling window t_{sampl} to 30s and 5s respectively. Note that, in our setup, we limit the resource adaptation to Processing Service, because it requires considerably more computational power and, therefore, has much longer invocation time; in other words, it is the bottleneck.

To validate our approach, we used the Apache JMeter workload generator to emulate two different workloads, shown in Fig 5.3 and 5.4, with customer profiles consistent with the statistical data discussed in Section 5.4.2. In both workloads, *Preview ID1* and *Preview ID2* represent requests from browsing e-commerce web sites and from search engines, respectively. Both are instances of *getFilfillmentPreview* with *SourceType*. We use *Preview* to refer to requests including *Preview ID1* and *Preview ID2*, and use *Order* to refer to *createFulfillmentOrder()* requests. Both workloads have a baseline input pattern (about 40 requests every 5 seconds for both *Preview ID1* and *Preview ID2*), which simulates normal load on the FC. Note that the number of *Order* requests correlates with that of *Preview*, according to the statistical model. In the first workload, there are two spikes of preview requests, which both last for about 15 seconds; in the second, the two spikes last for about two minutes. The spikes of preview requests lead to spikes of *Oder* requests after roughly two minutes. For both workloads, the first spike generates about 120 *Preview ID1* requests, and about 140 *Preview ID2* requests every 5 seconds; the second spike, on the other hand, generates 70 *Preview ID1* and 100 *Preview ID2* requests every 5 seconds.

To generate the performance model, we obtained the maximum number of requests over 5 seconds (t_{sampl}) that each private operation can handle and the average response time observed when the service providing that operation is saturated. Theoretically, the maximum number of requests over 5 seconds that a public operation can handle is equal to that of the bottleneck of its comprising operations. However, our experiments show that this number is in fact smaller, because i) stress testing of individual operations does not factor in network communication time, which is unavoidable in service composition; ii) sequential service composition means that, when a service is invoked, it depends upon both the availability of the service and the completion of the previous service involved in the composition. Therefore, we introduce a load factor

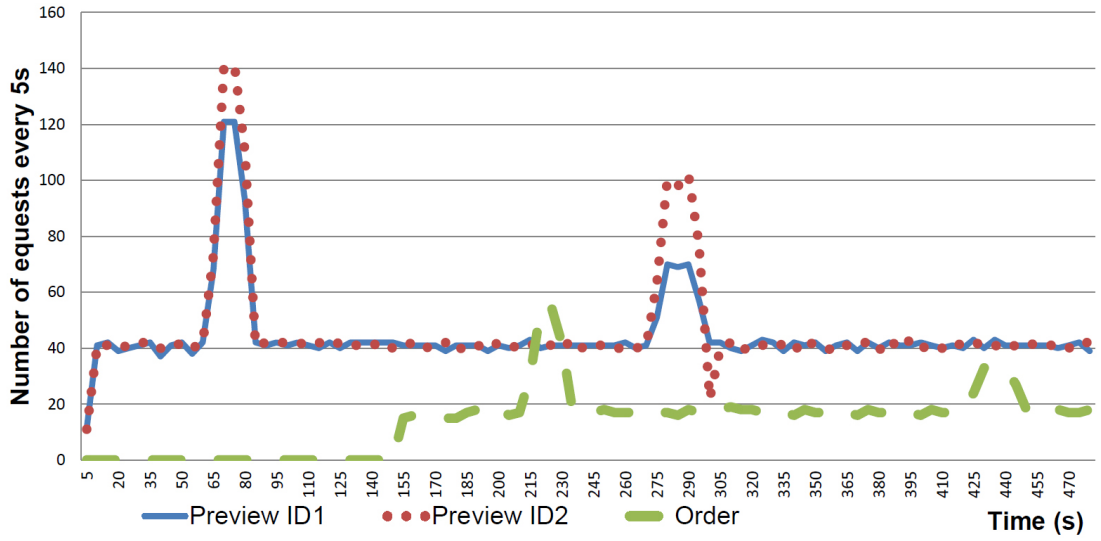

Figure 5.3. Workload One

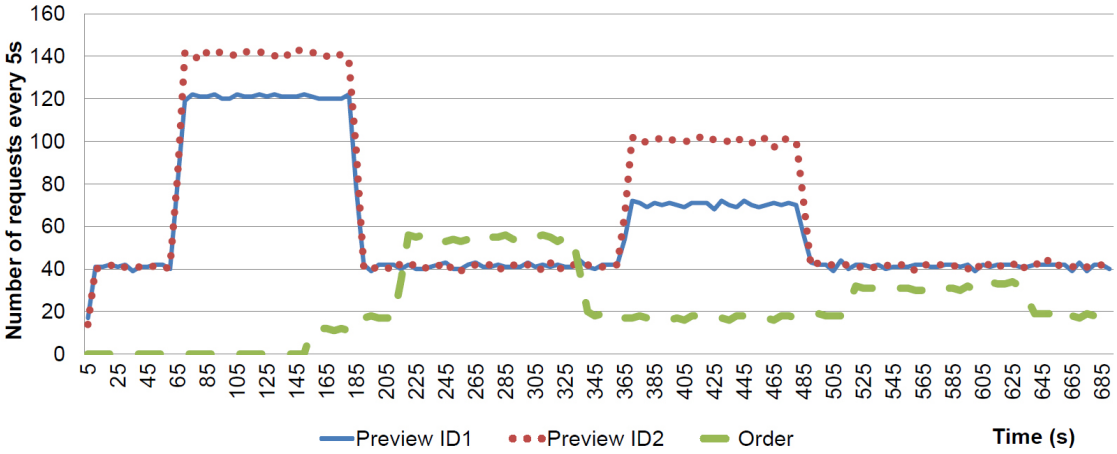

Figure 5.4. Workload Two

(lf) to discount the maximum number of requests handled by individual services. The calculation of lf follows Eq. 5.1,

$$lf = MaxReqNo(o_p) / \min_{o_i \in O}(MaxReqNo(o_i)) \tag{5.1}$$

where o_p is the public operation in a composite service, O is the set of private operations

composing o_p, and *MaxReqNo* is a mapping from an operation to the maximum number of requests that the operation can handle. Note that we distinguish between an operation and the service that provides the operation, because a service can provide multiple operations. Table 5.3 shows the performance model used in our experiments with 0.7 as the load factor. The SLA for *Preview* and *Order* are obtained by summing up the response time of individual operations involved.

We compare the effectiveness of SOPRA with that of a reactive approach and an over-provisioning approach with respect to average response time and resource utilization. The reactive approach is implemented by setting the number of service instances that provide a web service operation op to n_r for every $t_{adaptation}$ time, defined in Eq. 5.2, where $t_{rsp}(op)$ is the current average response time of invoking op during the t_{sampl} interval, and $t_{SLA}(op)$ is the average response time promised by SLA defined in Table 5.3. This equation assumes a linear approximation of the relationship between response time and the number of resources. Note that, in our implementation of the reactive adaptation, $t_{sampl} = t_{adaptation}$ always holds.

$$n_r = \lceil t_{rsp}(o)/t_{SLA}(o) \rceil \tag{5.2}$$

Because our experimental setup only optimizes the number of Processing Service instances, the reactive approach only monitors the response time of operations provided by the Processing Service (i.e. *computePackaging* and *quickEstimatePackaging*). The responsiveness of the reactive approach is determined by $t_{adaptation}$(or t_{sampl} since they are equal in our implementation). According to these two factors, we run the workloads with two reactive adaptation strategies: i) $t_{adaptation} = 30s$; ii) $t_{adaptation} = 2s$, called Reactive (1) and Reactive (2), respectively. The adaptation interval of Reactive (1) is inline with that of SOPRA. We choose $2s$ for Reactive (2), because that's the largest

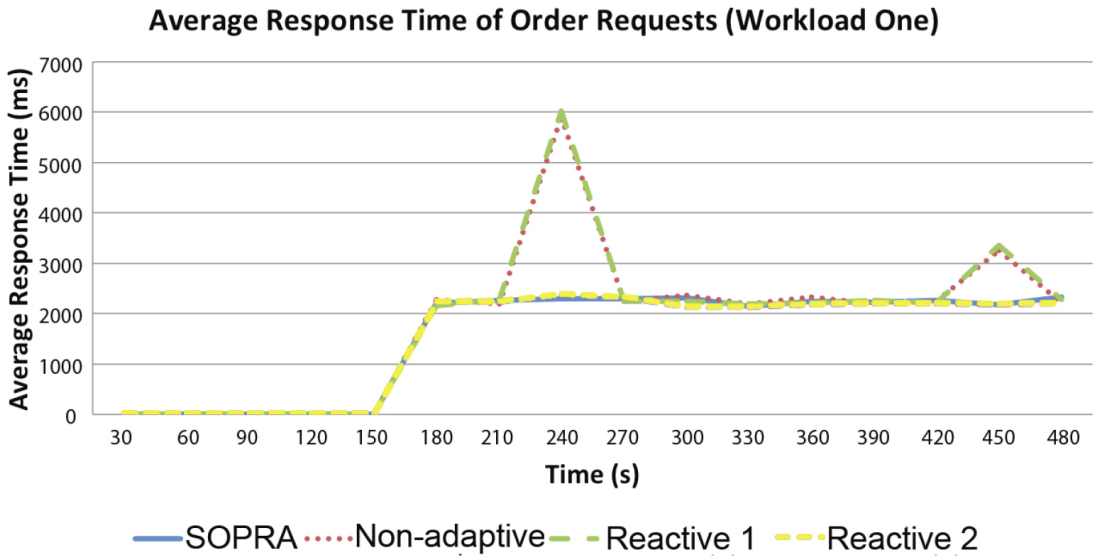

Figure 5.5. Avg. Rsp. Time of Orders for Workload One

Table 5.3. The Performance Model

Operation Name	Max Req. No. per 5s	SLA - Avg. Rsp. Time (ms)
GetSizeWeightAvail	350	216
GetDeliveryCostAndData	350	256
UpdateInventory	350	350
QuickEsitmatePackaging	280	311
ComputePackaging	28	1678
SchedulePickUp	420	250
Preview SLA	783 ms	
Order SLA	2494 ms	

adaptation interval that enables it to be competitive with SOPRA in terms of response time violations. Obviously, the second strategy is more aggressive than the first one. To establish a baseline for comparison, we also run the workloads with an non-adaptive approach, using one Processing Service instance for all requests. For all approaches, we averaged over three runs to smooth stochastic anomalies.

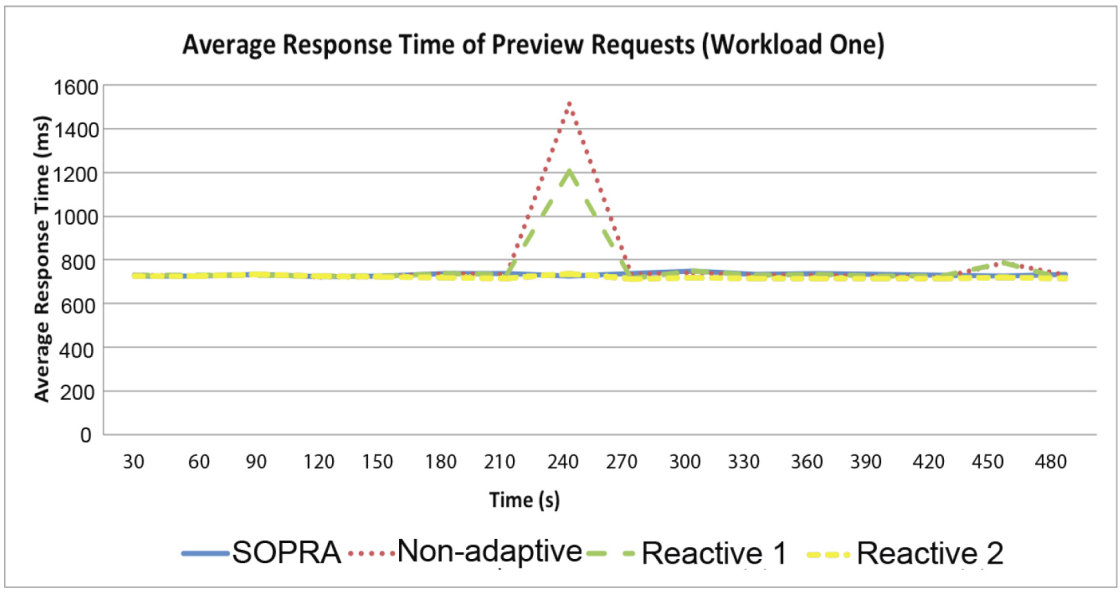

Figure 5.6. Avg. Rsp. Time of Previews for Workload One

5.6.2 Evaluation of Response Time

Fig. 5.5 and Fig. 5.6 show the average response time of *Order* and *Preview* requests, respectivel, for the first workload (Fig. 5.3). Since the spikes only last for 15 seconds, and adaptation happens every 30 seconds, Reactive (1) does not react fast enough to adapt the amount of allocated resources, meaning the *Order* spikes have already passed when more resources are finally allocated. The high volume of *Order* requests also leads to increased response time of *Preview* requests for Reactive (1) due to resource contention for Processing Service. Only SOPRA and Reactive (2) guarantee the SLA during the spikes, achieved in two distinct ways. SOPRA predicts that there will be a big spike of *Order* requests starting at $t = 210$, and a small spike starting at $t = 420$, based on the observation that a significant spike of *Preview* requests happens at $t = 60 - 75$, and a smaller spike happens at $t = 270 - 295$. Accordingly, SOPRA allocates three Processing Service instances, hence 3 physical machines, for the 30 seconds starting at $t = 60$, and two processing services for the 30 seconds starting at $t = 270$. After the spikes, SOPRA

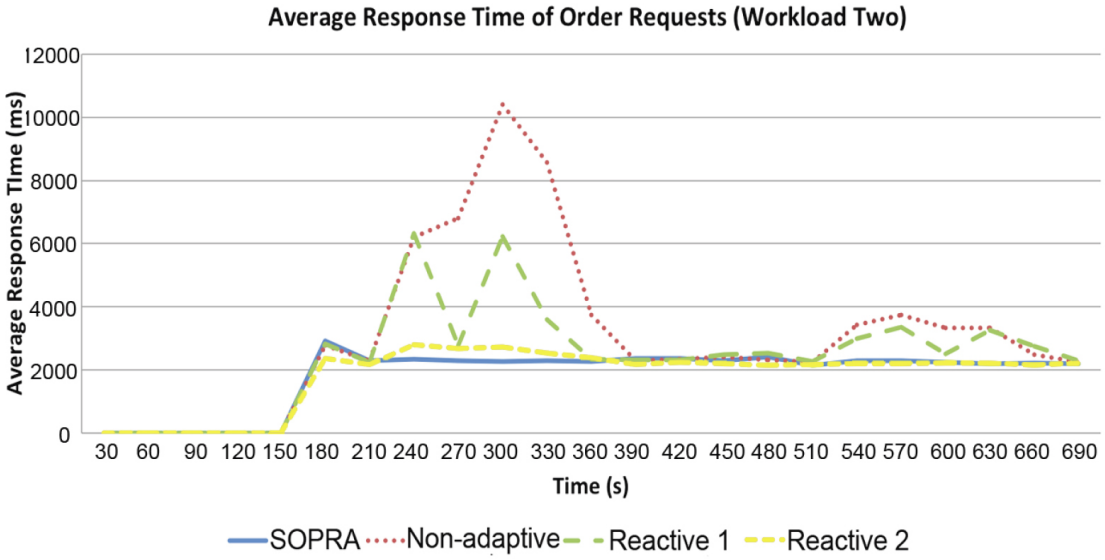

Figure 5.7. Avg. Rsp. Time of Orders for Workload Two

adaptively reduces the number of Processing Service instances by predicting that the incoming loads will return to normal levels. Reactive (2) monitors the average response time for each processing operation, and responds by setting service instances according to Eq. 5.2 every $2s$. In this regard, SOPRA achieves up to a 60% reduction in response time during spikes when compared to a reactive approach with the same adaptation frequency.

Fig. 5.7 and Fig. 5.8 illustrate the average response time of *Order* and *Preview* requests, respectively, for the second workload (Fig. 5.4). Again, only SOPRA and Reactive (2) guarantee the SLA during the spikes. Apart from the first workload, the performance of Reactive (1) improves under this workload. We observe wave-shaped curves in the spikes for Reactive (1), oscillating every 30 seconds. The pullback of the curve ($t = 240 - 270$) indicates that the system reacts to request spikes and accordingly adapts the configuration by increasing the number of Processing Service instances. However, this is followed by a response time increase ($t = 270 - 300$), because the system falsely believes that the current workload has been reduced based on the fact that response time has been reduced, therefore reducing the number of Processing Service

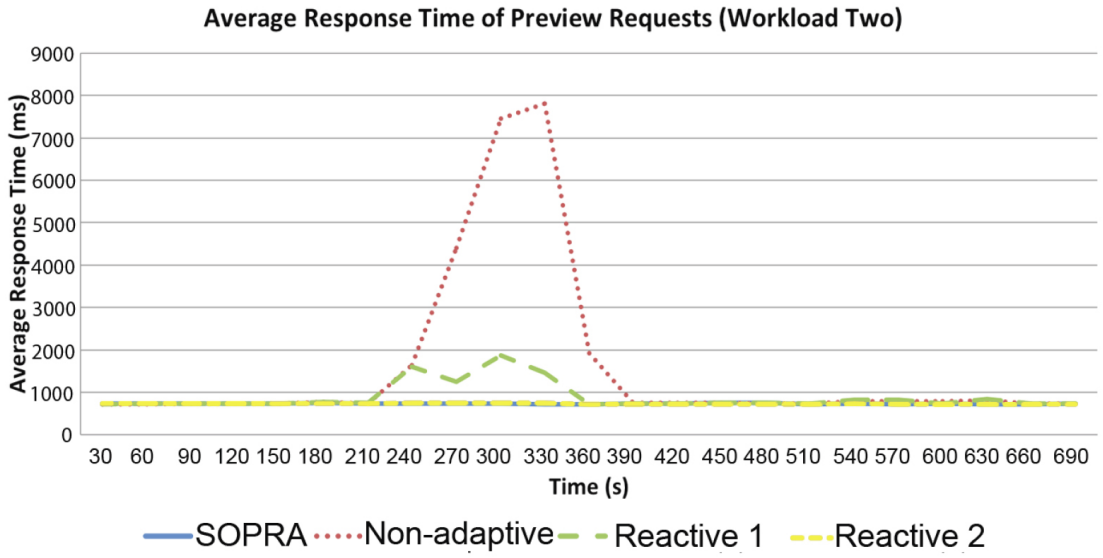

Figure 5.8. Avg. Rsp. Time of Previews for Workload Two

instances; this is caused by the limitations of our basic implementation of Reactive (1). The same behavior exists in Fig. 5.8. The point is that, in order to achieve satisfactory response time, the reactive approach should decrease its adaptation interval and therefore adapt more aggressively (once every 2s in our case).

5.6.3 Evaluation of Resource Allocation

Solely performing better with respect to response time does not justify the superiority of an approach over others, since one could simply allocate all available resources all the time to meet possible workload spikes. For instance, in our experiment, we can make all three Processing Service instances available throughout the whole workload. This is what is commonly done when an over-provisioning approach is adopted; however, this is a suboptimal method in terms of resource utilization. To illustrate the point, we present the percentage of time the approaches allocate one, two or three Processing Service instances for *Order* requests in the second workload, shown in Table 5.4. We only consider approaches that achieve the SLA – that is SOPRA, Reactive (2) and

Table 5.4. Resource Utilization for Orders in Workload Two

No. of Processing Service	1	2	3
SOPRA	68%	16%	16%
Reactive (2)	82%	14%	4%
Over-provisioning	0	0	100%

the over-provisioning approach. Compared to the over-provisioning approach, SOPRA reduces unnecessary resource allocation 84% of the time (68% of the time using one instance and 16% of the time using two instances instead of three). Interestingly, Reactive (2) outperforms SOPRA in this aspect because of employing a much more aggressive adaptation strategy by using a very short sampling and adaptation interval (2s). However, such an aggressive strategy could not be feasible in practice, given that one needs to factor in the time to allocate the machines. Thus, even though Reactive (2) is able to calculate new configurations, it does not have sufficient time to carry out adaptations.

In summary, SOPRA did not incur in any SLA violation, while simultaneously consistently out-performing, in terms of violations and resource allocation, both the non-adaptive and the reactive approach, given the same adaptation frequency. Reactive (2) showed similar results to SOPRA, but it required a 15-fold shorter adaptation frequency in order to achieve those results, hence making SOPRA the best solution overall.

5.7 Conclusion and Future Work

SOPRA is a framework that supports proactive allocation of resources in a data center. In this paper, we extended SOPRA by making it *customer-aware*, that is, able to distinguish among types of customers with different usage patterns. We also proposed an extension to service interfaces to carry user types. Usage statistical data are combined with an execution model to generate customer-aware predictions of future levels of workload. These predictions are used to proactively allocate resources right before they

are needed, thus improving resource utilization while minimizing SLA violations. We implemented a prototype of the framework and applied it to a real life example. The experimental results show up to a 60% reduction in response time during spikes when compared to a reactive approach with the same adaptation frequency; they also show up to an 84% reduction in the amount of servers allocated, hence spared, over time compared to a typical over-provisioning approach. These servers could then be used to do extra work or turned off to save on energy costs.

Future directions for this work include increasing the data center throughput thanks to a smart scheduling of multiple public web services (in this work we had only one) and batch jobs on partially utilized physical machines. It is important to point out that, even though the focus of this paper is on the allocation of resources, this is not the only possible adaptation strategy that our predictions enable. In fact, the same kind of predictions could be leveraged to smartly migrate VMs, recompose the SOA with different service providers, and even detect workload anomalies, such as denial of service attacks. These alternatives are left for future work.

5.8 Acknowledgments

The work in this chapter has been funded by NSF grant 0821155, MuSyC, EC FP7 grant 285939 (ACROSS) and Cisco Systems, Inc.

This chapter, in full, is a reprint of the material as it appears in Seracini, Filippo; Zhang, Xiang; Rosing, Tajana; Krüger, Ingolf, "A Proactive Customer-Aware Resource Allocation Approach for Data Centers" at the 12th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), IEEE, 2014. The dissertation author was the primary investigator and author of this material.

Approach for Data Centers" at the 12th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), IEEE, 2014.

Chapter 6

Conclusions and Future Outlook

In this chapter, we conclude the work of this dissertation and propose directions for future research opportunities in this field.

6.1 Conclusions

In this dissertation, we have made two contributions: *1) we have proposed an approach to predict future workloads by leveraging correlations between requests sent to an application. 2) We have shown how to estimate, with acute accuracy, the impact that those workloads have on the performance of the currently allocated infrastructure.* With this work, we have proven that the problem with the allocation of resources in data centers can be largely mitigated with an integrated holistic top-down approach, that proactively adapts the amount of resources allocated, based on the incoming future workloads. We have shown that our approach can be applied to workloads where different requests are correlated, such as service-oriented Internet applications. We have incrementally accomplished our vision of proactive adaption in chapters 3 to 5, where we have demonstrated that, by combining a performance model with the statistical and execution models of an application, we were able to predict incoming workloads, understand their impact on the performance of the currently allocated infrastructure, and proactively adapt it whenever needed. Our proactive approach significantly improves

the state of the art, yielding up to a 52% increase in energy savings and up to an 84% reduction in the amount of resources allocated, without incurring any additional SLA violations.

We first started in Chapter 1 by analyzing the current problems and limitations with the allocation of resources in data centers that host Internet applications. We modeled the relationship between SLAs and the services that build such Internet applications and discussed how, given the current state of the art, this relationship yields to extreme under-utilization of data centers.

We then introduced our approach by building the roadmap for this research work, shown in Figure 1.7. We introduced the Statistical, the Execution, and the Performance models and explained the advantages that a proactive approach can bring to resource allocation.

In the first step of our road-map, Chapter 3, we built a performance model that takes into consideration the entire hardware and software stack. With this holistic approach, the performance model was able to achieve predictions with very high accuracy, enabling the autonomic resource management system to dynamically allocate just the right amount of resources at any given point in time. Thanks to our performance model, we were able to reduce the amount of resources allocated by 42.5%, compared with an existing approach from scientific literature.

In the second step of the road-map, Chapter 4, we introduced the Service Oriented PRoactive framework, which, thanks to the Statistical and Execution models, was able to predict incoming workloads and proactively allocate resources. We showed that, compared with a static, offline approach, we could significantly reduce the amount of resources allocated, yielding up to 52.49% of energy savings, without introducing any extra SLA violations.

In the final step, Chapter 5, we further improved our workload prediction accuracy

by tailoring those predictions to customers' usage patterns. We showed that different customers can have very different usage patterns and, by taking those patterns into account, SOPRA was able to achieve up to an 84% reduction in the amount of servers allocated, compared to a typical offline provisioning approach.

It is important to note that one the main contributions brought by this dissertation was to take a top-down approach to resource management. We looked at the application layer, in particular that of service-oriented applications, and we leveraged the information already available at that layer to predict incoming workloads. We then proposed a multi-layer, holistic approach that leverages the entire software and hardware stack in order to assess the amount of resources that will be needed for the predicted workloads.

It is also important to note that the approach presented in this dissertation is complementary to many existing, also commercial, solutions. For example, our approach could be used in conjunction with an algorithm for smart placement of VMs; it would simply be an adaptation strategy. Our framework would detect the need for adaptation before an event occurs, and would then invoke the smart placement algorithm to determine where to allocate the required VMs. Moreover, a framework like SOPRA could be implemented by leveraging existing components, such as for example, for the monitoring part or to build the statistical model, hence making this whole approach reusable and quickly marketable. This property was one of the main reasons why this research won the Von Liebig Entrepreneurism Center's "Most Promising Innovation in IT Award" at the 2013 ResearchExpo at the University of California, San Diego.

6.2 Future Outlook

In each of the previous chapters, we mentioned possible future directions related to the approach proposed in this dissertation. Most of them concerned themselves with about specific technical details and improvements to the proposed work in each chapter.

In this section, we propose longer-term research ideas and suggest different kinds of applications where this approach could be leveraged.

6.2.1 Increasing the Throughput

The resource utilization problem in data centers can be tackled from two complementary directions. The first direction aims at minimizing the amount of resources allocated to satisfy a given workload. In other words, this direction strives to minimize the cost of providing a service by maximizing the use of the resource allocated, while releasing or turning off the remaining ones. The work presented in this dissertation mostly followed this direction.

The second direction instead, aims at maximizing the amount of work a service provider can offer, given its infrastructure. In other words, this directions strives to maximize the amount of work that a service provider can serve at any point in time. As also mentioned in the previous chapters, this could be achieved by allocating the unused capacity to low priority jobs and/or tasks that can be preempted whenever additional capacity is needed for workloads with more stringent SLAs.

6.2.2 Possible Applications

In this section we discuss possible applications of the approach proposed in this dissertation.

Virtual Machine Migration

Even though, in the experiments presented in this dissertation, we only focused on applications deployed on bare hardware, the capability of predicting incoming workloads could also be leveraged to proactively change the allocation and the configuration of virtual machines. Virtualization is indeed the de facto standard in public clouds and is increasingly being adopted also by enterprises. In the case of a higher incoming

workload, the consolidation rate (i.e. the number of VMs allocated per physical server) could be lowered by spanning VMs across more physical hosts. Furthermore, whenever the workload mix is expected to change, another possible adaptation strategy could be to change the configuration of the VMs running that workload. So, for instance, if the workload mix shifts towards a more CPU-intensive kind of load, the VMs could be reconfigured to have more CPU-cores allocated to them or moved to physical hosts with faster CPUs, hence leveraging the hardware heterogeneity that exists in data centers [76]. Changing the configuration of the VMs could also be extended to renting particularly high performance machines on a public cloud during spikes. The ability to dynamically scale to the public cloud is commonly referred to as hybrid cloud.

It is easy to see how many combinations of the adaptations described above could be put in place, especially considering the different price offerings of cloud providers. Hence, devising an autonomic system that supports hybrid clouds, that can be easily reconfigured depending on the economic offering, and that automatically determines the most convenient adaptation approach, would be a very interesting research direction as well as a very marketable product.

SOA Recomposition

In a service-oriented application, the unit of adaptation is not limited to the physical server or the VM hosting the service(s). Indeed, an SOA is an application that offers a set of functionality by aggregating multiple services in a coordinated way. These services can be located anywhere and it is often the case that the same service can be offered by many providers, e.g. a weather forecast service. Similarly to what was discussed above in the context of hybrid clouds, an SOA could adapt its composition by selecting service providers based on different price and performance (i.e. SLA) offerings. So, for instance, with a predicted increase of the incoming workload, an SOA

could autonomously reconfigure itself by selecting a service provider that offers better performance, but at a higher price. Once the workload decreases, the SOA could then switch back to a provider with a cheaper and less performing service. A scenario like this is better supported by a proactive system, since it has the time to complete the adaptation before the spike actually hits. In fact, it could be the case that multiple adaptation steps need to be accomplished in order to switch from a provider to another, e.g. reconfigure the load balancer, the network configuration, turn OFF/terminate unused services, buy additional capacity, etc.

Anomalies Detection

Whenever we can predict an incoming load and assess the impact that such load will have on the allocated infrastructure, several anomaly detection strategies can be put in place. Let us assume for instance that we predicted a load and we estimated a certain utilization of the infrastructure. We can then monitor the actual usage and, if we observe significant drift from the expected value, we know that something is wrong. If the utilization is much lower than expected, it could mean that a network malfunctioning is happening somewhere, preventing the load from reaching the system. This information could then be used to trigger network diagnostics. On the contrary, if an unexpected load hits the system, it could be the result of a Denial of Service attack. Again, we could use this information to increase the protection level and trigger mitigation strategies. Similarly, if the workload mix drifts from the expected one, it could be the result of some virus or attack to the infrastructure and remediation actions could be triggered.

In summary, the ability of predicting future workloads, associated with a highly accurate performance predictor, enables a wide set of scenarios, where detection and mitigation strategies can be put in place to support the business of a service provider.

Bibliography

[1] Aerial view of microsoft data center in san antonio, texas. http://www.datacenterknowledge.com/wp-content/uploads/2009/11/Microsoft-SanAntonio-web.jpg.

[2] Amazon AWS Auto Scaling. http://aws.amazon.com/autoscaling/.

[3] Amazon elastic compute cloud documentation. https://aws.amazon.com/documentation/ec2/.

[4] Amazon Fulfillment Web Service. http://docs.amazonwebservices.com/fws/1.1/GettingStartedGuide.

[5] Analytics | SAS. http://www.sas.com/technologies/analytics/.

[6] Apache JMeter. http://jmeter.apache.org/index.html.

[7] Autoscaling and Windows Azure. http://msdn.microsoft.com/en-us/library/hh680945(v=PandP.50).aspx.

[8] Google.com, council bluff, iowa. http://www.google.com/about/datacenters/gallery/index.html#/locations/council-bluffs/2.

[9] The Green Grid productivity indicator. White Paper #15:10.

[10] IBM cloud and smarter infrastructure. http://www-01.ibm.com/software/tivoli/.

[11] Microsoft | System Center 2012 R2. http://www.microsoft.com/en-us/server-cloud/system-center/system-center-2012-r2.aspx.

[12] Microsoft Windows Server 2012 R2. http://www.microsoft.com/en-us/server-cloud/products/windows-server-2012-r2/default.aspx#fbid=wlUKTJfKdVF.

[13] Mule ESB. http://www.mulesoft.org/home.

[14] Open source software for building private and public clouds. http://www.openstack.org/.

[15] RUBiS – Rice University Bidding System – http://rubis.ow2.org/.

[16] VMware virtualization for desktop & server, public & private clouds. http://www.vmware.com/.

[17] Wikipedia - blade server. http://en.wikipedia.org/wiki/Blade_server.

[18] Industry perspective: Energy efficiency and renewable sources for the data center. *Data Center Journal*, 2013.

[19] Agency. Report to Congress on server and data center energy efficiency public law 109-431. Technical report, United States Environmental Protection Agency, 2007.

[20] V.A.F. Almeida and D.A. Menasce. Capacity Planning an Essential Tool for Managing Web Services. *IT Professional*, 4:33–38, August 2002.

[21] F.J. Almeida Morais, F. Vilar Brasileiro, R. Vigolvino Lopes, R. Araujo Santos, W. Satterfield, and L. Rosa. Autoflex: Service agnostic auto-scaling framework for iaas deployment models. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 42–49, 2013.

[22] Nikolay Archak, Anindya Ghose, and Panagiotis G. Ipeirotis. Show me the money!: deriving the pricing power of product features by mining consumer reviews. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 56–65, New York, NY, USA, 2007. ACM.

[23] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani. PAWS: a framework for executing adaptive web-service processes. *Software, IEEE*, 24(6):39 –46, December 2007.

[24] D. Ardagna, M. Comuzzi, E. Mussi, B. Pernici, and P. Plebani. Paws: A framework for executing adaptive web-service processes. *Software, IEEE*, 24(6):39 –46, nov.-dec. 2007.

[25] Danilo Ardagna, Barbara Panicucci, Marco Trubian, and Li Zhang. Energy-Aware Autonomic Resource Allocation in Multitier Virtualized Environments. *IEEE Transactions on Services Computing*, 5(1):2–19, 2012.

[26] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[27] Matthew Arrott, Barry Demchak, Vina Errnagan, Claudiu Farcas, Emilia Farcas, Ingolf H. Kriiger, and Massimiliano Menarini. Rich services: The integration piece of the soa puzzle. *2013 IEEE 20th International Conference on Web Services*, 0:176–183, 2007.

[28] Rafael Aschoff and Andrea Zisman. Qos-driven proactive adaptation of service composition. In Gerti Kappel, Zakaria Maamar, and HamidR. Motahari-Nezhad, editors, *Service-Oriented Computing*, volume 7084 of *Lecture Notes in Computer Science*, pages 421–435. Springer Berlin Heidelberg, 2011.

[29] L. Baresi and S. Guinea. Self-supervising bpel processes. *Software Engineering, IEEE Transactions on*, 37(2):247 –263, march-april 2011.

[30] Luciano Baresi and Sam Guinea. Self-Supervising BPEL Processes. *IEEE Transactions on Software Engineering*, 37(2):247–263, 2011.

[31] Luciano Baresi and Sam Guinea. Event-based Multi-level Service Monitoring. In *Proceedings of the 20th IEEE International Conference on Web Services (ICWS)*, pages 83–90, 2013.

[32] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward Open-World Software: Issue and Challenges. *Computer*, 39(10):36–43, 2006.

[33] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[34] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu. Model-based Adaptive DoS Attack Mitigation. In *Proceedings of the 2012 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 119–128, June 2012.

[35] Luiz Andr Barroso and Urs Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, December 2007.

[36] Anton Beloglazov and Rajkumar Buyya. Energy efficient resource management in virtualized cloud data centers. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 826–831, Washington, DC, USA, 2010. IEEE Computer Society.

[37] Josep Ll. Berral, Ricard Gavalda, and Jordi Torres. Adaptive scheduling on power-aware managed data-centers using machine learning. In *Proceedings of the 2011 IEEE/ACM 12th International Conference on Grid Computing*, GRID '11, page 6673. IEEE Computer Society.

[38] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. JMT: Performance Engineering Tools for System Modeling. *SIGMETRICS Performance Evaluation Review*, 36(4):10–15, 2009.

[39] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. Power aware computing. chapter The Case for Power Management in Web Servers, pages 261–289. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[40] 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, 2nd edition edition, 2006.

[41] V. Cardellini and S. Iannucci. Designing a broker for qos-driven runtime adaptation of soa applications. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 504 –511, july 2010.

[42] G. Casale and G. Serazzi. Bottlenecks Identification in Multiclass Queueing Networks using Convex Polytopes. In *Proceedingds of the 12th IEEE Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 223–230, 2004.

[43] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 103–116, Banff, Alberta, Canada, 2001. ACM.

[44] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 303–314, Banff, Alberta, Canada, 2005. ACM.

[45] Yu Dai, Lei Yang, and Bin Zhang. Qos-driven self-healing web service composition based on performance prediction. *Journal of Computer Science and Technology*, 24:250–261, 2009.

[46] Frederico Alvares de Oliveira,Jr. and Thomas Ledoux. Self-management of applications QoS for energy optimization in datacenters. In *Green Computing Middleware on Proceedings of the 2nd International Workshop*, GCM '11, page 3:13:6, New York, NY, USA, 2011. ACM.

[47] Gaurav Dhiman, Giacomo Marchetti, and Tajana Rosing. vGreen: A System for Energy-Efficient Management of Virtual Machines. *ACM Transactions on Design Automation of Electronic Systems*, 16(1):1–27, 2010.

[48] Staff Editor. Gartner Estimates ICT Industry Accounts for 2 Percent of Global CO2 Emissions. *Gartner*, 2007.

[49] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 13–23, New York, NY, USA, 2007. ACM.

[50] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal Autoscaling in a IaaS Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)*, pages 173–178, 2012.

[51] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008.

[52] Steve Hamm. It's Too Darn Hot. *BloombergBusinessweek*, 2008.

[53] R.R. Harmon and N. Auseklis. Sustainable it services: Assessing the impact of green computing practices. In *International Conference on Management of Engineering Technology,PICMET*, pages 1707–1717, Aug 2009.

[54] Yuxiong He, Zihao Ye, Qiang Fu, and Sameh Elnikety. Budget-based Control for Interactive Services with Adaptive Execution. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)*, pages 105–114, 2012.

[55] Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In Petri Mhnen, Klaus Pohl, and Thierry Priol, editors, *Towards a Service-Based Internet*, volume 5377 of *Lecture Notes in Computer Science*, pages 122–133. Springer Berlin Heidelberg, 2008.

[56] Stacey Higginbotham. Did facebook finally let its server count slip? *Gigaom.com*, 2013.

[57] T. Imada, M. Sato, Y. Hotta, and H. Kimura. Power management of distributed web savers by controlling server power state and traffic prediction for QoS. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008*, pages 1 –8, April 2008.

[58] W. Iqbal, M.N. Dailey, and D. Carrera. Black-box approach to capacity identification for multi-tier applications hosted on virtualized platforms. In *2011 International Conference on Cloud and Service Computing (CSC)*, pages 111 –117.

[59] Dragan Ivanovi, Manuel Carro, and Manuel Hermenegildo. Constraint-based runtime prediction of sla violations in service orchestrations. In Gerti Kappel, Zakaria Maamar, and HamidR. Motahari-Nezhad, editors, *Service-Oriented Computing*, volume 7084 of *Lecture Notes in Computer Science*, pages 62–76. Springer Berlin Heidelberg, 2011.

[60] Dragan Ivanovi, Manuel Carro, and Manuel Hermenegildo. Constraint-based runtime prediction of SLA violations in service orchestrations. In Gerti Kappel, Zakaria Maamar, and Hamid Motahari-Nezhad, editors, *Service-Oriented Computing*, volume 7084 of *Lecture Notes in Computer Science*, pages 62–76. Springer Berlin / Heidelberg, 2011.

[61] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long. Optimal cloud resource auto-scaling for web applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 58–65, 2013.

[62] Gueyoung Jung, M.A. Hiltunen, K.R. Joshi, R.D. Schlichting, and C. Pu. Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 62–73, 2010.

[63] R. Kaewpuang, S. Chaisiri, D. Niyato, Bu-Sung Lee, and Ping Wang. Adaptive power management for data center in smart grid environment. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 119–126, July 2012.

[64] Ioannis Kamitsos, Lachlan Andrew, Hongseok Kim, and Mung Chiang. Optimal sleep patterns for serving delay-tolerant jobs. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking*, e-Energy '10, page 3140, New York, NY, USA, 2010. ACM.

[65] Krishna Kant. Challenges in distributed energy adaptive computing. *SIGMETRICS Perform. Eval. Rev.*, 37(3):3–7, January 2010.

[66] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[67] Rouven Krebs, Christof Momm, and Samuel Kounev. Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 91–100, 2012.

[68] D. Kusic and Nagarajan Kandasamy. Risk-Aware Limited Lookahead Control for Dynamic Resource Provisioning in Enterprise Computing Systems. In *Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC)*, pages 74–83, 2006.

[69] Palden Lama and Xiaobo Zhou. AROMA: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)*, pages 63–72, 2012.

[70] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[71] YoungChoon Lee and AlbertY. Zomaya. Energy efficient utilization of resources in cloud computing systems. *The Journal of Supercomputing*, 60(2):268–280, 2012.

[72] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. Monitoring, prediction and prevention of sla violations in composite services. In *2010 IEEE International Conference on Web Services (ICWS)*, pages 369 –376, July 2010.

[73] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai, and Thomas Sandholm. What's inside the cloud? an architectural map of the cloud landscape. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 23–31, Washington, DC, USA, 2009. IEEE Computer Society.

[74] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance Model driven QoS Guarantees and Optimization in Clouds. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing (CLOUD)*, pages 15–22, 2009.

[75] Liang Liu, Hao Wang, Xue Liu, Xing Jin, Wen Bo He, Qing Bo Wang, and Ying Chen. Greencloud: A new architecture for green data center. In *Proceedings of the 6th International Conference Industry Session on Autonomic Computing and Communications Industry Session*, ICAC-INDST '09, pages 29–38, New York, NY, USA, 2009. ACM.

[76] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "Homogeneous" Warehouse-scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 619–630, 2013.

[77] M. Maurer, I. Brandic, and R. Sakellariou. Self-adaptive and resource-efficient sla enactment for cloud computing infrastructures. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 368–375, 2012.

[78] Michele Mazzucco. Towards Autonomic Service Provisioning Systems. In *Proceedings of the 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGRID)*, pages 273–282, 2010.

[79] Alexandre Mello Ferreira, Kyriakos Kritikos, and Barbara Pernici. Energy-aware design of service-based applications. In Luciano Baresi, Chi-Hung Chi, and Jun Suzuki, editors, *Service-Oriented Computing*, volume 5900 of *Lecture Notes in Computer Science*, pages 99–114. Springer Berlin Heidelberg, 2009.

[80] Massimiliano Menarini, Filippo Seracini, Xiang Zhang, Tajana Rosing, and Ingolf Kruger. Green web services: Improving energy efficiency in data centers via workload predictions. In *2013 2nd International Workshop on Green and Sustainable Software (GREENS)*, pages 8–15, 2013.

[81] D. Menasce. Two-level Iterative Queuing Modeling of Software Contention. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 267–276, 2002.

[82] D.A. Menasce and H. Gomaa. A Method for Design and Performance Modeling of Client/Server Systems. *IEEE Transactions on Software Engineering*, 26(11):1066–1085, 2000.

[83] Rich Miller. Apple Planning $1 Billion iDataCenter. *Data Center Knowledge*, 2009.

[84] Rich Miller. Report: Google uses about 900,000 servers. *Data Center Knowledge*, 2011.

[85] Rich Miller. Google Expansion Boosts Iowa Investment Past $1 Billion. *Data Center Knowledge*, 2012.

[86] Rich Miller. Ballmer: Microsoft has 1 million servers. *Data Center Knowledge*, 2013.

[87] Rich Miller. Microsoft Accelerates Its Data Center Expansion. *Data Center Knowledge*, 2013.

[88] Ramon Nou, Samuel Kounev, Ferran Julia, and Jordi Torres. Autonomic QoS Control in Enterprise Grid Environments using Online Simulation. *Journal of Systems and Software*, 82(3):486–502, 2009.

[89] Ehsan Pakbaznia and Massoud Pedram. Minimizing data center cooling and server power costs. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, pages 145–150, San Fancisco, CA, USA, 2009. ACM.

[90] Jrmy Philippe, Nol De Palma, Fabienne Boyer, and Olivier Gruber. Self-adapting service level in java enterprise edition. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '09, page 8:18:20, New York, NY, USA, 2009. Springer-Verlag New York, Inc. ACM ID: 1656991.

[91] V. Poladian, A. Arlan, M. Shaw, M. Satyanarayanan, R. Schmerl, and J. Sousat. Leveraging resource prediction for anticipatory dynamic configuration. In *First*

International Conference on Self-Adaptive and Self-Organizing Systems, 2007. SASO '07., pages 214 –223, July 2007.

[92] J.A. Rolia and K.C. Sevcik. The Method of Layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, 1995.

[93] Nilabja Roy. Qos assurance and control of large scale distributed component based systems, 2010. Copyright - Copyright ProQuest, UMI Dissertations Publishing 2010; Last updated - 2014-01-22; First page - n/a; M3: Ph.D.

[94] Alan Roytman, Aman Kansal, Sriram Govindan, Jie Liu, and Suman Nath. PAC-Man: Performance Aware Virtual Machine Consolidation. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC)*, pages 83–94, 2013.

[95] Ratnesh K. Sharma, Rocky Shih, Cullen Bash, Chandrakant Patel, Philip Varghese, Mohandas Mekanapurath, Sankaragopal Velayudhan, and Manu Kumar, V. On Building Next Generation Data Centers: Energy Flow in the Information Technology Stack. In *Proceedings of the 1st Bangalore Annual Compute Conference*, COMPUTE '08, pages 8:1–8:7, New York, NY, USA, 2008. ACM.

[96] Ratnesh K. Sharma, Rocky Shih, Cullen Bash, Chandrakant Patel, Philip Varghese, Mohandas Mekanapurath, Sankaragopal Velayudhan, and Manu Kumar, V. On building next generation data centers: energy flow in the information technology stack. In *Proceedings of the 1st Bangalore Annual Compute Conference*, COMPUTE '08, pages 8:1–8:7, New York, NY, USA, 2008. ACM.

[97] Upendra Sharma, Prashant Shenoy, and Donald F. Towsley. Provisioning Multi-tier Cloud Applications using Statistical Bounds on Sojourn Time. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)*, pages 43–52, 2012.

[98] Rahul Singh, Upendra Sharma, Emmanuel Cecchet, and Prashant Shenoy. Autonomic Mix-aware Provisioning for Non-stationary Data Center Workloads. In *Proceedings of the 7th International Conference on Autonomic Computing (ICAC)*, pages 21–30, 2010.

[99] Bill Snyder. Server virtualization has stalled, despite the hype. *InfoWorld*, 2010.

[100] Sebastiano Spicuglia, Mathias Björkqvist, Lydia Y. Chen, Giuseppe Serazzi, Walter Binder, and Evgenia Smirni. On Load Balancing: a Mix-aware Algorithm for Heterogeneous Systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 71–76, 2013.

[101] Touch Tohmatsu and CFO Research Services. The next wave of green IT. http://www.deloitte.com/assets/Dcom-UnitedStates/Local%20Assets/Documents/us_es_NextWaveofGreenIT.pdf.

[102] Davide Tosi, Giovanni Denaro, and Mauro Pezze. Towards autonomic service-oriented applications. *International Journal of Autonomic Computing*, 1(1):58–80, January 2009.

[103] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Automatic Computing (ICAC)*, pages 217–228, 2005.

[104] T. Vercauteren, P. Aggarwal, Xiaodong Wang, and Ta-Hsin Li. Hierarchical forecasting of web server workload using sequential monte carlo training. 55(4):1286 –1297.

[105] Daniel Villela, Prashant Pradhan, and Dan Rubenstein. Provisioning Servers in the Application Tier for e-commerce Systems. *ACM Transactions on Internet Technology*, 7(1), 2007.

[106] Long Wang, Rubing Duan, Xiaorong Li, Sifei Lu, T. Hung, R.N. Calheiros, and R. Buyya. An iterative optimization framework for adaptive workflow management in computational clouds. In *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 1049–1056, July 2013.

[107] Rui Wang and Nagarajan Kandasamy. On the Design of Decentralized Control Architectures for Workload Consolidation in Large-scale Server Clusters. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC)*, pages 115–124, 2012.

[108] Andreas S. Weigend. Analyzing customer behavior at amazon.com. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 5–5, New York, NY, USA, 2003. ACM.

[109] Linda Wilbanks. Green: My favorite color. *IT Professional*, 10(6):64, 63, 2008.

[110] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Sandpiper: Black-box and gray-box resource management for virtual machines. *Comput. Netw.*, 53(17):2923–2938, December 2009.

[111] Wei Zhang, Mingfa Zhu, Limin Xiao, Jiajun Liu, Qimeng Wu, Ying Song, and Yuzhong Sun. Autonomic resource allocation in virtualized data centers. In *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 192–198, July 2012.

[112] Xiaobo Zhou and Dennis Ippoliti. Resource Allocation Optimization for Quantitative Service Differentiation on Server Clusters. *Journal of Parallel and Distributed Computing*, 68(9):1250–1262, 2008.