# Engineering Model-Based Adaptive Software Systems

Cornel Barna

A Dissertation Submitted to the Faculty of Graduate Studies in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Graduate Program in Computer Science
York University
Toronto, Ontario

December 2015

# Abstract

Adaptive software systems are able to cope with changes in the environment by self-adjusting their structure and behavior. Robustness refers to the ability of the systems to deal with uncertainty, i.e. perturbations (e.g., Denial of Service attacks) or not-modeled system dynamics (e.g., independent cloud applications hosted on the same physical machine) that can affect the quality of the adaptation. To build robust adaptive systems we need models that accurately describe the managed system and methods for how to react to different types of change.

In this thesis we introduce techniques that will help an engineer design adaptive systems for web applications. We describe methods to accurately model web applications deployed in cloud in such a way that it accounts for cloud variability and to keep the model synchronized with the actual system at runtime. Using the model, we present methods to optimize the deployed architecture at design- and run-time, uncover bottlenecks and the workloads that saturate them, maintain the service level objective by changing the quantity of available resources (for regular operating conditions or during a Denial of Service attack). We validate the proposed contributions on experiments performed on Amazon EC2 and simulators.

The types of applications that benefit the most from our contributions are web-based information systems deployed in cloud.

# Contents

# List of Tables

# List of Figures

# Acknowledgements

I offer my gratitude to my supervisor, Dr. Marin Litoiu whose support throughout my Ph.D. and his assistance has been vital. I attribute the level of my Ph.D. degree to his encouragement and effort; without him, this thesis not have been completed.

I warmly thank other members of Center of Excellence for Research in Adaptive Systems. Special thanks to my fellow Ph.D. student Hamoun Ghanbari for his constructive comments during my Ph.D.

*I dedicate this thesis to my family who suported my efforts during my Ph.D:*

*to my parents, Sofia and Petru,*

*and to my brother, Andrei.*

# Co-Authorship

This thesis is based on the following papers (published in proceedings of conferences and journals):

[ 1 ] **Barna, Cornel**; Marin Litoiu; and Hamoun Ghanbari. "Autonomic load-testing framework." In Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC 2011), pp. 91-100. ACM, 2011. http://dx.doi.org/10.1145/1998582.1998598

[ 2 ] **Barna, Cornel**; Marin Litoiu; and Hamoun Ghanbari. "Model-based performance testing (NIER track)." In Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), pp. 872-875. ACM, 2011. http://dx.doi.org/10.1145/1985793.1985930

[ 3 ] **Barna, Cornel**; Mark Shtern; Michael Smit; Vassilios Tzerpos; Marin Litoiu, "Model-based adaptive DoS attack mitigation," ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012), pp.119,128, 4-5 June 2012. http://dx.doi.org/10.1109/SEAMS.2012.6224398

[ 4 ] Litoiu, Marin; and **Cornel Barna**. "A performance evaluation framework for web applications." Journal of Software: Evolution and Process 25, no. 8 (2013): 871-890. http://dx.doi.org/10.1002/smr.1563

[ 5 ] **Barna, Cornel**; Mark Shtern; Michael Smit; Vassilios Tzerpos; and Marin

Litoiu. “Mitigating DoS Attacks Using Performance Model-Driven Adaptive Algorithms,” Journal of ACM Transactions on Autonomous and Adaptive Systems 9 (TAAS 2014): 3. http://dx.doi.org/10.1145/2567926

[ 6 ] **Barna, Cornel**; Mark Shtern; Michael Smit; Hamoun Ghanbari; and Marin Litoiu. “Model-driven Elasticity DoS Attack Mitigation in Cloud Environments”. In 11th International Conference on Autonomic Computing (ICAC 2014). 2014.

[ 7 ] **Barna, Cornel**; Hamoun Ghanbari; Marin Litoiu; and Mark Shtern, “Hogna: A Platform for Self-Adaptive Applications in Cloud Environments,” ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), May 2015. http://dl.acm.org/citation.cfm?id=2821357.2821372

# Chapter 1

# Introduction

In the recent years there has been an explosion of applications and services that make use of the Internet in one way or another; either applications located somewhere online, which can be accessed by users through client software (e.g. a browser) or online services invoked remotely by the user's systems to enhance their functionality. In either case, applications and services share one important characteristic: they have a remote component hosted on an always-on system, that should be able to handle multiple clients at the same time. Depending on factors like type of application and its popularity, the number of clients can vary from a few hundreds or thousands per month (e.g. personal homepages) to milions per day (e.g. Google search engine).

The applications themselves also increased in complexity and often reach points where it becomes imposible for humans to manage them directly; either the skill requirement is too high and there are not enough people capable to do the job, or the application is very large with many components that puts it beyond human ability [1]. This complexity creates the need for better ways to manage an application, creates

the need for *adaptive systems* (or *autonomic systems*[1]).

An adaptive system is a system capable to function properly, within parameters, without human intervention. The system is capable to extract data from the environment where the web application resides (using a series of sensors), analyze it (identify problems that might prevent the application to function optimally or within parameters), create an adaptation plan (if necessary) and implement it. The web application and the resources it uses become the *managed resources*, while the rest of the system are part of the *application manager*.

The data extracted could include workload (e.g., the number of users getting service, the distribution of the users among different functionalities offered), utilization of various resources (e.g., CPU, memory, storage, bandwidth), the perceived user experience (e.g., response time for a request, requests that result in error) and various other metrics. The quantity of data can be very large, especially if we consider complex web applications where each component (software and hardware) is monitored separately. An adaptive system can start analyzing the data immediately when it becomes available; this way the problems can be discovered when they appear and a mitigation strategy can be deployed without delay.

To react to problems identified in data, the manager needs a good representation of the web application. It needs to know the deployed topology, the characteristics of each component, how these components interact and influence each other—it needs an accurate model of the entire application. On the model, the manager can evaluate different adaptation strategies in order to select best one (based on some goal specified by an administrator); it can test suppositions (e.g., what would happen if the workload will continue its current trend?); and it can use the model to uncover

[1] In this thesis we use the terms *autonomic systems* and *adaptive systems* interchangeably.

saturated bottlenecks. The accuracy of the model is the determining factor in the effectiveness of the manager.

Creating a model at the development time requires the developer to know the deployed topology, the resources that will be used and how these resources interact with each other, available networks and their characteristics. Also the developer needs to specify what services are provided by the web application and which resources are used by each service. Tuning the model (that can be done also at runtime) means setting the model parameters with newest measured/estimated values: demands for resources, quantity of the resources (clouds enable easy provisioning and decommission), network delays, workloads.

The capacity to *handle large quantities of data*, the ability to *use complex algorithms* to analyze the data in order to find problems and the *small reaction time* when the problems appear, make autonomic systems appealing solutions to manage complex systems.

Building adaptive systems is not a trivial task. The challenges start from the design phase of a web application when the architect has to choose from many possible architecture variants. That is a complex undertaking that requires deciding among the various software architecture styles, their implementation technologies, hardware and network layouts, economic and social factors and needs to consider all performance requirements [2]. During the lifetime of the application, the operating conditions can change, and the initial architecture needs to be updated. Each architecture variant has its own bottlenecks. When a bottleneck becomes saturated, the overall performance of the application suffers, so it's important to know *where the bottlenecks are and what type of workload will saturate them*; this information could be used in

deciding when and how an architecture needs to be updated.

Applications deployed in clouds are more challenging to manage than classic applications [3]. The difficulty derives from the multi-tenant nature of clouds, where on the same hardware can coexist multiple unrelated applications. Multiple applications running on the same hardware can influence each other despite the isolation that cloud provides (or promises to provide), especially when one of them executes a resource intensive task. This influence is perceived as a degradation in application's performance (and captured in the performance metrics), but the source of the extra work remains unknown. In these conditions, one important question that arrises with regards to modelling the application is *how we can handle the cloud variability*, so the model remains accurate and useful during the lifetime of the application.

Despite major progress in the field, there are still substantial challenges in designing and implementing self-adaptive systems. *What is the* ***optimal*** *combination of resources and their quantity that needs to be added/removed* as part of the adaptation strategy in the presence of a goal is such a challenge. For large systems with many resources, answering this question still proves to be elusive. Limited efforts have been invested in developing an engineering methodology and formal mathematical foundations, which makes the development and the verification of the self-adaptive systems tedious and time consuming. A systematic method would primarily enable the automation in designing self-adaptive systems, but also the effective verification of the automatic adaptation cycle. A design space methodology has been proposed by Brun et al. [132] to guide the designer along several dimensions, including identification, observation and control. However, this effort is subject to the peculiarities of individual interpretations and implementations.

Control theory has been proposed as a foundation formalism. The control theory is based on a formal model of the system, specified in a canonical form, from which a controller (or Autonomic Manager) is synthesized based on the goals of the system. Initial steps in using control theory as foundation for designing and implementing self-adaptive systems have been studied before [133, 132]. But existing approaches assume a *static linear model* of the system and a *static controller*. This static assumption limits the efficiency of the controller because software systems are highly dynamic and volatile; their models change at runtime and over time the controller may be based on the wrong assumptions. The design and implementation of adaptive systems become even harder for applications deployed in the cloud. The lack of transparency and control on the environment adds to the uncertainty of the models and impedes the design of the autonomic manager. As a result, most of the tools and frameworks available in industry for designing autonomic systems [134, 135, 136] are simple rule-based systems ("ON condition, DO action") that leaves the practitioner to do all the hard work in designing, implementing and verifying adaptive systems.

The popularity of web applications also means that the applications attract unwanted attention from malicious users. As more of the application's components are exposed on internet, the more vulnerable to attacks the application becomes because each component has its own weaknesses that can be exploited [4, 5]. Also, depending on the nature of the system, some components may be perceived as having less importance and their security weaknesses may be left unattended, thus allowing potential malicious users to use them as entry or attack points.

One type of attack that has seen an increase in both number of occurrences and severity is the (Distributed) Denial of Service attack [6, 7, 8]. The attacker

attempts to generate enough workload to saturate some resource, and render the web application unresponsive, thus denying service to the legitimate users. As a consequence, *protecting the web applications against Denial of Service attacks* is a major concern for systems administrators.

Mitigating (D)DoS attacks has proved to be very difficult. Internet protocols have been designed without built-in security features, relying on the good intentions of all entities. This "good behaviour" assumption became more and more obsolete as the Internet grew larger [9]. The lack of security in network protocols resulted in difficulties in discriminating between legitimate traffic and malicious traffic because the source of the attack cannot be reliably identified [5]. The availability of easy-to-use tools for generating (D)DoS attacks [10] and heterogenity of devices connected to Internet, each with its own weaknesses and vulnerabilities, only added to the complexity of defending against attacks [5].

We propose a model-based method for engineering adaptive systems for web applications deployed in clouds. The major contributions of this thesis are:

- a model-based method to explore the workload space in order to uncover and saturate the bottlenecks of a deployed web application;
- a performance model for web applications deployed in clouds capable to handle cloud variability;
- a robust adaptation architecture and method (Model Identification Adaptive Control) capable to synthesize a controller at runtime that will provide the adaptation strategy based on a goal;
- model-based adaptive architectures and algorithms focused on detecting DoS

attacks at the web application level and mitigating them appropriately;

The remainder of the document is structured as following: Chapter 2 introduces background concepts and reviews relevant work. In Chapter 3 we describe a model-based method, called Software Performance for Autonomic Computing (SPAC), that can make decisions regarding the selection of the best deployment architecture of a web application. SPAC can be also used at runtime, as part of an autonomic manager, to update the architecture of the web application in order to compensate for the changing operating and environment conditions.In Chapter 4 we present a method to test deployed web applications from a performance point of view, and uncover bottlenecks. The method makes use of performance models to guide the search of the workload mixes that saturate bottlenecks and then generates workloads against the deployed web application to fine-tune the results of the model.

In Chapter 5 we present a model-based adaptive architecture and algorithm for detecting Denial of Service attacks at the web application level and mitigating them. Using a performance model to predict the impact of arriving requests, a decision engine adaptively generates rules for filtering traffic and sending suspicious traffic for further review.The results from Chapter 5 are further extended in Chapter 6. In Chapter 6 we present a model-driven adaptive management mechanism which can correctly scale a web application deployed in cloud, mitigate a DoS attack, or both, based on an assessment of the business value of workload. This approach is enabled by modifying a layered queuing network model previously used to model data centers to also accurately predict short-term cloud behavior, despite cloud variability over time.

In Chapter 7 we show how to implement a model identification adaptive controller

(MIAC) using a combination of performance and control models. We show that our approach can account for uncertainty and modelling errors and efficiently adapt a cloud deployment.

# Chapter 2

# Background and Related Work

In this chapter we introduce the relevant concepts an existing work along three axis: the fundaments of building autonomic systems are presented in section 2.1; in section 2.2 we present concepts about modelling web applications; and section 2.3 discusses Denial of Service attacks.

## 2.1 Autonomic Systems

Autonomic systems are systems capable to function properly within parameters specified by Service Level Objective (SLO), without human intervention. These are systems that are self-configuring, self-optimizing, self-healing and self-protecting (for short, *self-** systems). The term of *autonomic computing* was introduced by IBM [53] and is used to describe such systems.

The four aspects of a self-managing system, as they have been identified in [54], are:

- *Self-Configuration.* Autonomic systems should be able to configure themselves

based on some high-level requirements (that specify the desired result, and not how it is accomplished [54]).

- *Self-Optimization.* Autonomic systems should be able to fine-tune their own parameters. They should monitor their performance, experiment with different values and learn optimal ones. Also, it is expected that these systems exist in a dynamic environment, so the optimization should be a continous process.

- *Self-Healing.* Autonomic systems should be able to detect by themselves failures in hardware and software, diagnose the root source of such failures and ultimatelly fix it. They could use advanced methods, algorithms and models in their diagnosis efforts; coupled with a knowledge-base, a fix-plan should be created and automatically implemented.

- *Self-Protection.* Autonomic systems should be able to protect themselves from outside threats or failures from their own components.

Systems that contain these elements, at least to some dregree, exist for some time. For example, the Windows operating system is capable to detect that core files have been modified (maybe as a result of a virus activity) and then restores them to the original version, thus providing self-healing. Streaming services like Netflix and YouTube can detect network congestion and adapt the bitrate accordingly, increasing or decresing the quality of the video the users see; thus incorporating self-optimization.

Architecturally, autonomic systems use the well-known Monitor-Analyze-Plan-Execute (MAPE-k) loop sugested by IBM [53]. The loop has four major components (see Figure 2.1) plus a knowledge base that stores information about the system, used during the loop execution.

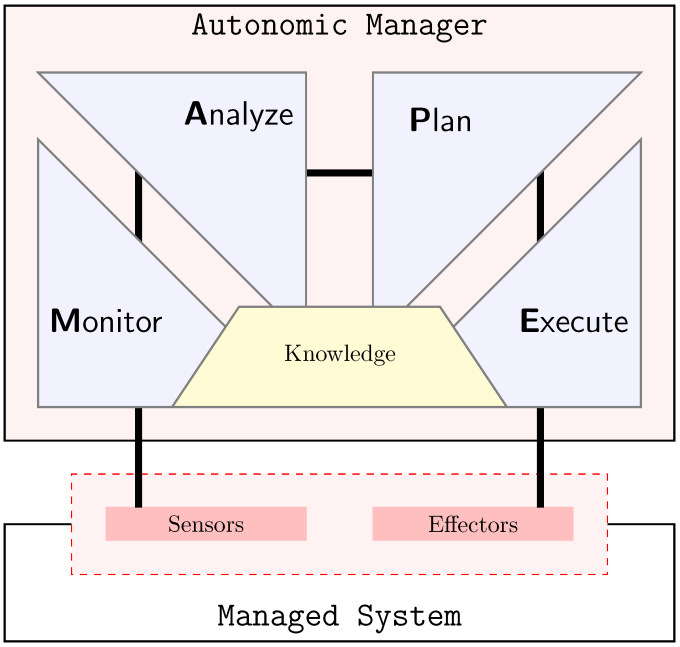



**Figure 2.1:** *The MAPE loop and it major components.*

The MAPE-k loop is part of the management subsystem; the management is responsible with trigeering loop iterations, integrate MAPE-k components with eachother and marshaling data between them. The managed element can be any type of resource (hardware or software).

The *sensors* are responsible to extract data about the system. Usually, each sensor is responsible for data regarding one aspect of the system (or multiple aspects that are closely related).

*Monitor*: This component is responsible for extracting information about the system from the sensors. The data must be relevant to the adaptation strategy that is employed and could include *CPU utilization*, *response times*, *arrival rate*, *throughput*, *memory*, *disk utilization*, etc. In order to obtain such data, the sensors must already be deployed on the managed system; these can be standard ones like

those offered by SNMP [55], JMX, OS Performance Counters or can be custom sensors, developed specifically for the monitored system. The monitoring component can also do aggregation and consolidation of the metrics that come from multiple sensors. This is specifically important for cloud-deployed systems that use multiple virtual machines, geographically-spread on a large area.

*Analyzer*: The *analyzer* will receive the consolidated metrics from the *monitor* and apply algorithms to evaluate the health of the system. This component is responsible to determine if some corrective measures need to be taken to respond to changes in the environment and prevent the system to break its Service Level Objective (SLO). Also, optimization algorithms can decide that there is room for improvement. By using some forecasting algorithms for the workload, the *analyzer* can detect which bottlenecks will be saturated.

*Planner*: The *planner* is responsible for creating a set of actions that need to be executed to solve the problems or to carry-on the optimizations identified by the *analyzer*. For a cloud application, the actions include provisioning/decomisioning resources (e.g. start/stop virtual machines), redirecting traffic, resize or migration of the virtual machines, etc.

*Executor*: The action plan created by the *planner* will be implemented by the *executor*. This component will handle all the details regarding communication with the cloud providers, credentials management, communication with custom effectors, reconfiguration of the software, etc. To implement the action plan, the executor makes use of the *effectors*—components not part of the autonomic manager that can execute specific actions. The effectors can be very diverse, just like the sensors, and are closely tied to the managed system. The capabilities of an effector can vary from

very simple like changing a parameter in a configuration file to more complex like starting a new virtual machine.

Monitoring a live system is challenging because it involves decisions like what type of data can be extracted and what data is useful, desigining and deploying custom sensors, how often the sensors should be queried, how to handle missing data, etc.

In [56], the authors identify two types of monitoring for autonomic systems:

- *Passive Monitoring* that can be done using already-available tools (e.g. `top` command in linux)
- *Active Monitoring* that require some modification of the monitored application code (e.g. injecting probes into compiled Java code)

Another classification for monitoring types can be found in [57]:

- *Continuous Monitoring* that continuously extracts metrics
- *Adaptive Monitoring* that collects data only about a few selected features; if problems are identified, the effort to collect more metrics is intensified and focused on the features that show anomalies.

In order to gather information, the sensors must interact with the system and use its resources. The monitoring component of the MAPE-k loop should take this overhead into account and minimize it. In [58] a monitoring system, called QMON, is introduced. QMON is capable to adapt the frequency of sensor's interogations, the quantity of data extracted based on some policies, aiming to minimize the overhead while preserving the utility of extracted data. As Huebscher et al. [56] call it, QMON is an autonomic monitor framework for autonomic systems.

In model-based adaptive systems, the planner uses a model that expresses the architecture and behaviour of the system. The planner component can select prebuilt alternative plans or generate them on-the-fly, and then, iteratively, test them on the model checking if the issue found by the analyzer is solved and ultimatelly select the best. The model used by the planner provides great advantages because, under the assumption that it is accurate, it can be used to check that the integrity of the system is preserved when applying the action plan [59].

Building a model for the system is not a trivial task. The person who builds the model needs to have an overall understanding of the system, its components and how they interact. The granularity of the model components is not restricted: it can be a virtual machine as a whole, an application running on the virtual machine, with some authors going as deep as modeling each individual java class of the application itself. The model designer must find the right balance between the level of details (more detailed model, more accurate results) and time needed to compute the model (more detailed model, more time necessary for computations).

## 2.2 Models

Web applications have become critical components in almost all business processes and services, private or public. Because most of the web applications follow a three-tier architecture, my work focuses on them.

In order to manage web applications, researchers have turned to models. A model is a representation of the architecture or the behaviour of an application, and can be used to make predictions, to test suppositions, or even to uncover errors and bugs (when the expected results of the application differ from the observed ones, it can be

because of bugs that exist in the application).

The researchers have used different types of models for web applications:

- *Analytical Models.* Analytical models capture the system structure and behaviour using mathematical equations. Their validity is established using mathematical tools (e.g. theorems and proofs). The best known analytical models for web applications are the *queuing network models.*

- *Empirical Models.* Empirical models are built from observations and rely only on the data. Their validity is asserted through experimentation and observation. The two major techniques used to build empirical models are *interpolation* (find a function that contains all the measured data points) and *model fitting* (find a function as close as possible to all the measured data points; in this category are include regression techniques).

- *Simulation Model.* The simulation models are computer programs that try to mimic the behaviour of a system [11]. They are used to gather metrics or observe the behaviour of the system in an artificial but close to reality experimental environment.

### 2.2.1 Regression Models

Regression analysis is a statistical technique useful in investigating the relationship between variables [12]. In their simplest form, regression methods attempt to find a function that matches, as close as possible, a set of observed datapoints. It is important to note that regression doesn't expose a causal relation between the parameters of the identified function and the result of the function. Regression simply states

that there is a correlation between the parameters and the values of the function, without saying anything about the nature of the corelation.

Linear regression is the simplest type of regression. It can be applied when there is a suspicion that an observed value depends linearly on one or multiple parameters. Formally, the linear regression problem is presented in Equation 2.1, where $x_i$ are the independent variables (also called *regressors*), $\beta_i$ are coefficients, and $y$ is the dependent variable (also called *response*). The assumption is that $y$ depends linearly on $x_i$.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n + \epsilon \tag{2.1}$$

In real systems, $y$ depends on other factors as well, possibly unknown, so it's not fully explained by the regressors. $\epsilon$ is designed to capture this error and noise; in linear regression the error is assumed to be random and with mean 0.

Building a linear regression model, means finding the right values for the coefficients $\beta$.

The main advantage of linear regression is its simplicity; it is the first method to be used in many problems. However, real-world problems are often complex and do not fit in a linear model. Also, linear regression is very sensitive to outliers (outliers are datapoints that are significantly different from other datapoints, and do not fit in a pattern), and gives bad predictions if the regressors are not really independent.

Linear regression creates a global model, that should fit the entire data-space. However, the relation between the regressor(s) and the response variable is often complicated, and is very hard (even impossible sometimes) to build a linear model that captures it over the entire domain. Breaking the domain into smaller divisions,

it is possible to build different models that have a good-enough precision in their respective division.

A regression tree uses a binary tree[1] as a main support data structure, in which the leaves represent divisions of the domain and each one has attached to it a simple predictive model. All other nodes (non-leaves) are decision nodes that have attached to them a boolean expression; when traversing the tree, if the boolean expression evaluates to *true* then the path should continue with the left node, else the path should continue to the right. The path ends with a leaf node; applying the model associated with it should provide the estimated value of the dependent variable. Thus, traversing a tree means making a guided search for the model that should be applied.

The main advantage of regression trees is that making predictions is very fast. The tree is usually easy to undestand (while a complex linear regressive model can be confusing). Also there are many algorithms in the literature, and implementations, that can build reliable trees.

### 2.2.2 Queueing Network Models

In queuing theory, the user's interaction with a hardware-software system is modeled using *classes of services* (or simply *classes* or *scenarios*[2]). A class of service is a service or a group of services that have similar statistical behavior and have similar requirements. When a user begins interacting with a service, a *user session* is created, and persists until the user logs out or becomes inactive[3]. The number of active users

[1] It is possible to create regression trees that are not binary, but it's fairly trivial to transform those trees into binary ones.

[2] In this thesis the terms *classes of service*, *classes* and *scenarios* are used interchangeably.

[3] In this thesis, a user has the following behaviour: makes a request to the web application, waits for the reply (response time), reads and processes the reply (assimilated to the think time between requests), then makes another request, etc.

at some moment $t$ is defined as $N$; these users can be distributed among different classes of services. The set of existing classes in the system is $\mathcal{C}$, and contains $m$ classes. The number of users that receive service from class $C \in \mathcal{C}$ is noted with $N_C$, thus $N = N_{C_1} + N_{C_2} + \cdots + N_{C_m}$. $N$ is also called *workload intensity* or *population* while combinations of $N_C$ are called *workload mixes* or *population mixes*.

Any software-hardware system can be described by two layers of queuing networks [13, 14]. The first layer models the software resource contention, and the second layer models the hardware contention.

Each resource has a *demand* (or *service time*, i.e. the time necessary for a single user to get service from that resource) for each *class*. The service times (demands) at the software layer are the *response times* of the hardware layer. Ideally, hardware demand is based on measured values; however, this is impractical for CPUs because of the overhead imposed by collecting such measurements.

Early work was done to analyze a system from a performance point of view in [15, 16, 17, 18]. In [19, 20] the authors investigated the influence of workload mixes on the performance of the system, how bottlenecks change with the workload mix and when they become saturated.

Balbo et al. [19] showed analytical relations between the workload mixes and utilization at the saturated bottlenecks as well as analytical expressions for asymptotic (with saturated resources) response times, throughput, and utilization within the saturation sectors. The results were presented for one queuing network layer consisting of hardware resources.

Early work in finding bounds on response time and throughput for one dimension of the workloads (one class) was done in [15, 16, 17, 18]. In [19] the authors showed

that in multiple workload mixes, multiple-resources systems, changes in workload mixes can change the system bottleneck; the points in the workload mix space where the bottlenecks change are called *crossover points*, and the sub-spaces for which the set of bottlenecks does not change are called *saturation sectors*. The same authors, in the same paper, showed analytical relations between the workload mixes and utilization at the saturated bottlenecks as well as analytical expressions for asymptotic (with saturated resources) response times, throughput, and utilization within the saturation sectors. The results were presented for one queuing network layer consisting of hardware resources.

The results from [19] were extended to non-asymptotic conditions (non-saturated resources) [20]. The authors used linear and non-linear programming methods for finding maximum object utilization across all workload mixes. That technique involved only the hardware bottlenecks.

There is no fully automatic method for building the structure of a performance model, however, there are available tools that can help in building a structure of the performance model [21]. Other papers, like [22, 23, 24], have shown how to build a tracking filter and a predictive QNM such that the model's outputs always match those of the real system. Performance parameters like the service time, think times, and the number of users can be accurately tracked and fed into a QNM.

Capacity planning of distributed and client-server software systems, particularly for web applications, is a common application area; a popular approach is using queuing models to model web applications at operational equilibrium [25, 26, 27] which has led to the automatic construction of measurement-based performance models [28, 29] or capacity calculators [30]. Others have tried to model the effect of

application and server tuning parameters on performance using statistical inference, hypothesis testing, and ranking (e.g. [31, 32]). Another approach automates the detection of potential performance regressions by applying statistics on regression testing repositories (such as Jiang in [33] and related earlier work). This has led to an approach for identifying subsystems that show performance deviations in load tests [34].

Tools have been developed to model and analyze a system from a performance point of view [35, 21, 36]. In [35], the authors present a tool designed to model *software* systems using layered queuing networks. The resources are grouped in layers and the requests move from layer to layer to get service. Once the model is solved, the output contains throughputs and utilizations for the software resources, distributions for the service time, queuing delays, etc.

## 2.3 Denial of Service attacks

**D**enial **o**f **S**ervice (DoS) attacks have increased in both volume and sophistication [8]. Attack targets include not only businesses and media outlets but also service providers such as DNS, Web portals, etc. A sophisticated DoS attack can be mounted by attackers without advanced technical skills. There are many advanced attacking toolkits freely available on the Internet [37], including LOIC (low-orbit ion cannon) [38]. DoS attacks are motivated by a variety of reasons (financial, political, ideological [39]), but regardless of motivation, thet have similar impact: lost revenue, increased expenses, lost customers, and reduced consumer trust.

Figure 2.2 shows a typical deployment architecture for a web application. There are multiple web servers (or application servers) that handle user requests; each user

is routed to a server by a load balancer. The web server interracts with the data tier (querying, inserting or updating data in a database), creates a reply and sends it back to the the user. Usually, the system is protected by a firewall that can filter the incoming/outgoing traffic based on some rules set by an administrator.

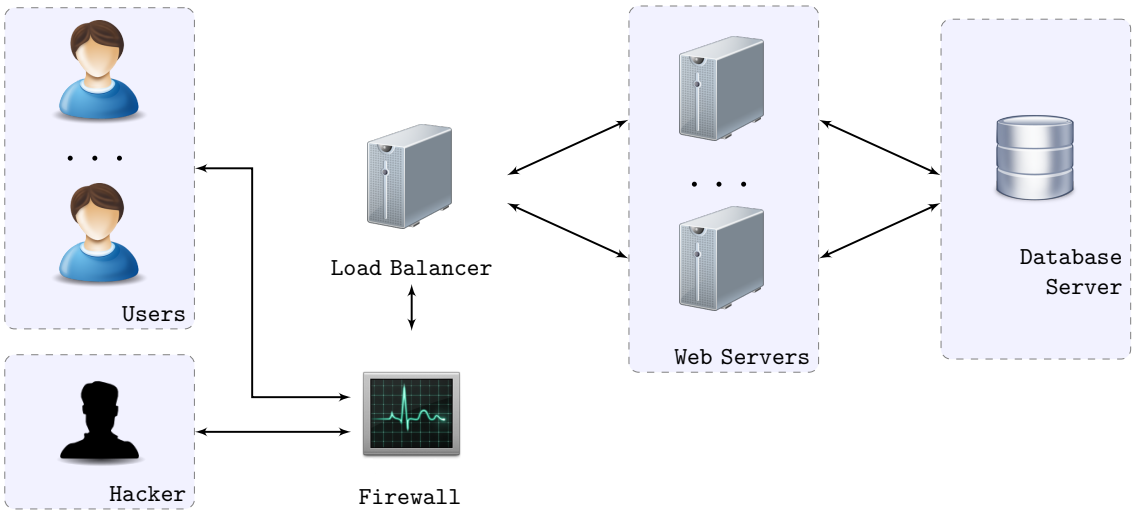



**Figure 2.2:** *A typical web application deployment architecture.*

The work and resources necessary to handle one request can vary greatly. For example, in case of a GPS application a user request to find the best route between two locations will require a query to the database for the data regarding the street layout in the area and the current traffic; then the data must be analyzed and a route has to be computed. Such a request will require work from the database server (extracting only the relevant data), network bandwidth (communication between the database and web servers is handled by an internal network with limited capacity) and also work on the web server (analyzing the data and computing the best route). On the other hand, a user request for the (static) home page of the same application will generate very little computation and no comunication with the data tier (hence

no use of bandwith on the internal network). A user will require the same amount of resources to generate either request, but that's not true for handling them on the targeted system.

An attacker's objective will be to identify what type of requests generate the most work, and then make enough requests so the targeted system becomes unresponsive, preventing it to offer service to legitimate users. When a system is under attack, it cannot accept new requests for service (and it simply rejects new users) or the response time becomes so long that the system is virtually paralyzed.

A DoS attacker will send many repeated requests that require resources to generate replies. These requests may be low-level TCP requests or higher-level application requests (like GET requests for web pages). The attacker discards the replies, meaning it takes fewer resources to send requests than it does to send responses (this problem is compounded when the target uses SSL; a recently released prototype tool demonstrates a dangerous type of SSL DoS attack [40]). Even with this beneficial ratio, the attacker may not be able to achieve denial of service with a single machine. Distributed Denial of Service attacks harness the power of many distributed attackers to attack a single target.

A *Distibuted* Denial of Service (DDoS) attack is a DoS type of attack where the malicious traffic comes from multiple sources. Behind the attack can be a single person that takes control over multiple computers or multiple people that synchronize their attacks against the same target. This synchronization can be achieved using emails, forum posts, and instant messages.

One of the main source of DDoS attacks is represented by botnets. A *botnet* is a network of compromized machines under the control of single individual (or a group

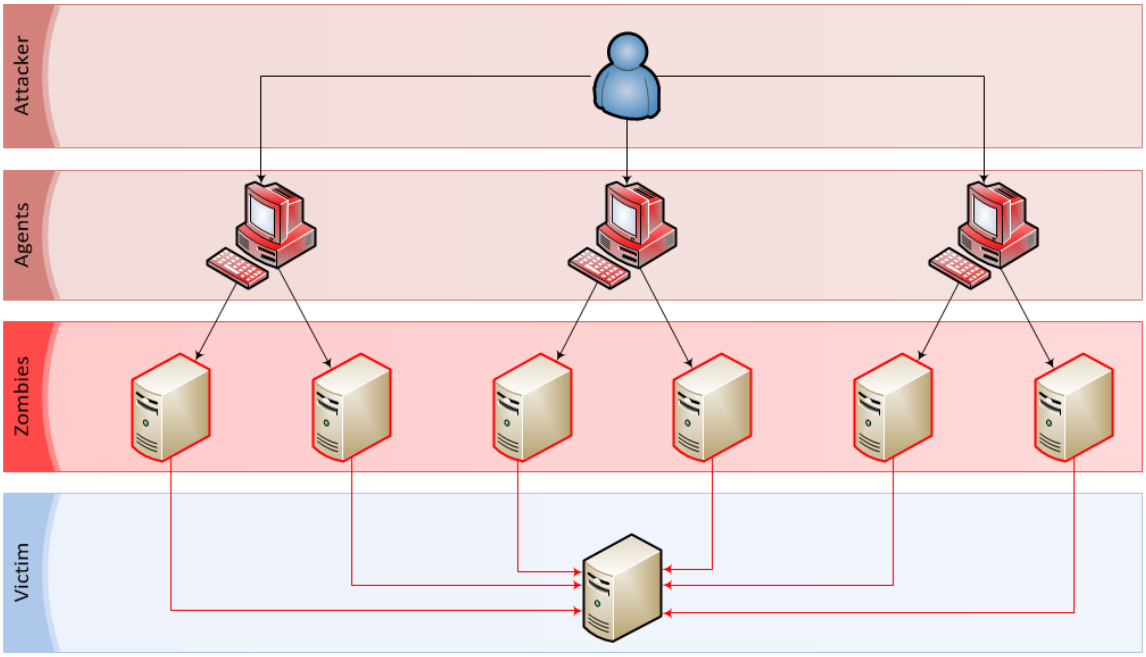



**Figure 2.3:** *Distributed Denial of Service*

of individuals), and is used for malicious purposes. Because of the large size, such networks can do a lot of damage and generate significant losses to companies that are targeted.

In case of DDoS, even a "small" botnet of 1000+ computers is enough to prevent the victim computer to offer its services. For an average of 128KBit/s upload bandwith (which, today, is available to most home computers), a botnet of 1000 computers can generate 100MBit/s in traffic; that is more than most businesses can handle (including corporations).

A typical setting of a DDoS attack is presented in Figure 2.3. The figure shows all entities involved, organized in different levels. On the top layer is the *attacker* that coordinates all the aspects of the attack. On the second level are the *agents* that act as the Command and Control for the "zombies" (the computers in the botnet

are "zombies"). An agent controlling a botnet, instructs the "zombies" to send the requests to the victim, but the agent does not directly send requests to the victim. On the third level are the "zombies"—these are the computers that directly generate the traffic in order to overwhelm the victim.

A big weakness of a (D)DoS attack, from the attacker's perspective, is that the source(s) can be identified once the attack has been discovered. Identifying the source can lead to deployment of mitigation techniques that filter the traffic coming from compromized computers/networks; but also exposes the attacker to legal consequences if law enforcement agencies are involved. A Reflective Denial of Service (RDoS) attack addresses these problems.

In a RDoS type of attack, innocent third parties are used to launch and amplify the attack. The attacker will spoof the source address in the TCP/IP packets with the address of the victim, and then sends them to a third party computer, called reflector[4]. The reflector will process the packet and then replies to the victim computer, because this is the address that the reflector sees as the source. The victim will experience the combined traffic from many reflectors, and will be unable to discriminate between "good packets" and "bad packets". Keeping the traffic low for each reflector prevents the reflector to notice it is being used in a DoS attack. Using a very large number of reflectors, the victim computer becomes overwhelmed and the diffuse nature of the traffic prevents it to identify the source. An attacker can easily add more reflectors or drop some of those in use, making very difficult for the victim to successfuly mitigate the situation.

A Distributed Reflector Denial of Service (DRDoS) attack combines the characteristics of both distributed attacks and reflective attacks. In this type of attacks the

[4]The reflector can be any server on internet that replies to the TCP/IP packets.

attacker is in control of a botnet and instructs each "zombie" to send traffic to the victim via third parties.

The distributed nature of DDoS (DRDoS), creates difficulties in mitigation. Creating rules to filter the traffic is not an easy task because the traffic comes from multiple IPs, located in different networks. A skilled attacker, in control of a botnet, could make his traffic look legitimate.

This has made some researchers claim that DDoS is not a security problem, but a scalability one [41]. Attackers will attempt to make their own requests indistinguishable from the rest of the traffic, and thus defeat the detection mechanism. When the discrimination between the "bad traffic" and "good traffic" cannot be made, the only method to ensure service is to increase the quantity of resources. But this is a very expensive solution, that puts it out of reach for the vast majority of companies that have an online presence. In [41] the author argues that, considering the current network architecture and protocols, the only sensible method to tackle DDoS is to improve detection, which makes attackers adapt their attack strategy, which leads to a *new* improved detection, and so on. He claims that the only way to break the cycle is to design a new network architecture that is resistent to distributed denial of service attacks. Any architecture that will treat DDoS as a *security* problem (as do the current architectures and protocols) will simply make the current cycle of "improve defense - improve attacks" to resurface.

Besides generating enough traffic to saturate some resource on the victim's side, the attacker can also take advantage of software bugs (in firmware and drivers for the network devices, in operating systems' implementation of various network protocols, in applications that handle network traffic), exploit vulnerabilities in the design of

network protocols and features, or simply exploit misconfigurations on victim systems.

DoS attacks which overload computer resources are known to be challenging to defend. Some experts argue the only possible solution for this problem is to improve security for all Internet hosts and prevent attackers from running DoS attacks [37]. An example of these source-end defence schemes is D-WARD [42]. Sachdeva et al. [43] identify a number of problems with source-end defence, principal among them doubt that such mechanisms will be widely implemented.

Researchers who agree with the challenge of defence at the source suggest defence at the victim site. For example, Kargl suggests using available DoS protection tools augmented with load monitoring tools to prevent clients (or attackers) from consuming too much bandwidth [37]. Other examples include QoS regulation [44] and cryptographic approaches [45]. Sachdeva et al. [43] also identified challenges when defending from the victim network, including the computational expense of filtering traffic, the possibility of the defence tools themselves being vulnerable to DDoS, and incorrectly dropping legitimate traffic. While a variety of other approaches have been suggested (e.g., [46, 47, 48, 49, 50]), the current state of the art does not fully mitigate DoS attacks [51, 52].

# Chapter 3

# A Performance Evaluation Framework for Web Applications

Web applications have become critical infrastructures for both public and private sectors running essential business processes and services. The topologies of such systems span across two or more tiers and rely on industrial middleware [60].

This chapter introduces a framework to compare and rank software architectures in the presence of complex performance requirements. The framework considers the interest of different stakeholders (such as architects, end users, and system administrators) and the capabilities of the candidate architectures in terms of performance and offers the decision-maker the necessary support to assess software architectures. The framework can be used for the whole life cycle of a Web application; however, the focus of this chapter is on the design and runtime phases. We use a rather broad definition of software architecture as being a representation of "the structures of the system which comprise the software components, the external visible properties of those components and the interactions between them" [61]. A software architecture

is a juncture for reasoning on how the system will fulfill some important functional and quality attributes.

The proposed framework, called Software Performance for Autonomic Computing (SPAC), takes as input the performance requirements, software architecture alternatives, a performance model of the system, and workloads. The output of the framework is a list of ranked architecture alternatives and is obtained by:

1. solving the performance models associated with each architecture;
2. matching the actual performance metrics obtained from the solved models against the performance requirements, and
3. aggregating the results of the matching.

The framework relies heavily on performance models. We introduce an architecture performance specification language, a solver, and an optimization technique that quantifies a software architecture in terms of the performance attributes. The performance of the architecture is measured across many use case scenarios and across multiple workloads. Any workload has two dimensions, the workload intensity $N$ (or the number of users in the system) and the workload mix (the combination of users in different scenarios).

This chapter extends the results from [62] in several ways (and this is where my contributions over the original work lie):

- we introduce the performance specification language and the performance solver OPERA
- we extend the framework to include support for runtime performance estimation

- we extend the case study to evaluate the runtime performance rating and ranking of runtime architectures.

Software Performance for Autonomic Computing offers a possible solution to the design of adaptive software systems that adapt to changes in the workload. There is an ongoing concern in the industry [1] that, with the increase in the number of software systems, the future will find us with a shortage of skilled workers to administer those systems. Therefore, the concept of autonomic computing is gaining momentum. New frameworks are needed to automate most of the administrative work; among those, heterogeneous workload management is essential. SPAC is also based on multicriteria decision-making, a theory that bases its outcomes on many attributes of the decision process [63].

The types of systems that can benefit from SPAC are Web-based systems, information-based and transaction-based software applications, such as e-commerce, insurance claim submission, Web banking, brokerage, and others. In these systems, users log-in, alternates requests with think times and then log-out. In terms of performance modeling, these systems are best described by closed performance models [64].

The remainder of this chapter is organized as follows: Section 3.1 describes the framework; Section 3.2 describes the performance tool used in our framework; Section 3.3 presents a case study; related work can be found in Section 3.4 and a summary and the conclusions are presented in Section 3.5.

## 3.1 SPAC

### 3.1.1 Overview

Figure 3.1a shows the main components of the SPAC framework.

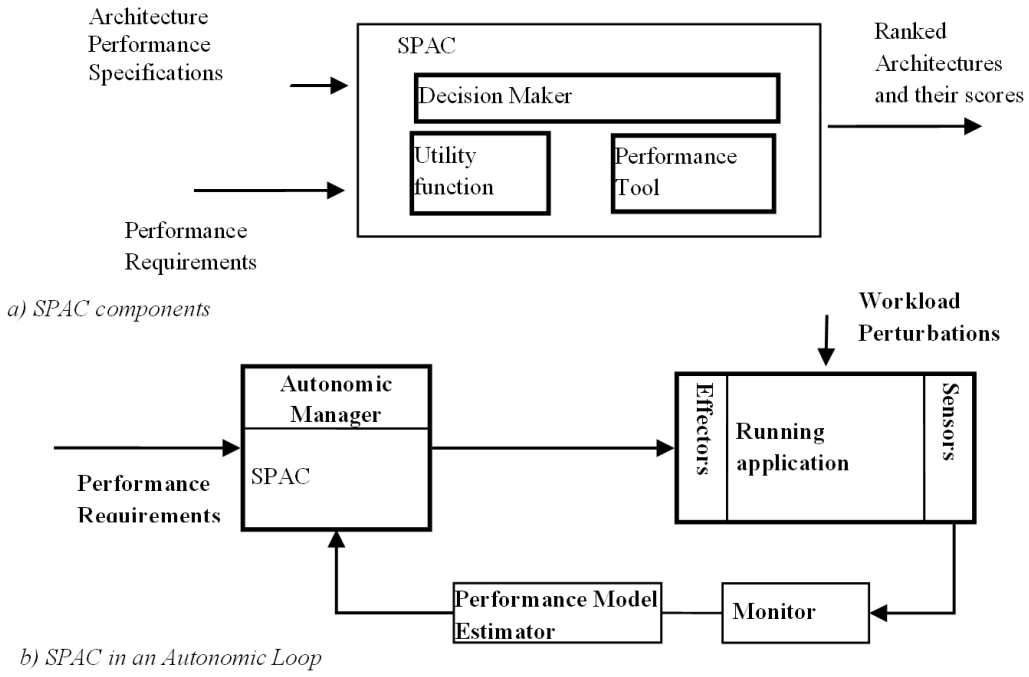



**Figure 3.1:** *SPAC components (a) and SPAC in a feedback loop for autonomic systems (b).*

*Architectures performance specifications* capture the performance characteristics of the software architectures. The performance characteristics are scenario based and capture the call chain of software components, the middleware and the hardware specifications. Section 3.2 will detail the language for performance specifications.

*Performance Requirements* are deduced by modeling the user perception with regard to the performance behavior of the system. What we try to optimize is not

an absolute value for response time but rather the user utility function.

*Utility Function* aggregates the performance of application scenarios in one scalar score such that the alternative architectures can be compared and ranked.

*Performance tool* computes the performance characteristics of the architectures under consideration. It parses the performance specifications and performance requirements and builds internal performance models and then solves them. *Performance models* are abstractions of the architectures and capture the performance characteristics in terms of input, output, queues, and waiting time in queues. The performance model structure is derived from the architecture specifications while the quantitative information is provided by the designer as the result of his/her previous experience or estimation. In the on-line use of the framework, data of the model can be derived from live monitoring of the deployed application. Delta architectures from the deployed one can have their approximate performance model obtained through corresponding changes to the deployed architecture model. A *Performance Model Estimator* can enhance the accuracy of the performance model by compensating for measurement and modeling errors.

The *Decision Maker* performs the Algorithm 3.1 to rank the architectures.

**Algorithm 3.1:** Decision Maker Algorithm

**input** : performance specifications of the architectures, performance requirements;
**output** : the list of ranked architectures and their scores.

1 Build a performance model for each architecture;
2 Using the performance model, evaluate the performance of each architecture;
3 Using a computed utility metric, rank all architectures;
4 Select the best architecture for deployment.

The output of the decision maker is an architecture that best satisfies the performance requirements. If there is no such architecture, a reiteration of the process is

triggered. An alternative option of the decision process is to provide a sorted list of alternative architectures.

There are three software lifecycle phases in which SPAC framework can be applied in the context of autonomic web systems:

1. *The development phase.* This phase analyzes candidate architectures; compares their performance outputs against the performance requirements. The output of this phase is an architecture that is implemented and deployed in production. The activities in this phase involve manual activities supported by tools.

2. *The runtime feedback loop* is illustrated in Figure 3.1b. In this loop, an Autonomic Manager analyzes the performance of the system, and based on a performance model changes the architecture of the system based on predefined algorithms. This loop is completely automated and the application is equipped with sensors and effectors as well as with a Monitor, a Performance Model Estimator and an Autonomic Manager.

3. *The evolution phase* connects the previous two phases. In this phase, information collected at run-time, such as performance traces and workloads, is used to enhance the design time information. Likewise, information from the design time, such as the structure of a performance model, is propagated to runtime, so that the runtime decisions are possible. This third phase involves manual activities as well and is used when moving to a new version of the application.

The presence of the three phases has the goal of building *homeostatic software systems* resilient to changes in workload. In general, a homeostatic system built by man or evolved by the nature [65], has an excess of capacity which allows it to function

normally when the environment changes are small. It is also equipped with feedback mechanisms that change its internal state to cope with big variations in environment. Applying SPAC at development time aims at building applications with an excess of capacity to accommodate a large number of workloads (but not all workloads) with minimal hardware infrastructure. At runtime, SPAC is used, for example, to optimize the underlying infrastructure, by provisioning and un-provisioning the application when the workload changes. Through provisioning, we add and remove replicas of the software components and eventually hardware and operating system support.

### 3.1.2 Performance Requirements and Utility Function

This section introduces a user utility function derived from a user perception of the performance behavior of the system.

Qualitative modeling of the user's perception of the system response times has been researched by the human-computer interaction community. For example, Geist and others [66] suggested the introduction of user perception of system response times in the design of computer systems. Human-computer interface (HCI) text books [67] address the user perception of the response times and offer corrective solutions such as "filling the interval". These corrective solutions are implemented with hourglass, incremental loading, hints and tips, etc.

Quantitative modeling of the user's perception of the Quality of Service (QoS) is extensively studied in Internet multimedia applications. Response time, jitters, throughput, or transmission rates can be quantified with utility functions that measure how they affect the user satisfaction.

In information and transaction Web systems, the quantitative modeling of the user

satisfaction with the performance is less studied. In this paper we use a performance utility function based on the following observations: the user satisfaction depends on performance of the task at hand and sometimes on the perceived workload.

User studies [68, 69] show that:

- Response times are expected to be appropriate to the *task* (or use case scenario[1]), varying from tens of milliseconds for mouse movement to the order of tens of seconds for more complex use case scenarios such as transactions. In other words, users expect some scenarios to be a lot faster than others;

- Faster is not always better, though; a shorter response time leads to shorter think time; therefore, the error rate increases with the rate of interaction.

A system faces a variety of workloads. Workloads are defined as the number of users that interact with the system (workload intensity) and the user distribution across different scenarios (workload mixes) or as per scenario arrival rate. Designers or system administrators distinguish sometime between high and low workloads levels [70], and aim at required response times for different workload intensities. These design decisions have to do with different deployments of the architectures as well as with user satisfaction. In the same time, user studies have shown [69] that users are more tolerant when they know there are more people in the systems (as happen during shopping hours or during winter holidays). Users are not that tolerant when they expect fewer people in the system. It is therefore justified to define performance requirements for different workload intensities.

[1] Although scenario has a broader meaning, in this context it can be assimilated to any web user action like browsing, searching, data entry, etc.

From the above observations we can conclude that the response time requirements should be on scenario and workload intensity basis. Also, we can conclude that the response time requirements per scenario $C \in \mathcal{C}$ should be expressed as an interval $\left[R^C_{Lo}, R^C_{Up}\right]$ where $0 \leq R^C_{Lo} < R^C_{Up}$ and $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ is the set of all scenarios. The lower bound $R^C_{Lo}$ should be seen as a preferred response time; the upper bound $R^C_{Up}$ as a maximum acceptable response time. For example, a performance requirement for a scenario $C \in \mathcal{C}$ may read like this: the response time should be greater than 0.2 s and less than 2 s.

The degree to which an architecture $A$ fulfills the performance requirements when $N_C$ users are executing the scenario $C$, can be computed with a utility function $Utility^{C,N_C}(A)$, which takes values in the range $[0, 1]$ such that:

- It is 1 when the actual response time is less than the preferred response time;
- It is 0 when the actual response time is higher than maximum acceptable response time;
- It is a strictly monotonic function that materializes the following rule: the closer the actual response time is to the preferred response time, the closer the value is to 1.

$$Utility^{C,N_C}(A) = \begin{cases} 1, & \text{if} \quad R^{C,N_c} \leq R^C_{Lo} \\ \frac{R^C_{Up} - R^{C,N_C}}{R^C_{Up} - R^C_{Lo}}, & \text{if} \quad R^C_{Lo} < R^{C,N_C} \leq R^C_{Up} \\ 0, & \text{otherwise} \end{cases} \tag{3.1}$$

An analytic expression of *Utility* is given by the equation 3.1, where $R^{C,N_C}$ is the actual response time for scenario $C$ when $N_C$ users are executing it and the utility

function is considered linear[2] between $R^C_{Lo}$ and $R^C_{Up}$.

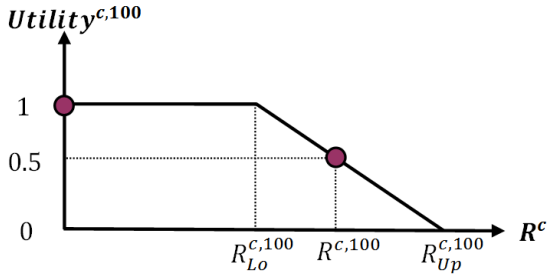



**Figure 3.2:** *User* Utility *per scenario C when $N_C$ users are executing scenario C, for an architecture A.*

A graphical representation of the utility function for an workload intensity of 100 users is shown in Figure 3.2, together with an example of actual response time $R^{C,100}$. The corresponding user utility in the light of this response time is 0.5.

### 3.1.3 Decision Maker

A decision maker matches the performance of architecture against the performance criteria. The performance of the architecture is assessed with a performance tool described later.

#### 3.1.3.1 Performance Criteria for Evaluating Alternative Architectures

In SPAC framework, workloads are criteria used to assess an architecture, and the number of workloads equals the number of criteria.

The architecture chosen at the development time has to achieve two main goals:

1. to accommodate many workload changes;

[2]It is expected that the user satisfaction is non-linear; for the simplicity of presentation, we approximate it with a linear function.

2. to allow reconfiguration at run-time (in other words, to allow for even more architectural changes at run-time).

In this section, we will explore the first goal, how we can choose a robust architecture.

In Section 3.1.2 we defined the user satisfaction with regard to a response time of a scenario for a workload $N$. Any workload has two dimensions, the workload intensity $N$ (or the number of users in the system) and the workload mix (the combination of users in different scenarios).

Formally,

$$N = N_1 + N_2 + \cdots + N_m \tag{3.2}$$

where $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ is the set of all scenarios offered by the web application and $N_i$ is the number of users executing scenario $C_i$, $\forall i = 1, \ldots, m$.

Any $N_1$, $N_2$, ... $N_m$ combination satisfying equation 3.2 represents a workload mix; workload mixes can also be expressed as ratios of users in every scenario and total number of users, $\beta_i = \frac{N_i}{N}, \forall i = 1 \ldots m$.

In combinatorial analysis terms, the number of mixes in equation 3.2 is a permutation that allows repetitions. If we consider the total number of customers $N$ as a string of $N$ 1s, and the scenarios as bins separated by $m-1$ 0s, a total of $N+m-1$ bits, then permutations of $N+m-1$ yields the total number of mixes:

$$\text{Number of workload mixes} = \binom{N+m-1}{m-1} \tag{3.3}$$

where $N$ is the workload intensity (total number of users accessing the application) and $m = |\mathcal{C}|$ is the total number of scenarios.

The direct consequence of the equation 3.3 is that there is not just one response

time for a workload intensity $N$, but a very large number, one for each workload mix. The number of workload mixes is huge and intractable when $N$ and $m$ are large. For example, for $N = 500$ and $m = 6$ there are $2.6 \times 10^{11}$ workload mixes. However, the maximum response time of each scenario over all workload mixes can be found in a reasonable amount of time when the system is described by a performance model [20]. Therefore, the meaning of the response time in the utility function 3.1 is that of the maximum response time across all mixes. Figure 3.3 shows the utility functions for scenario $C \in \mathcal{C}$ and two workload intensities, 100 and 500 users respectively. Although every workload mix will yield a different response time, we are interested in maximum response time over all mixes, namely $\max_\beta R^{C,N_C}$. The values of those metrics yield a utility function 0.1 for 100 users and 0.5 for 500 users.

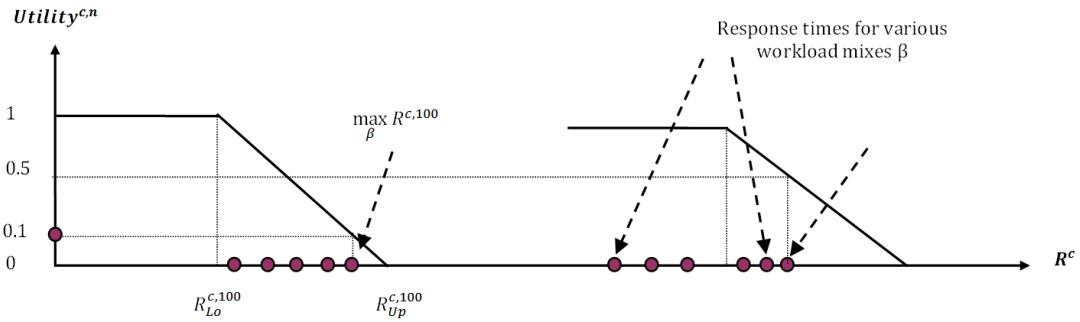



**Figure 3.3:** *Response times of scenario $C$ and utility functions for two workload levels.*

To add to the complexity, although every performance requirement is defined for an $N$, the workload intensity goes from 0 to $N$. The number of workloads implied by a performance requirement is given by the equation 3.4:

$$TotalWorkloads = \sum_{i=1}^{N} \binom{N+m-1}{m-1} \tag{3.4}$$

The decision to consider all workloads specified by 3.4 or just those present in the performance requirements lies with the stakeholders and depends on the specifics of the application. Workloads specified by 3.3 uncover the worst case response time and therefore their utility functions can be more pessimistic than the reality. However, a comprehensive characterization of the architecture's scalability is better achieved by considering the workloads in 3.4.

At runtime, the architectures are evaluated against individual workload mixes. While at the development time it is better to select an architecture which is robust to workload changes, therefore the architecture has a good performance over a large spectrum of workload mixes and intensities, at run-time the goal is quickly adapt to changes in the workload with minimum hardware requirements and with minimum overhead.

Although all workloads are important, therefore each criterion is important, there are situations when we should take into account the relative importance of the workloads in the life cycle of the system. In SPAC, each criterion may be weighted to reflect its importance or its probability of occurrence. Weights are expressed as normalized real numbers between 0 and 1 and the sum of the weights should be 1. The higher the weight, the more important is the criterion.

The need to weight workloads arises mainly from the Service Level Agreements between the stakeholders. It also can reflect the designer estimation of criteria importance or the measurement of workloads distribution on existent similar systems.

#### 3.1.3.2 Rating and ranking alternative architectures

Performance criteria are measurable by solving a performance model associated with the application. The use of the performance models (as opposed to measuring directly the application) is justified by the fact that the alternative architectures are too many to implement and because evaluating each alternative to find the maximum response time across all mixes requires less time. The degree in which each alternative satisfies each criterion is calculated by using the response time given by the model as input in the utility function 3.1. In the next section we look at a tool that estimates the response time of the application and then how to calculate a preference metric used to rank alternative architectures.

**Preference metric.** Another aspect to clarify is the aggregation of the weights, criteria, and utility functions. The goal of the aggregation is to characterize each alternative architecture by a unique measure, usually called *Preference*, which aggregates the weights, criteria, and utilities. There are several aggregation methods that stakeholders can use. The most used method in decision making is the *Simple Additive Weighting*. It computes the *Preference* of an alternative architecture $A_a$ as a weighted sum: each term of the sum is the product of the weight of the criterion (workload) $w_j$ and the utility of the alternative $A_a$ in criterion $j$, $Utility^{C,j}$.

$$P(A_a) = \sum_{C,j} w_j \times Utility^{C,j}(A_a); \quad a = 1 \dots A; \quad j = 1 \dots J \tag{3.5}$$

The above method assumes there is a compensatory effect of the utilities with regard to the user satisfaction. In other words, if an alternative architecture performs poorly in one workload, this can be compensated by a higher utility in other workloads.

When there is no compensatory effect, the stakeholders can use a non-

compensatory aggregation method such as *maxmin*. This method calculates the *Preference* of each architecture as the weakest or minimum of their utilities (min). The best alternative architecture is the one that has the maximum weakest utility (max). The logic is that a chain is as strong as its weakest link. Analytically:

$$P(A_a) = min_{j,C} \left( Utility^{C,j}(A_a) \right), \quad a = 1, \ldots, A; \quad j = 1, \ldots, J \tag{3.6}$$

The *maxmin* method 3.6 applies when criteria are all equally important. In the case of weighted criteria, *maxmin* uses a weighted utility:

$$P(A_a) = min_{j,C} \left( (1 - w_j) \times Utility^{C,j}(A_a) \right), \quad a = 1, \ldots, A; \quad j = 1, \ldots, J$$

The use of *Simple Additive Weighting* or *maxmin* depends on the class of the software architecture being built and on the working of the performance requirements within the Service Level Agreements (SLA). *Simple Additive Weighting* gives the average performance behavior across all workloads, while the *maxmin* points to the worst case scenario. The two methods can also be used in combination: use 3.6 to eliminate all the architectures that have a *Preference* lower than a threshold, and then use *Simple Additive Weighting* to rank the remaining architectures.

## 3.2 Performance Tool

To support SPAC we needed a versatile performance tool. We developed OPERA [36], "Optimization, Performance Evaluation and Resource Allocation" tool that can be used to assess the performance of information and transaction based web applications

and feed the utility function 3.1. OPERA has a performance specification language, PXL, and a corresponding model solver. In OPERA, the application logic is decoupled from the middleware and from the deployment topology. Software components have multiplicity levels (to count for threading and replicas) for horizontal or vertical scaling and interact within scenario through synchronous and asynchronous calls. Software components are deployed on hardware topologies made of nodes interconnected by networks. Nodes have multiplicity levels to accommodate for clusters and multiple-CPUs units and allow for easy reconfigurable topologies.

The tool models the application as a layer of two queuing network models, a software and a hardware layer and a solver iterates between the two layers until a fix point solution is reached.

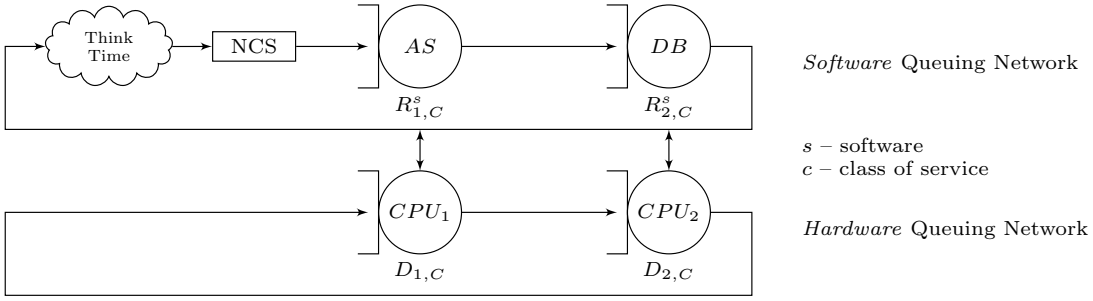



**Figure 3.4:** *Software and hardware layers in a two tier web system*

To illustrate the OPERA idea, let us consider a two tier system that is running web applications (see Figure 3.4). The system has a web application software server (AS) and a database software server (DB), each one running on dedicated hardware (for simplicity reasons, we consider in this example only $CPU_1$ and $CPU_2$ as hardware resources; for the "real" system other resources can be taken into account: memory, disk, etc.). Also, let's consider that we have an online store application with two

scenarios (or classes of service[3]):

- browse – the user is browsing through the available items in the store;
- buy – the user decides to buy some items and add them in the shopping cart.

The system can be modeled with two queuing networks, one for the software layer and another one for the hardware layer. The software layer has two queuing centers (AS and DB) which queue requests when there are no more threads available, non-critical sections where there are no queuing delays and a Think Time center used to model the user think time between requests. Again, for simplicity of the example, we didn't consider other possible queuing centers like *semaphores*, *critical sections*, etc.

Each resource has a demand associated, which is the time necessary to handle a request at the resource, in a class of service. In our example, the demands for the two CPUs for requests in class $C \in browse, buy$ are $D_{1,C}$ and $D_{2,C}$. At the software layer the service times are the *response times* of the hardware layer ($R^s_{1,C}$ and $R^s_{2,C}$), which include the demand and waiting time at the hardware resources (we use the superscript $s$ to denote metrics that belong in the software layer).

The demands at the hardware layer can be measured or estimated (using Kalman filters or particle filters). However, this cannot be done for the software layer. For the software layer OPERA solves the model for the hardware layer, extracts the response time and uses it as the demand.

After each iteration (one iteration means solving both layers once) OPERA checks if the model is stable (the model is said to be stable if in two consecutive iterations

[3]A *class of service* is a scenario or a group of scenarios that have similar statistical behavior and similar requirements. In this thesis we will use the term *class* to refer to a class of service.

the queues at the resources change less than a given value). If the model is not stable, the users will be redistributed in the two layers and a new iteration will start.

Outputs of the tool include: mean response time, utilization of resources and throughput for each scenario for individual workload mixes, or maximum utilization and response time across all workload mixes and for each scenario.

A formal definition of the model and algorithms used by one layer of the tool can be found in [71].

### 3.2.1 Performance Specifications

OPERA allows to model the architecture of the system using the Performance Extensible Language (PXL). OPERA accepts as input a PXL file (which is a XML document) which contains four sections:

- Deployment *topology*: describes the topology of the system; the nodes, clusters, containers, middleware and networks—how they connect with each other and what are their performance characteristics;
- Usage *scenarios* (classes): describes what happens when the users execute some action (like click on the submit or search button on a web page);
- *Workloads*: describes some points of interests from the application (like the number of users or important mixes);
- *Performance requirements*: describes some expectations from the application (like response time).

By splitting the model in separate components gives flexibility in evaluating different architectures. For example, a change in the deployment topology won't

affect the scenarios, etc. We detail each component of the model in the remainder of the section.

**Deployment Topology**: The topology denotes the nodes and the network hosting the application, the clusters, containers and the middleware. Figure 3.5 shows the topology of the system presented earlier.

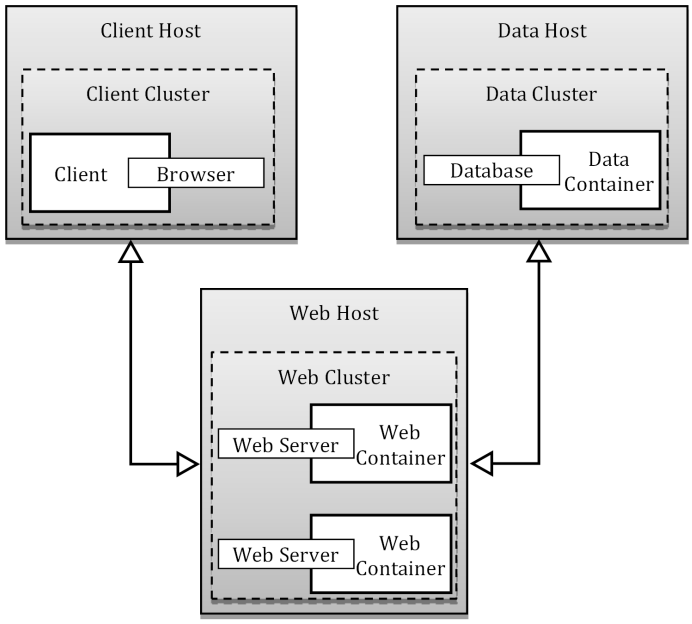



**Figure 3.5:** *A topology example.*

All services run in containers that are grouped in clusters which reside in nodes. The nodes are connected by a network.

In the PXL file, each element (node, cluster, container, network) is identified by a name, which will be used later, in the description of scenarios.

The nodes, which represent hosts, have two *ratio* attributes: for disk and CPU. These are real numbers that show the ratio between the CPU (disk) speeds of the host on which the demands of the services were collected and the CPU (disk) speeds

of this host. A ratio of 0.5 means that the CPU (disk) of the modeled system is two times faster than the CPU (disk) of the system where the measurement was done. The parallelism attributes are used to specify the multiplicity of the CPU (disk). Each node can be a client or a server, as specified by the attribute type.

The number of threads that a container can use is specified using the attribute parallelism for the tag container. If, for example, the web server can use 20 threads, we set the value for the attribute parallelism to 20 (for `WebContainer`), and, when the services are described, we specify that the "Web Server" runs in `WebContainer` (see Figure 3.5).

For the `Network` tag, the following attributes can be used:

- `connectsHosts` – the name of the hosts connected by the network. The names are separated by spaces;
- `latency` – the interval between the time a bit is sent and the time when this bit is received at destination;
- `overheadPerByte` – the time required to transmit a byte.

Listing 3.1 shows, in PXL syntax, the topology from Figure 3.5.

**Listing 3.1:** *The topology description, using PXL syntax.*

```
<Node CPUParallelism="1" CPURatio="1" DiskParallelism="1"
      DiskRatio="1" type="client" name="ClientHost"/>
<Node CPUParallelism="2" CPURatio="1" DiskParallelism="2"
      DiskRatio="1" type="server" name="WebHost"/>
<Node CPUParallelism="1" CPURatio="1" DiskParallelism="1"
      DiskRatio="1" type="server" name="DataHost"/>
```

```
<Cluster name="ClientCluster">
  <Container name="Client"
             canMigrate="false" parallelism="1000"
             runsOnNode="ClientHost" server="false"/>
</Cluster>
<Cluster name="WebCluster">
  <Container name="WebContainer"
             canMigrate="false" parallelism="20"
             runsOnNode="WebHost" server="true"/>
</Cluster>
<Cluster name="DataCluster">
<Container name="DataContainer"
           canMigrate="false" parallelism="150"
           runsOnNode="DataHost" server="true"/>
</Cluster>
<Middlware name="http"
           fixedOverheadReceive="0" overheadPerByteReceived="0"
           fixedOverheadSend="0" overheadPerByteSent="0"/>
<Network connectsNodes="ClientHost WebHost DataHost"
         latency="0" name="Internet" overheadPerByte="0"/>
```

**Scenarios**: Scenarios are triggered by user actions and denote traces through the application. Scenarios can be derived from Use Cases, from Class diagrams, as defined by UML, or by tracing the application using application profilers.

First the services will be described. This will connect the scenarios with the hosts that will execute them. For example, in Listing 3.2 we are specifying that the service `WebServer` runs inside the `WebContainer` (which has 20 threads) that resides inside `WebHost` (which can use two CPUs).

Each scenario is characterized by a name and set of calls for service (see Listing 3.2 for the description of the browse scenario introduced earlier). CPU and disk *demands* are specified as real numbers, and represent the time required for CPU and disk, respectively, to execute a single call. The number of bytes sent and received by the calling service can be also be specified.

In our example (Listing 3.2) the user that executes scenario `browse` will generate a request to the web server (the first call is from the `Browser` to the `WebServer`). The web server will interrogate the database to extract the items to be displayed on the web page (the second call from the `WebServer` to `Database`). In both these calls, the number of bytes transmitted over the network is ignored (both attributes have the value 0). A call can be synchronous or asynchronous and that is specified by the attribute `type`.

**Workload**: In this section of the PXL file the maximum number of users and the mixes of interest are specified. In the root tag, the `kind` attribute can be used to specify for which kind of workload the application is optimized. The possible values are:

- HL – High Population Level;
- ML – Medium Population Level;
- LL – Low Population level.

The tags that can be used in this section are:

- `Users` – the total number of users that the application should support. This number is used for finding the worst response time and highest utilization across all workload mixes;

- `WorkloadMixes` – defines the number of users for each scenario (the workload mixes are independent of the `Users` element described above). The workload mixes can consider the system as a closed or open model. When the system is considered as open, then the `openModel` attribute should be set to `true`. When the system is modeled as a closed model, then the `openModel` attribute is set to `false`. The value of the `openModel` attribute has implications on the meaning of the `Mix` elements;

- `Mix` – defines the load of a given scenario by setting the `load` attribute. When the system is modeled as an open system (`openModel="true"`) then the `load` attributes define the *arrival rate* in that scenario. When the system is modeled as a closed system (`openModel="false"`) then the `load` attributes define the number of users in that scenario. In this latter case, the number of users is complemented by the `ThinkTime` element;

- `ThinkTime` – Think times for each scenario. They denote the user idle time between two requests in milliseconds. These values are considered in tandem with the `Users` element defined above or with those `WorkloadMixes` that refer to closed models.

**Listing 3.2:** *Description of the scenario browse, using PXL syntax.*

```
<Services>
  <Service canMigrate="false" name="Browser"
           runsInContainer="Client"/>
  <Service canMigrate="false" name="WebServer"
           runsInContainer="WebContainer"/>
  <Service canMigrate="false" name="Database"
```

```
            runsInContainer="DataContainer"/>
  </Services>
  <Scenario name="browse" triggeredByService="Browser">
    <Call bytesReceived="0" bytesSent="0" callee="WebServer"
          caller="Browser" invocations="1" type="s">
      <Demand CPUDemand="41.3207" DiskDemand="0.3423" />
    </Call>
    <Call bytesReceived="0" bytesSent="0" callee="Database"
          caller="WebServer" invocations="1" type="s">
      <Demand CPUDemand="11.7812" DiskDemand="1.2432" />
    </Call>
  </Scenario>
```

Listing 3.3 shows an example of the workloads section for the system introduced earlier. We are interested in how the system will behave when there are 100 concurrent users; 70 of them execute the scenario `browse` and 30 execute `buy` (note that this is not an open model). The average think time for the first scenario is 3000 milliseconds and for the second is 1000 milliseconds.

**Listing 3.3:** *Workload for scenarios browse and buy.*

```
<Workloads kind="ML">
  <Users>100</Users>
  <WorkloadMixes openModel="false">
    <Mix load="70" scenario="browse"/>
    <Mix load="30" scenario="buy"/>
  </WorkloadMixes>
  <ThinkTimes>
    <ThinkTime time="3000" scenario="browse"/>
    <ThinkTime time="1000" scenario="buy"/>
```

```
    </ThinkTimes>
  </Workloads>
```

**Requirements**: The requirements section allows users to add to OPERA some expectations for the system. Currently only response time is accepted. For each scenario, a lower (`minResponseTime`) and an upper (`maxResponseTime`) value of the targeted response time can be specified. The values are real numbers, greater than zero. The tool will optimize the distribution of the services that can migrate so it reaches the lower value for each scenario. An example can be seen in Listing 3.4.

**Listing 3.4:** *The requirements section of a PXL file.*

```
<Requirements>
  <ResponseTime maxResponseTime="100000" minResponseTime="100"
                scenario="browse"/>
  <ResponseTime maxResponseTime="100000" minResponseTime="100"
                scenario="buy"/>
</Requirements>
```

### 3.2.2 Discussion: How OPERA supports different architecture types

The specification language PXL and the OPERA tool support the design, deployment and runtime architectures [72], as follows. The XML elements `Services` and `Scenarios` allow the designer to define the architecture components and connections, respectively. The granularity of the service is at the latitude of the architect, and it can range from a programming language class to an entire application. `Calls`

are generic connections; they can be synchronous or asynchronous method calls or message passing. Well known qualitative performance principles such as locality, parallelism, and sharing [73] can be used to define alternative architectures. Other qualitative principles such as modifiability, security, can lead to even more architectures. Choosing one architecture over another is a decision that can be based on SPAC.

In the implementation and deployment phase of the software system life cycle, in the process of capacity planning, when it comes to architecture decisions, the architects have to decide, among other things, if the scalability can be better accomplished by a *vertical* or a *horizontal* architecture [2]. In this phase, multiple replicas of software entities are created, on different computers (*horizontal*) or on the same computer (*vertical*). PXL supports that by providing containers and multiplicity levels for those containers. The `Middleware` and `Clusters` XML elements of PXL allow the architect to define multiple deployment architectures.

At run time, there are still predefined architectures to choose from when adapting the system to face an increase in workload or to adapt to new business needs. Self-tuning, self-balancing and self-provisioning [1] are 3 facets of autonomic computing, which explore many software architectures before deciding which one satisfies the user requirements. A model based self-provisioning architecture which includes SPAC is shown in Figure 3.1. An Autonomic Manager monitors the performance metrics of an application and the workload through a Monitor. With the collected data, a performance model is tuned by a Model Estimator. Based on the prediction of the model, different alternatives are evaluated using SPAC and the application is provisioned by adding or removing instances of replicable software and hardware components.

PXL support for autonomic run time changes is exemplified by multiplicity levels on nodes and clusters.

## 3.3 Case Study

The purpose of this section is two-fold: (a) to illustrate the application of SPAC framework and the associated tool OPERA in selecting the architectures at development and runtime and (b) to validate that the proposed framework selects the most appropriate architecture with regard to performance requirements and architecture characteristics. The section assumes that the performance models of architectures are accurate. Proving that accurate models can indeed be built is beyond the scope of this chapter; they have been covered in [74, 75], and will further be covered in Chapter 4.

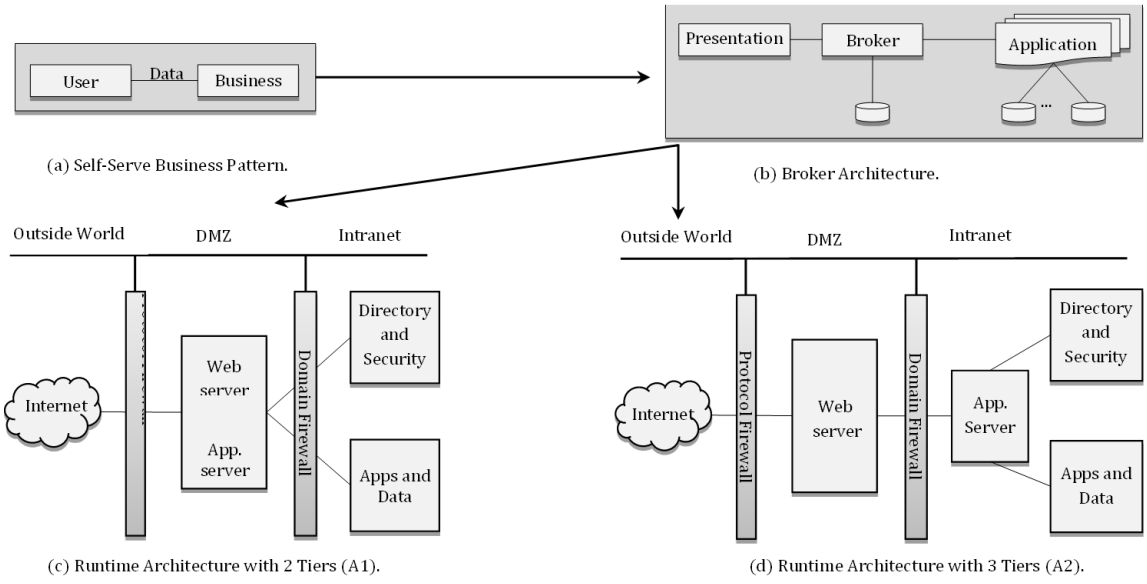



**Figure 3.6:** *Two runtime architectures for Broker Application Pattern.*

At development time, we consider two architectures: A1 and A2 (Figure 3.6). They were obtained following the industrial practice presented in [72]: we consider a *self-serve* business pattern, one of its corresponding application patterns, namely the *Broker* application pattern, and further on, two of the possible architectures. A self-serve business pattern (Figure 3.6a) establishes a channel of communication between a user and a business. Examples of applications that fit this pattern include insurance claim submission, Web banking, brokerage, and others. Application patterns represent the partitioning of the application logic and data together with the styles of interaction between the logic tiers. In the broker architecture shown in Figure 3.6b, the *Broker* component determines the nature of the client (or protocol) the request is coming from and then dispatches the request to the corresponding *Application*. The communication between the Broker and Application components is synchronous. The Broker Architecture is further realized by two deployment architectures, A1 and A2, shown in Figure 3.6c and Figure 3.6d. In the architecture A1, both Web and Application servers are between the two firewalls (Demilitarized Zone, DMZ), while in the second architecture A2, only the Web server is in the DMZ, the Application server being placed within the company domain.

The performance requirements are as follows: the application is supposed to support up to 1000 users concurrently and three scenarios. The performance requirements are specified for three workload intensities: 250, 500 and 1000 users. The maximum response times for scenarios 1 and 3, for the three workload intensities and across all workload mixes, have to be within the intervals [200, 400], [400, 600] and [1000, 6000] ms, respectively. The lower and upper values of the intervals correspond to the $R^c_{Lo}$ and $R^c_{Up}$ described in Section 2.1 For the scenario 2, the maximum

response times for the three workload intensities have to be within the intervals [300, 500], [500, 700] and [1000, 6000] ms, respectively.

### 3.3.1 Development-time decisions

Figure 3.7 shows the response time (in milliseconds) for the two architectures, when users execute the three scenarios. Using OPERA tool, seven workload mixes were found as being the worst and we used them to find how architectures A1 and A2 would behave under load. With red lines (with no decoration) is the response time when architecture A1 was used, and with blue lines (decorated with squares) when A2 was used. Solid, dashed and dotted lines represent scenarios 1, 2 and 3, respectively. The response times are average values.

From the plots, it can be seen that A1 has lower response time when the workload intensity is low. Increasing the number of users, we notice that the performance of A1 deteriorates fast, while architecture A2 is able to service users and keeping the response time very low. Through a visual inspection of the graphs, we can conclude that A2 is a better deployment architecture if we care more about higher workloads. SPAC method should select A2 as well.

SPAC results are explained next. Table 3.1 shows the actual response times of the two architectures A1 and A2, in three scenarios of requests, when deployed on the same hardware topology. The results were obtained with performance tool OPERA. The specification file PXL has the structure exemplified in Section 3.2 and the data obtained in an off line identification phase. The three columns groups correspond to the three workload intensities and show the maximum response times across all workload mixes for each scenario 1, 2 and 3. The architecture A1 has better

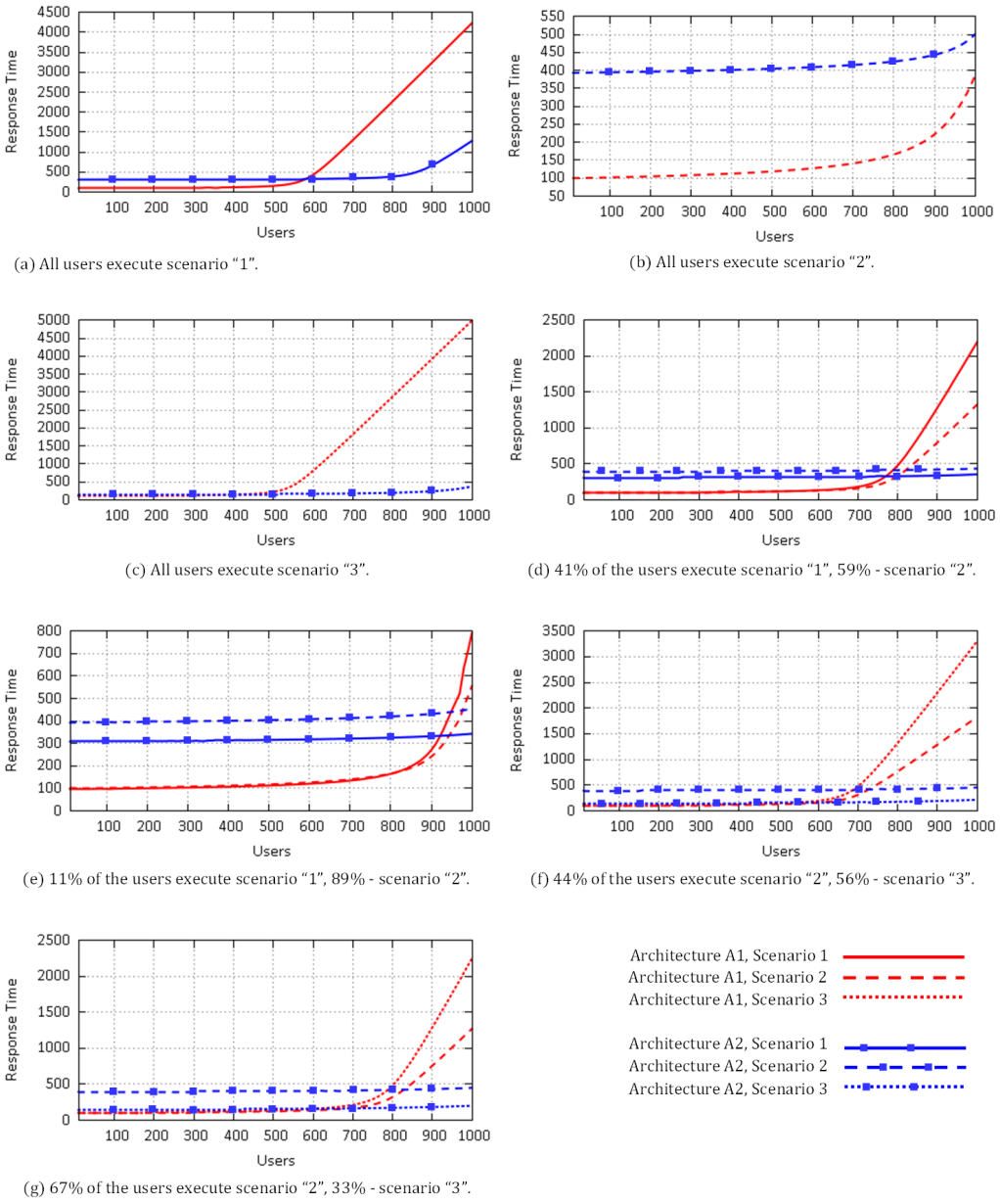



**Figure 3.7:** *The response times for the worst mixes.*

response times at lower workloads (250 users), because the communication overhead of A2 between the Web server and the application server is having an impact on the overall response. At higher loads (1000 users), the communication overhead becomes

insignificant compared with the queuing delays, and therefore the architecture A2 provides better response times.

| N | 250 | | | 500 | | | 1000 | | |
|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| A1 | 105 | 106 | 117 | 154 | 119 | 221 | 4231 | 385 | 4990 |
| A2 | 312 | 397 | 148 | 320 | 404 | 159 | 1282 | 500 | 374 |

**Table 3.1:** *Per scenario response times (in milliseconds) for two* design architectures *and three workloads.*

Table 3.2 shows the utility functions computed with the relation 3.1 for the performance requirements of this particular case study. The workloads are weighted to count for their relative importance. The sum of the weights is 1 and the weights are distributed across the three workloads. Since for this particular application the scalability is very important, the highest workloads have higher weights.

| N | 250 | | | 500 | | | 1000 | | |
|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| Weight | 0.10 | 0.10 | 0.10 | 0.30 | 0.30 | 0.30 | 0.60 | 0.60 | 0.60 |
| A1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 0.35 | 1.00 | 0.20 |
| A2 | 0.44 | 0.51 | 1.00 | 1.00 | 1.00 | 1.00 | 0.94 | 1.00 | 1.00 |

**Table 3.2:** *Utility functions for two* design time *architectures and three workloads.*

The preferences for A1 and A2, calculated by using the Simple Additive Weighted given by equation 3.5, are 2.13 and 2.86, respectively, which tells us quantitatively, that architecture A2 is better than A1.

If we consider the *maxmin* aggregation given by 3.6, we find that the preferences are 0.20 and 0.44 respectively (they are given by the workload 1000 and 250 users, respectively). Based on this quantitative measure, the architecture A2 is better than architecture A1.

Given that both metrics above recommend the architecture A2, the conclusion is clear: A2 is the architecture of choice to be implemented and this matches the conclusion drawn by visual inspecting the performance of the two architectures (Figure 3.7).

Architecture A2 will guarantee a good response time across a large range of workloads. However, if the workload intensity goes above 1000 users, then the response time might deteriorate. In the reminder of the section, we show how SPAC can help the provisioning decisions at runtime.

### 3.3.2 Runtime decisions

For runtime experiments, we used the framework shown in Figure 3.8 which is the implementation of the design from Figure 3.1. The autonomic manager will use OPERA as the modeling tool. The monitor component of the framework will use JMX [76] and SNMP [55] agents to extract relevant metrics from the distributed system. Using these metrics, the autonomic manager will update the model. To eliminate the noise from the measured metrics, we used Kalman Filters [77].

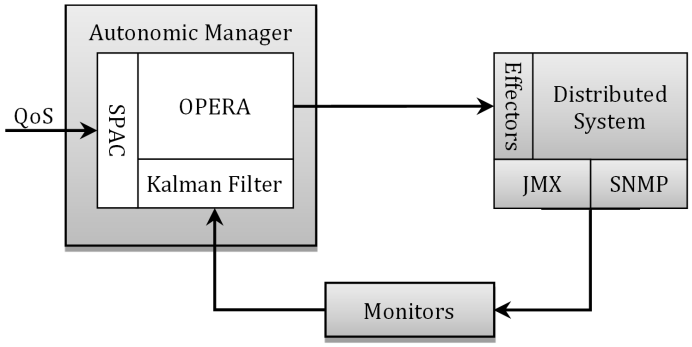



**Figure 3.8:** *The framework used in experiments.*

At workload intensities just above 1000 users, the response time deteriorates due

to the saturated bottlenecks at application and data servers, respectively. To cope with this increase in the workload, in a usual provisioning scenario, a provisioning autonomic manager will add replicas to the application server or to data server or to both. Table 3.3 and Table 3.4 show two runtime architectures (A21 and A22), their response times and the utility functions. Three predicted workload mixes are considered. The mixes are (250, 250, 1000), (250, 1000, 250) and (1000, 250, 250). They have the same response time requirements for high workloads, [1000 6000] ms. The response time, at runtime, are calculated for individual workload mixes.

| N | 250 | 250 | 1000 | 250 | 1000 | 250 | 1000 | 250 | 250 |
|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| A21 | 607 | 1267 | 1443 | 754 | 1686 | 944 | 356 | 445 | 205 |
| A22 | 3479 | 2183 | 2821 | 1798 | 1251 | 1414 | 4104 | 2512 | 3314 |

**Table 3.3:** *Per scenario response times for two* runtime architectures *and three workload-mixes.*

| N | 250 | 250 | 1000 | 250 | 1000 | 250 | 1000 | 250 | 250 |
|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| Weight | 0.10 | 0.10 | 0.10 | 0.30 | 0.30 | 0.30 | 0.60 | 0.60 | 0.60 |
| A21 | 1.00 | 0.95 | 1.00 | 1.00 | **0.86** | 1.00 | 1.00 | 1.00 | 1.00 |
| A22 | 0.50 | 0.76 | 0.64 | 0.84 | 0.95 | 0.92 | **0.38** | 0.70 | 0.54 |

**Table 3.4:** *Utility functions for two* runtime architectures *and three workload mixes.*

A21 and A22 are variations of the architecture A2 depicted in Figure 3.6d: A21 has 2 replicas of the Application Server while A22 has 2 replicas of the Data Server. The assumption is that the Autonomic Manager only has one computer available that it can provision either Application Server or Data Server.

The preferences for A21 and A22, calculated by using the Simple Additive Weighted given by Equation 3.5, are 2.9 and 1.9. The *maxmin* values for the two

architectures are 0.86 and 0.38 respectively. Both metrics suggest architecture A21 is by far the best choice.

Next, we wanted to see what are the worst mixes, in terms of response time, when the number of users is 1500. We used the OPERA tool to find the bottlenecks in architectures A21 and A22 and then we computed the workload mixes.

OPERA found that there are seven bottlenecks which give us the workload mixes: (1500, 0, 0), (0, 1500, 0), (0, 0, 1500), (614, 886, 0), (170, 1330, 0) (0, 667, 833) and (0, 1000, 500). The response time for each is shown in Table 3.5. In Table 3.6 are shown the utility functions when the response time requirements is set to [1000, 6000].

| N | 1500 | 0 | 0 | 0 | 1500 | 0 | 0 | 0 | 1500 | 614 | 886 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| A21 | 350 | - | - | - | 2986 | - | - | - | 1782 | 450 | 788 | - |
| A22 | 4672 | - | - | - | 553 | - | - | - | 2934 | 2389 | 1560 | - |

| N | 170 | 1330 | 0 | 0 | 667 | 833 | 0 | 1000 | 500 |
|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| A21 | 986 | 2412 | - | - | 1352 | 741 | - | 2065 | 1155 |
| A22 | 711 | 669 | - | - | 1511 | 1816 | - | 1079 | 1161 |

**Table 3.5:** *Response times for 1500 users on two runtime architectures.*

We notice that when all of the users are executing scenario 1, A22 has very poor results, compared with A21 (see first workload mix in Table 3.5). Also, A22 has poor results when users favor scenario 3; but if users start to shift toward scenario 2, the performance of A22 improves until provides better results than A21 (see workload mixes 2 and 5 in Table 3.5).

The preferences calculated for A21 and A22 architectures, using Simple Additive Weight and equal weight for all mixes, are 1.40 and 1.34 respectively, which makes

| N | 1500 | 0 | 0 | 0 | 1500 | 0 | 0 | 0 | 1500 | 614 | 886 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| A21 | 1 | - | - | - | 0.60 | - | - | - | 0.84 | 1 | 1 | - |
| A22 | 0.27 | - | - | - | 1 | - | - | - | 0.61 | 0.72 | 0.89 | - |

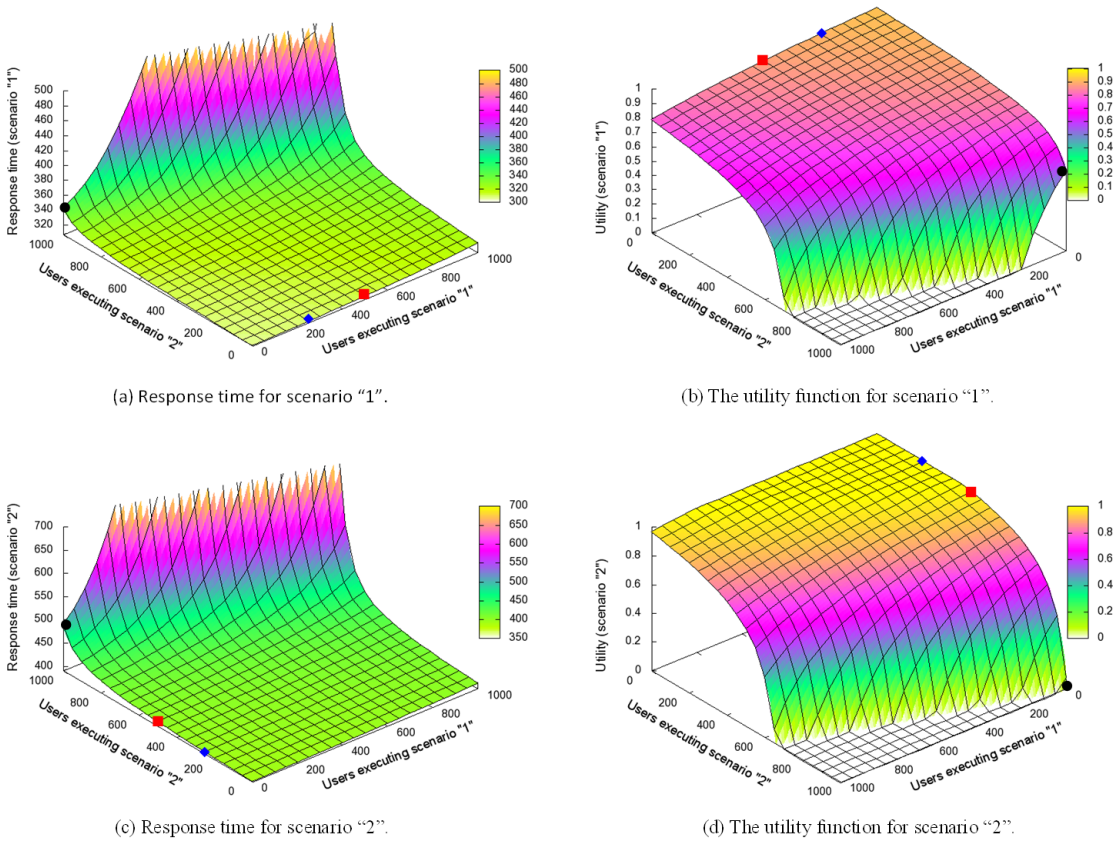
| N | 170 | 1330 | 0 | 0 | 667 | 833 | 0 | 1000 | 500 |
|---|---|---|---|---|---|---|---|---|---|
| Scenario | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| A21 | 1 | 0.72 | - | - | 0.93 | 1 | - | 0.79 | 0.97 |
| A22 | 1 | 1 | - | - | 0.90 | 0.84 | - | 0.98 | 0.97 |

**Table 3.6:** *Utility functions for 1500 users on two runtime architectures.*

A21 better than A22. However, by analyzing the usage pattern of the system and switching between the architectures at runtime the performance of the system can be improved.

Figure 3.9 shows the response times and utility functions for architecture A21 when there are only two scenarios. The X0Y plane represents the workload mix for those scenarios, while on the 0Z axis is the response time (Figure 3.9a and Figure 3.9c) and the utility function (Figure 3.9b and Figure 3.9d). In subfigure (a) and (c) it can be seen that when the number of users increases, the response time will increase (for displaying purposes, we put a limit on the response time shown). To compute the utility, we require that the response time be in interval [300, 400] for the first scenario and [400, 500] for the second one. Subfigures (b) and (d) contain the plot of the utility, and can be seen that when the response time increases, the utility decreases (note that the X0Y plane is rotated 180 degrees in (b) and (d)).

For a workload intensity of 250 users, for architecture A21, the model found that the highest response time for scenario 1 is generated by the mix $\langle 250, 0 \rangle$ (shown with a blue diamond in Figure 3.9a). Similarly, mixes $\langle 500, 0 \rangle$ and $\langle 1, 999 \rangle$ have the highest response time for workload intensities of 500 and 1000 users (shown in Figure 3.9a




(a) Response time for scenario "1".

(b) The utility function for scenario "1".

(c) Response time for scenario "2".

(d) The utility function for scenario "2".

**Figure 3.9:** *Response times and utility functions for architecture A21 when there are only two scenarios.*

with a red square and a black dot, respectively).

For scenario 2, mixes $\langle 0, 250 \rangle$, $\langle 0, 500 \rangle$ and $\langle 0, 1000 \rangle$ have the highest response time (see the blue diamond, red square and black dot in Figure 3.9c).

Similarly, Figure 3.9b and Figure 3.9d have these workload mixes marked with blue diamonds, red squares and black dots.

## 3.4 Related Work

There are several interrelated methodologies—all proposed by the Software Engineering Institute—for analyzing and ordering architectures according to quality attributes and stakeholders' assessments. Software Architecture Analysis Method (SAAM) [61] defines key architecture quality attributes such as performance, modifiability, and availability. Architecture Tradeoff Analysis Method (ATAM) [78] takes SAAM further and offers a methodology on how to assess the implications of architectural decisions with regard to SAAM's attributes. ATAM's declared goal is not to offer quantitative methods for assessing the quality attributes; it rather focuses on the methodology of eliciting stakeholders' assessments of the attributes for *use case*, *growth* or *exploratory scenarios*. More recently, Cost Benefit Analysis Method [79, 80] offers an economic method to quantify architectures in terms of benefit per dollar. Our proposed framework is quantitative and although it uses economic decision methods, it applies to performance. Also, our method looks at complex performance requirements.

Deciding the architecture of a distributed system has been a design decision so far. However, recent research has been done in architecture changes at runtime. For example, in [81], Kramer and Magee proposed a layered reference architecture for self-managed systems. The architecture has been inspired by robotics application; nevertheless, it is broadly applicable. The architecture has three layers: Component Control, Change Management, and Goal Management. *Component Control* supports the ability to add, remove, and reconnect components. A review of the current state of the art in runtime adaptation is presented in [82]. Previous work in self-managing systems highlights planning and policies as the two main adaptation mechanisms. Policies describe how the systems should be modified when certain conditions have

been met (a system that uses policies to make architectural changes is presented in [83]) and planning involves generation of a sequence of actions that need to be executed in order to achieve a goal. Sykes et al. present in [84] a three layer model that uses planning for adaptable software architecture. SPAC proposes a quantitative methodology that can be applied to one control level.

There are other authors that make use of models to when a change in architecture of a system is required. A very complex model is [85]; the authors propose a model that has full information about all the components (java classes) in the application. This approach might prove to be too complex and unfeasible to implement, even though the accuracy of it is high.

Some authors propose a multi-model framework [86, 87, 88, 89], each model designed to capture different characteristics of the underlying system, like how it behaves under low workloads and high workloads. Unlike these approaches, we work with a dynamic model that is changed and updated every time new data from the monitored system becomes available. This way, our framework is capable to capture characteristics of the system under heavy load and low load. Also, by capturing utilization scenarios in the model, we manage to obtain high accurate performance data that can be used to make architectural change decisions.

## 3.5 Conclusions

We presented a framework for analyzing and evaluating the performance of software architectures. The framework begins with identifying the performance requirements of the software systems. These requirements have to account for different scenarios, upper and lower values of the response times, and the multitude of workloads a

software system might face.

A utility function was also defined. It takes into account the upper and lower values of the required response time and the actual response time, and calculates the user satisfaction as a value between 0 and 1. The workloads and the associated performance requirements are criteria for evaluating the software architecture. The actual response time of that architecture, obtained by solving a performance model, is matched against the performance requirements by the utility function, and tells how well the criteria are satisfied. By weighting the criteria to reflect their importance in the life cycle of the software architecture, an aggregated utility function is computed across all criteria. Two or more software architectures can therefore be compared by using these aggregated utilities.

Two aggregation methods inspired by economic models were used to calculate a unique score for each architecture: Simple Additive Weighting and Maxmin. Simple Additive Weighting gives the average performance behavior across all workloads, while the Maxmin points to the worst case behavior.

To respond well to heterogeneous workloads, an application should work like a homeostatic system: to be able to function well within a range of perturbation and, when the perturbations are out of range, to deploy feedback mechanisms to adapt to the perturbations. Following the above principles, SPAC can be used to deploy a software architecture that respond well to a variety of workloads. At runtime, when the workloads are exceeding the expectation, autonomic managers can pull more resources from a pool and adjust the capacity of the application accordingly. We showed trough a case study how the framework can applied.

Threats to the validity of the approach includes inaccurate performance models

and inaccurate performance requirements. The first refer to inadequate structure and data for the model while the latter to incomplete Service Level Agreements. In that case, the deployment architecture might not be the best choice; however the second level of adaptation, the runtime autonomic control can scale the application based on accurate models.

# Chapter 4

# Autonomic Load-Testing Framework

Performance testing is fundamental in assessing the performance of software components as well as of an entire software system. A major goal of performance testing is to uncover functional and performance problems under load and the root cause of those problems. Functional problems are often bugs, deadlocks and memory management bugs, buffer overflows. Performance problems often refer to high response time or low throughput under load.

In practice, the testing is done under operational conditions, that is, the testing is typically based on the expected usage of the system once is deployed and on the expected workload. The workload consists of the types of usage scenarios and the rate of these scenarios. A performance test usually lasts for several hours or even a few days and only tests a limited number of workloads. The major drawback of this approach is that expected usage and scenario rates are hard to predict. As a result, many workloads that the system will face remain uncovered by the stress test.

In this chapter we propose an autonomic framework that explores the workload space and searches for points in this space that cause the worst case behaviour for software and hardware components of the system. It is generally known that the performance of a software system is influenced by the *hardware* and *software bottlenecks*. Bottlenecks are those resources where the requests are queued and delayed because the processing capacity limits of that resource. When those limits are reached, we say that the bottlenecks are *saturated*. Consider for example a web based application in which a web server has 100 threads available. When there are more than 100 pending requests, the server is a saturated bottleneck because it has reached its capacity. If the requests keep coming, they will be buffered in a waiting queue that will eventually reach its limits as well. In a software system, there are many bottlenecks and, more importantly, those bottlenecks change as the workload changes.

Finding the workloads that cause the bottlenecks to change is a challenging but rewarding problem. We propose an autonomic load stress testing framework that drives the workloads towards the points that create the bottlenecks and eventually saturate them. We also show that the software performance metrics reach their maximum or minimum for those workloads that cause some bottlenecks to reach their capacity or policy limits. The method uses an analytical representation of the software system, a two-layer queuing model that captures the hardware and software contention for resources. The model is automatically tuned, using on-line estimators that find the model parameters. The overall testing method is autonomic, based on a feedback loop that *generates workloads* according to the outputs of the model, *monitors* the software system under test, extract metrics, *analyzes* the effects of each

workload and *plans* the new workloads based on the results of the analysis.

The type of software systems that would benefit the most from the proposed method are web based transactional systems. To model the interaction of users with such systems, we define *classes of services*, or *classes* in short. A *class* is a service or a group of services that have similar statistical behavior and have similar requirements. When a user begins interacting with a service, a *user session* is created. The session will be maintained active until the user logs out or when he is inactive for a specified period of time. If we define $N$ as the number of active users at some moment $t$, these users can use different classes of services. If we have $C$ classes and $N_c$ is the number of users in class $C$, then $N = N_1 + N_2 + \cdots + N_C$. $N$ is also called *workload intensity* or *population* while combinations of $N_c$ are called *workload mixes* or *population mixes*.

The main contributions of the work presented in this chapter are:

- we present an adaptive framework for performance testing of transactional systems
- we present a method to explore the workload space in order to uncover and saturate the bottlenecks
- we design a case study to evaluate the validity of the proposed method.

The remainder of the paper is organized as follows. Section 4.1 presents the theoretical foundations of the testing method. Section 4.2 describes the general testing framework and introduces the testing algorithm. A case study is presented in Section 4.3, related work is described in Section 4.4, and the conclusions are presented in Section 4.5.

## 4.1 Performance Stress Space

This section introduces the *stress space*, defined as the multidimensional domain that can be covered by the software performance metrics. To start with, a software-hardware system can be described by two layers of queuing networks [13, 14]. The first layer models the software resource contention, and the second layer models the hardware contention.

To illustrate the idea, consider a web based system with two tiers, an application software server (AS) and database software (DB) server (see Figure 3.4). Each server runs on its dedicated hardware, $CPU_1$ and $CPU_2$ computers respectively. Consider also that we have two classes of service. The hardware layer can be seen as a queuing network with two queues (for simplicity of presentation, we only consider the CPUs of the servers in this example) and with the *demands* (or *service times*) for class $C$ being $D_{1,C}$ and $D_{2,C}$ respectively, $C \in \{1, 2\}$. The software layer has two queuing centres, the processes AS and DB, which queue requests whenever there are no threads available. (Besides queuing for threads, the requests can queue for *critical sections*, *semaphores*, etc. Those can be represented as queuing centres as well.)

The software layer also has non-critical sections (NCS) where there are no queuing delays and a Think Time centre that models the user think time between requests. The service times (demands) at the software layer are the *response times* of the hardware layers. In our case they are $R^s_{1,c}$ and $R^s_{2,c}$, and they include the demand and the waiting time at the hardware layer (we use the upper script $s$ to denote software metrics that belong to the software layer).

### 4.1.1 Utilization Constraints

In multiuser, transactional systems, a hardware bottleneck is a device (CPU, disk, network) that has the potential to saturate if enough users access the system. In general, the device with the highest demand is the bottleneck. However, when there are many classes of requests with different demands at each device, then the situation becomes more complex. When the workload mix changes, the bottleneck in the system can change as well and there may be many simultaneous bottlenecks in the system at a given time.

To find all the hardware bottlenecks in the system, let's assume that the workload intensity is high enough to make the potential bottlenecks saturate. Workload mixes yield per class utilization at each resource; the sum of per class utilizations equals the total utilization of that resource. Total utilization of resource $K_i$ is a linear function of per class utilizations and has to be less that physical capacity or policy constraints [20]:

$$U_K = \sum_{\forall C \in \mathcal{C}} \frac{D_{K,C}}{D_{K_r,C}} U_{K_r,C} < b_K, \quad \forall K \in \mathcal{K} \tag{4.1}$$

or exact the physical capacity or policy constraints:

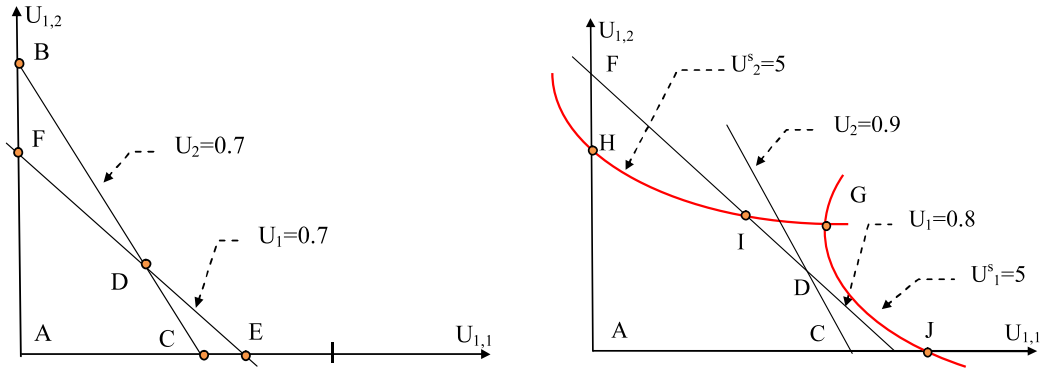
$$U_K = \sum_{\forall C \in \mathcal{C}} \frac{D_{K,C}}{D_{K_r,C}} U_{K_r,C} = b_K, \quad \forall K \in \mathcal{K} \tag{4.2}$$

where $0 \leq U_{K_r,C} \leq b_K, \forall C \in \mathcal{C}$; $\mathcal{C}$ and $\mathcal{K}$ are the sets of classes and resources; $K_r \in \mathcal{K}$ is a reference resource shared by all classes of request[1]; $U_K$ is the total utilization of resource $K$; $U_{K_r,C}$ is the utilization of resource $K_r$ by requests of class $C$, and $D_{K,C}$ is the demand of the resource $K$ in class $C$. $b_K$ is the utilization limit for resource

[1]The existence of a resource shared by all classes simplifies the analysis; the results presented in this thesis are valid without this assumption.

$K$; for example, the maximum utilization of a single CPU device is 1, the utilization of a dual core CPU is 2, etc.

For the example described in Figure 3.4, if device 1 ($CPU_1$) is the shared device ($K_r$) and if we represent the inequation (4.1) in the reference device utilization space, we obtain the diagram of Figure 4.1a, where each segment represents one of the equations (4.2). The *stress space* is within the area AFDC. The coordinates $(U_{1,1}, U_{1,2})$ of the points F, D, C can be found by solving the system of equalities (4.2). The coordinates are those values for which one or more equations reach the limits $b_K$. On segment FD, device 1 is the bottleneck and on segment DC device 2 is the bottleneck. Note that we cannot drive the system out of the performance stress area because we either would exceed the capacity limits or violate policy constraints.




**(a)** *Hardware.* **(b)** *Hardware and software.*

**Figure 4.1:** *Constraints on the stress space.*

### 4.1.2 Software Constraints

We can extend the above discussion for the software layer. Consider $1 \ldots L$ software queues at the software layer.

Since the software layer is a normal queuing network (*separable* queuing network, a subset of general networks of queues, where assumptions like Flow Ballance Assumption hold [17]), we can apply the general queuing laws. Thus, using the utilization law [17], the utilization of a software resource $L$ in class $C$ is:

$$U^s_{L,C} = X^s_C \times R^s_{L,C}, \quad L \in \mathcal{L}, C \in \mathcal{C} \tag{4.3}$$

where $\mathcal{L}$ denotes the set of all software resources in the distributed system and $\mathcal{C}$ the set of all classes of services. The total utilization of a software resource $L$ is:

$$U^s_L = \sum_{C \in \mathcal{C}} X^s_C \times R^s_{L,C} \tag{4.4}$$

Assuming that there exists a *hardware resource* $K_r$ shared by all classes (for example a shared web server's CPU), we can express the utilization of resource $K_r$ in class $C \in \mathcal{C}$:

$$U_{K_r,C} = X_C \times D_{K_r,C} \tag{4.5}$$

The throughput at the both hardware layers must be the same, at steady state both layers process the same number of requests/seconds, therefore $X_C = X^s_C$. By replacing $X^s_C$ in (4.3) with the one from (4.5) and performing some simple algebraic operations, we can express the utilization of any software resource $L$ in class $C$ as a function of utilization of hardware resource $K_r$ in the same class $C$:

$$U^s_{L,C} = U_{K_r,C} \frac{R^s_{L,C}}{D_{K_r,C}}, \quad \forall L \in \mathcal{L}, \forall C \in \mathcal{C} \tag{4.6}$$

Thus, using resource $K_r$ as reference, we can rewrite (4.4) as:

$$U_L^s = \sum_{C \in \mathcal{C}} U_{K_r,C} \frac{R_{L,C}^s}{D_{K_r,C}}. \tag{4.7}$$

The utilization of each software contention centre $L$ is limited by the capacity or policy constraints $b_L$, and that can be expressed as:

$$U_L^s = \sum_{C \in \mathcal{C}} U_{K_r,C} \frac{R_{L,C}^s}{D_{K_r,C}} < b_L, \quad \forall L \in \mathcal{L} \tag{4.8}$$

and equation (4.2) can be rewritten as:

$$U_L^s = \sum_{C \in \mathcal{C}} U_{K_r,C} \frac{R_{L,C}^s}{D_{K_r,C}} = b_L, \quad \forall L \in \mathcal{L} \tag{4.9}$$

where $0 \leq U_{L,C}^s \leq b_L$, $\forall C \in \mathcal{C}$.

These equations are non-linear because the terms $R_{L,C}^s$ depend non-linearly on $U_{K_r,C}$ [18], i.e $R_{L,C}^s = h(U_{K_r,C})$, where $h$ is a non-linear function. The function $h$ is the queuing network at the hardware layer. $R_{L,C}^s$ can be computed by solving the hardware queuing network model.

For the example described in Figure 3.4, if device 1 ($CPU_1$) is the shared device $K_r$ and if we represent the equation (4.9) in the reference device utilization space, then we obtain the diagram of Figure 4.1b. The *stress space* is within the area AHIDC and is certainly different than when we consider only hardware resources. The coordinates $(U_{1,1}, U_{1,2})$ of the points H, I, G, J and the corresponding segments can be found by solving the equation 4.3. The coordinates are those values for which one or more equations reach the limits $b_L$. Note that some of the points are outside

of the stress area, they cannot be reached. By taking into account the software constraints, the bottlenecks will evolve as follows: on segment HI, software entity AS is the bottleneck, on segment ID the hardware device 1 is the bottleneck and on segment DC device 2 is the bottleneck. Note that we cannot drive the system out of the stress area because either we would exceed the capacity limits or we violate policy constraints. Therefore, software entity DB is never saturated, although it comes very close.

In mathematical programming terms ([90]), the points B, C, D, E, F, H, I, G, J in Figures 4.1a and 4.1b are called *extreme points*. *Extreme points* are those points in the solution space where $U_K = b_K$ or $U_L^s = b_L$, for some hardware or software resource $K$ or $L$. The domain delimited by the most interior constrains (like AHIDC in Figure 4.1a) is our *feasible stress space*. In mathematical programming terms, a linear function will reach the maximum or minimum in the extreme points of its feasible space. A non-linear function will reach its extreme values on the boundary of the feasible space. Therefore, the maximum (or minimum) of any performance stress metric (response time, throughput, buffer length, utilization, number of threads, etc.) is achieved on the boundary of the feasible stress space. It turns out that if we can explore the boundary, then we can find the maximum or the minimum of those metrics.

### 4.1.3 Workload Stress Vectors

Since we know how to analytically compute the feasible stress space boundary, including the *extreme points*, we need a mechanism to reach those boundaries on the real system. Unfortunately, we cannot drive utilization directly. On the real system

we stress the system by generating the workloads, i.e. by accessing the URLs and by synthetically generating a number of users for each request type.

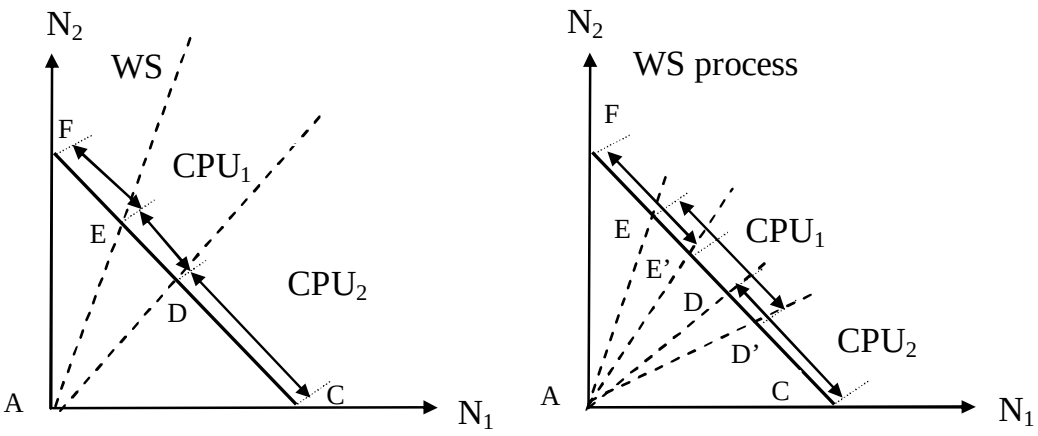
Note that neither $N$ nor $N_C$ are directly visible in (4.1) and (4.8), but they are directly involved in producing per class utilizations $U_{K_r,c}$. Our hypothesis is that it is possible to find $N_C$, when $N$ is known, by using the solutions of equations (4.2) and (4.9). We rely on an early result, established for the asymptotic case for one layer hardware queuing networks in [91]. Our conjecture is that, if a solution of equation (4.2) and/or (4.9) is $U^*_{K_r,C}$, then the workload mix that yields that solution can be approximated as:

$$\beta^*_i = \frac{N_C}{N} = U^*_{K_r,C}. \tag{4.10}$$

The vectors $\beta^* = \langle \beta^*_1, \ldots, \beta^*_{|\mathcal{C}|} \rangle$ are *workload stress vectors* and are found by solving the equations (4.2) and (4.9) and computing all per class utilizations using (4.6). Figure 4.2 shows the stress vectors in the space of $N_1$ and $N_2$ (dashed lines), the number of users in class 1 and 2 respectively. On the dashed lines the ratio of users remains constant. When the software and hardware entities do not saturate, there is one bottleneck on each sector (Figure 4.2a). When the entities saturate, then we can have multiple saturation devices for a range of population mixes. For example, both WS server process and the CPU of Application Server are bottlenecks on the segment EE' in Figure 4.2b.

## 4.2 The Autonomic Testing Framework

Figure 4.3 shows the proposed framework for autonomic performance testing. The framework will drive the system along the workload stress vectors until a performance




**(a)** *Non-saturated bottlenecks.* **(b)** *Saturated bottlenecks.*

**Figure 4.2:** *Bottlenecks in population mix space, $N = N_1 + N_2$. As population mix changes, the bottleneck shifts.*

stress goal is reached. A *stress goal* is target performance metric threshold, such as a software or hardware utilization value, a target response time or throughput for a class of request, etc.

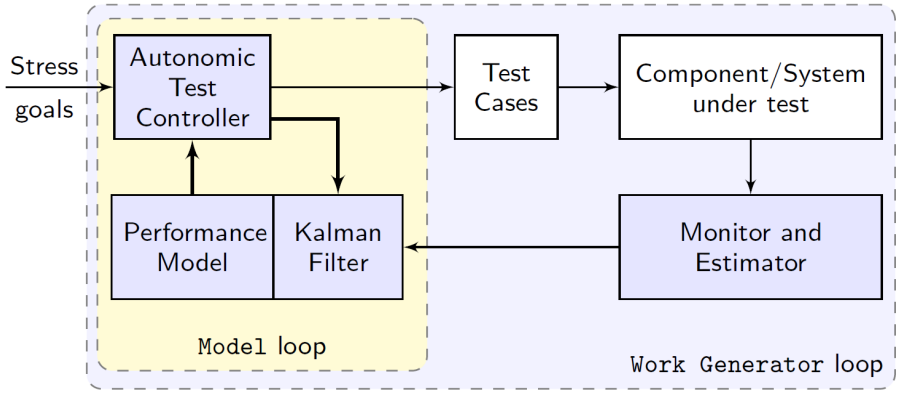



**Figure 4.3:** *Autonomic performance stress testing.*

An *autonomic test controller* runs the performance stress algorithm that will be presented later. In a nutshell, at each iteration it simulates a number of users that

simultaneously access the system/component that is tested. Based on current state of the system and on the *stress goals*, a new workload will be computed and generated. Basically, the algorithm drives the system along the feasible stress test boundary or along the stress vectors.

During the test, the system is continuously monitored by a *performance monitor* and performance data is extracted. Data includes *CPU utilization*, *CPU time*, *disk utilization*, *disk time*, *waiting time* (which includes time waiting in critical sections, thread pools, connection pools), *throughput*, etc. Also, the *monitor* component will extract information about the workload that generated the data and information about the system. The monitored data is filtered through an estimator for error correction and noise removal. Estimators, like Kalman filters [77], have been proven to be very effective in estimating the demands [75] and we have used those in our implementation.

The performance data is passed to the *performance model* made of two queuing network layers. The model has 3 main functions: (a) it computes the $R_{K,C}$ (see Figure 3.4); (b) provides the equations and solutions for the load stress vectors (4.10) and (c) is used by the workload generator in lieu of real system to navigate along the stress vectors.

### 4.2.1 The Autonomic Test Controller

This section presents in detail the algorithm run by the autonomic test controller. After the classes of service are defined, the framework will use a model to make estimations about the number of users required and the classes they should execute in order to reach the stress goal (for example, utilization or response time above a

specified threshold).

By solving equations (4.2) and (4.9) and then using (4.10) we can compute the *workload stress vector* $\beta$. Now the goal becomes finding the total number of users $N$ that will drive the system on the feasible space boundary along the stress vectors.

We have developed the *Stress Algorithm* that will find the number of users and their mix that will first reach the stress goal. That will guarantee that beyond that number, we either go beyond the policy constraints (when they limit the feasible space) or we are guaranteed we stay on the boundary of feasible space.

After solving equations (4.2) and (4.9) and getting the system bottlenecks, the algorithm has two loops:

- in the first loop (***`Model`*** *loop* in Figure 4.3) the number of users to reach the boundary of the feasible space on each stress vector and saturate a bottleneck is computed on the model;
- in the second loop (***`Work Generator`*** *loop* in Figure 4.3) the algorithm works with the real system, submitting requests and measuring the performance. This loop is initialized with the values from the first loop and corrects the eventual errors inherent in working with a model instead of the real system.

Both loops follow similar feedback ideas: having an *extreme point* $p$ and the total number of users $N$, the *workload stress vector* is computed by using relations (4.10); then the performance metric corresponding to the stress goal for this workload mix is determined, either by solving the model (***`Model`*** *loop*), or by generating workload and measuring the performance metric on the real system (***`Work Generator`*** *loop*). This performance metric is compared with the target value. If the stopping condition is not met, the framework will use a hill-climbing strategy to find a new value

**Algorithm 4.1:** Stress Algorithm – algorithm to find the number of users that will bring the performance metric of a resource $K \in \mathcal{K} \cup \mathcal{L}$ at a target value $PM_K$.

**input** : $N$ – the initial number of users
**input** : $PM_K$ – the targeted performance metric for resource $K \in \mathcal{K} \cup \mathcal{L}$
**input** : $err$ – accepted error

1 Tune the model by measuring and adjusting the service demands for each class;
2 Find all extreme points by solving the equations (4.2) and (4.9);
3 Compute the workload stress vectors, by using (4.10);
4 **foreach** *stress vector* $p \in \mathcal{P}$ **do**
5  $pm_{e,K} \leftarrow -1$; // estimated performance metric
6  $pm_{m,K} \leftarrow -1$; // measured performance metric
7  Tune the model for stress vector $p$ and $N$ users;
  // Stop when the estimated performance metric is within $err\%$ from the target performance metric
8  **while** $\left|1 - \frac{pm_{e,K}}{PM_K}\right| > err$ **do**
9  Compute $\langle N_1, N_2, \ldots, N_{|\mathcal{C}|} \rangle$ for $N$ and $p$;
10  Solve model for $\langle N_1, N_2, \ldots, N_{|\mathcal{C}|} \rangle$;
11  Update $pm_{e,K}$ with the estimated value;
12  **if** $\left|1 - \frac{pm_{e,K}}{PM_K}\right| > err$ **then**
13  Update $N$ using a hill climbing strategy;
14  Generate workload and measure the metrics;
15  Update $pm_{m,K}$ with the value measured;
16  **if** $\left|1 - \frac{pm_{e,K}}{pm_{m,K}}\right| > err$ **then**
17  go to line 7;
18  **while** $\left|1 - \frac{pm_{m,K}}{PM_K}\right| > err$ **do**
19  Compute $\langle N_1, N_2, \ldots, N_{|\mathcal{C}|} \rangle$ for $N$ and $p$;
20  Generate workload and measure the metrics;
21  Update $pm_{m,K}$ with the value measured;
22  **if** $\left|1 - \frac{pm_{m,K}}{PM_K}\right| > err$ **then**
23  Update $N$ using a hill climbing strategy;

for $N$ and a new iteration will start. In our algorithm we stop each loop when the predicted/measured performance metric is within $err\%$ from the target value. However, other conditions can be used, such as the performance metric is with at

most $err\%$ *above* the target or a combination of multiple resources' metrics.

In the first step (line 1) the algorithm tunes the two-layer model. In essence it estimates the demands $D_{K,C}$ for hardware resources. The demands are important for the system of equations (4.2) and (4.9). For most infrastructures, per class service times or demands are hard or costly to measure. We estimate those values by using Kalman filter as illustrated by the Model Tuning Algorithm.

**Algorithm 4.2:** Model Tuning Algorithm – estimate the demands for each resources in each class.

**input** : $N$ – the number of users
**input** : $err$ – the accepted error
**output** : $D$ – demands matrix, of size $|\mathcal{C}| \times |\mathcal{K} \cup \mathcal{L}|$

```
1  for i ← 1 to |C| do
2      N ← ⟨0, 0, ..., 0⟩;
3      N_i ← N;
4      Generate workload;
5      foreach K ∈ K ∪ L do
6          while |1 − pm_{e,K} / pm_{m,K}| > err do
7              Solve model;
8              Update pm_{e,K} with the model estimated value;
9              Update pm_{m,K} with the measured value;
10             if |1 − pm_{e,K} / pm_{m,K}| > err then
11                 Estimate service demands using Kalman filters;
12                 Update model with the estimated service demands;
13         Update D_{C_i,K} with the last value estimated by Kalman filters;
```

In order to get the demands for a class, we generate workload by considering that all users will be in that class and no user will access other classes (line 2). Then, for each resource we execute a loop to find the correct value for demand: we solve the model to extract the estimated value and also measure the performance metric—if the two values are close enough (again, our criterion is that the estimated value is within $err\%$ from the measured value, but other criteria can be used) then we accept the demand found by the Kalman filter, else we move to the next iteration.

## 4.3 Experiments

We tested our framework on a a web application deployed over cluster with three Windows XP machines: one Database Server (MySQL) and two Web Servers (Tomcat). Also we had on a machine a Workload Balancer (Apache) to distribute the incoming web requests to the two web servers. Figure 4.4 shows our deployed testing framework that is a materialization of the logical structure presented earlier in Figure 4.3. The purpose of the experiments is to show that our framework find the points in workload space that cause the worst case behaviour by investigating on the real system only a very small fraction of the total number of workload mixes.

On each machine we had installed monitoring tools to be able to measure the performance metrics: Windows XP SNMP agents, JMX and Windows Performance Counters. The workload generator and the analysis of the performance data was on a separate machine.

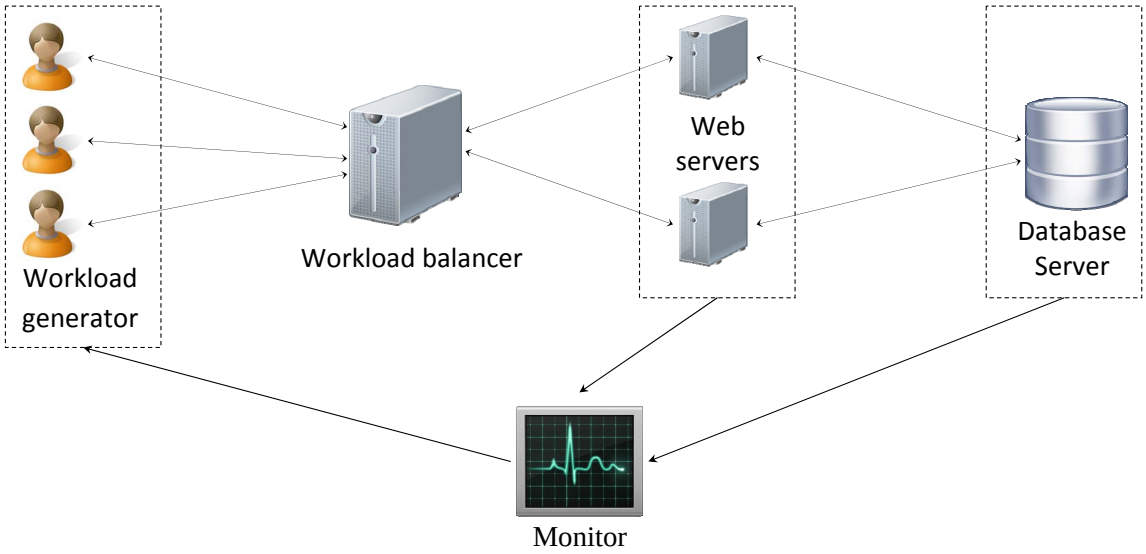



**Figure 4.4:** *The cluster used for experiments.*

On the cluster we have installed a typical 3-tier application–an online store–with 3 main scenarios:

- `browse` – the user is browsing through the available items in the store. Also the user will be able to specify how many items he wants to have in a single page (which is a parameter for the scenario);
- `buy` – the user decides to buy some items and add them in the shopping cart;
- `pay` – the user goes to checkout and pays for the content of the shopping cart;

The two-layer model of the application is the one represented in Figure 3.4, earlier in this chapter. At the software layer, we have the two software queuing centres, the Web Servers and the Database. The load balancer is not represented in this particular example as a queuing centre because it is performant enough not to queue requests (however, in a general case it should be represented). Therefore, $\mathcal{L} = \langle Web, Data \rangle$.

The hardware layer is made of two queuing centres $\mathcal{K} = \langle CPU_{web}, CPU_{data} \rangle$.

The application was modeled with Apera tool [21], developed by one of the authors.

Initially, we have 3 classes of requests, represented by the 3 scenarios, $\mathcal{C} = \{\texttt{browse}, \texttt{buy}, \texttt{pay}\}$.

### 4.3.1 Classes of Service

In general, the number of classes is equal with the number of scenario because, most of the times, the performance metrics are not significantly influenced by the arguments values and we can consider that a scenario is a single class of service. Alternatively, we can consider each scenario with the maximum argument for stress testing or average

argument for performance testing. However, there are situations when we need to split a scenario in more classes. In our testing framework, we probe each scenario with randomly generated arguments and we measure the resulting stress goal metric, for example CPU utilization[2] is measured. The scenarios that have a high variance in the performance metrics (utilization in our example) are most likely to provide significant improvements if they are split. The other scenarios will generate a single class.

When enough samples have been gathered, we can split the utilization interval in subintervals and for each such subinterval we will determine the corresponding range for scenario parameters. Each such range will generate a class of service.

A user executing the scenario `browse` will send a request to a web server, that will generate a request to a database server to select a number of items from database. The result will be sent back to the web server that will create a web page containing all the selected items. The number of items to be displayed will be specified as a parameter to the scenario.

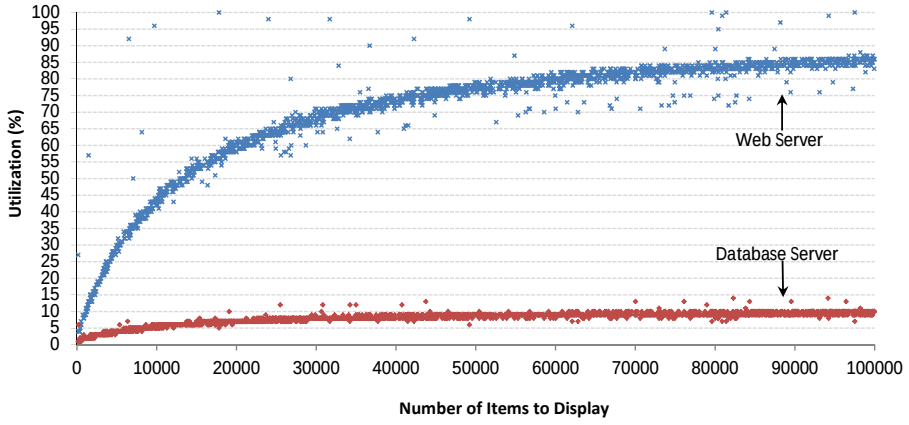



**Figure 4.5:** *The CPU utilization on the web server and database server when the* `browse` *scenario is executed.*

[2]Any other performance metric can be used: throughput, disk utilization, etc.

Figure 4.5 shows the measured CPU utilization on the web server and database server when the parameter goes from 0 to 100,000. We see that the CPU utilization at the web server increases fast at first (for values lower than 20,000) and then slows down.

The CPU utilization at the database server follows a similar pattern, although the maximum value is lower than 15%.

Because the web server CPU utilization grows faster and get closer to 100% we will use it to split the scenario into four classes (first class for utilization between 0 and 25%, second class for utilization between 25% and 50%, etc.). For that, considering that the utilization follows a logarithmic shape, we can do regression. Table 4.1 summarizes the ranges for the parameter corresponding to the four classes of service.

| Scenario | Range | | |
|---|---|---|---|
| `browse 0` | 0 | - | 2,855 |
| `browse 1` | 2,856 | - | 11,752 |
| `browse 2` | 11,753 | - | 48,370 |
| `browse 3` | 48,371 | - | 100,000 |

**Table 4.1:** *The ranges of the parameter when the scenario is split in four classes.*

Finding the right number of classes to split a scenario is a hard problem on it's own. If we have too many classes, the accuracy of the algorithm will increase, but so does the complexity. Too few classes, and the model will be solved very fast at the expense of precision.

### 4.3.2 Results

Once we decided the number of classes of services, we run the testing algorithm. At the first step it estimates the demands for each class of service using the Model

Tuning Algorithm (Table 4.2 shows the values found when the model is calibrated for CPU utilization).

| | $CPU_{web}$ | $CPU_{data}$ |
|---|---|---|
| `buy` | 5.38 | 5.60 |
| `pay` | 5.17 | 5.60 |
| `browse 0` | 41.32 | 11.78 |
| `browse 1` | 192.84 | 39.91 |
| `browse 2` | 769.29 | 153.20 |
| `browse 3` | 1,961.82 | 372.30 |

**Table 4.2:** *The values for demands for each scenario found using Kalman filters (milliseconds).*

We run the framework for 3 stress goals: (a) hardware utilization, (b) web container number threads and (c) response time for each scenario.

*The hardware utilization stress goal.* This goal aims at performance of the system when a hardware resource runs at a threshold utilization. This can be maximum utilization 100%, if reachable, or less than that (to resemble the operational conditions). When the system reaches the target conditions, performance metrics are collected and analyzed. For illustration purposes, we set the target utilization at 50% and want to find the number of users for each stress vector that will yield that utilization.

The stress algorithm found 22 bottlenecks in the system (the workload stress vectors are shown in Table 4.3). Each one is a vector of six values, each value representing the proportion of the users that have to access that scenario in order to saturate the bottleneck. The order of the scenarios considered is ⟨`buy`, `pay`, `browse 0`, `browse 1`, `browse 2`, `browse 3`⟩.

For each stress vector, the first loop of the stress algorithm uses the model to find the number of users $N$ that will bring the utilization at the specified threshold

| # | Workload stress vectors |
|---|---|
| 1 | ⟨ 1, 0, 0, 0, 0, 0 ⟩ |
| 2 | ⟨ 0, 1, 0, 0, 0, 0 ⟩ |
| 3 | ⟨ 0, 0, 1, 0, 0, 0 ⟩ |
| 4 | ⟨ 0, 0, 0, 1, 0, 0 ⟩ |
| 5 | ⟨ 0, 0, 0, 0, 1, 0 ⟩ |
| 6 | ⟨ 0, 0, 0, 0, 0, 1 ⟩ |
| 7 | ⟨ 0.947, 0, 0.053, 0, 0, 0 ⟩ |
| 8 | ⟨ 0.985, 0, 0.015, 0, 0, 0 ⟩ |
| 9 | ⟨ 0.952, 0, 0, 0.048, 0, 0 ⟩ |
| 10 | ⟨ 0.990, 0, 0, 0.010, 0, 0 ⟩ |
| 11 | ⟨ 0.952, 0, 0, 0, 0.048, 0 ⟩ |
| 12 | ⟨ 0.991, 0, 0, 0, 0.009, 0 ⟩ |
| 13 | ⟨ 0.953, 0, 0, 0, 0, 0.047 ⟩ |
| 14 | ⟨ 0.991, 0, 0, 0, 0, 0.009 ⟩ |
| 15 | ⟨ 0, 0.896, 0.104, 0, 0, 0 ⟩ |
| 16 | ⟨ 0, 0.970, 0.030, 0, 0, 0 ⟩ |
| 17 | ⟨ 0, 0.905, 0, 0.095, 0, 0 ⟩ |
| 18 | ⟨ 0, 0.980, 0, 0.020, 0, 0 ⟩ |
| 19 | ⟨ 0, 0.906, 0, 0, 0.094, 0 ⟩ |
| 20 | ⟨ 0, 0.981, 0, 0, 0.019, 0 ⟩ |
| 21 | ⟨ 0, 0.907, 0, 0, 0, 0.093 ⟩ |
| 22 | ⟨ 0, 0.982, 0, 0, 0, 0.018 ⟩ |

**Table 4.3:** *The workload stress vectors found.*

(50% in our experiments). Then the workload generator will simulate $N$ users and measure the metrics. If the measured and the estimated metric values are not close (in our experiments, within 10% from each other) a new calibration of the model is performed and then the first loop is executed again. If the two values are close, $N$ is finely tuned by the second loop which sends requests to the real system.

From Table 4.4, we can see that the number of users computed by the first loop (column `Usr (e)`) using only the model was very close to the actual number found by the second loop (column `Usr (m)`). Therefore very few workloads were used against the real system (column `Mx.`). For this particular example the hill climbing method of the second loop tried less than 10 workloads for each stress vector, with the exception

of the $20^{th}$ vector.

When we limit the number of threads on the web server to 5 (thus creating software queues), we see that most of the times the number of users required to bring the CPU utilization above the 50% threshold is less, as shown in Table 4.4b.

What we can infer from Table 4.4 is that the 50% utilization can be reached for a variety of workloads. For example, in Table 4.4a, if we go along the stress vector 1, the utilization is reached for 282 users. Along the vector 6, the same utilization is reached for 3 users. This certainly shows a very big limitation of the system when the workloads are shifted toward the class 6. Similar conclusions can be drawn for the case illustrated in Table 4.4b.

*Software utilization goal.* We run the algorithm again, targeting the software utilization of the web server. We set the maximum number of threads to handle requests on the web server to 5. The stress goal for the algorithm is 50% utilization of the container, meaning the use of at most 2.5 threads on average. When calibrated for this goal, the model found only 6 stress vectors, each one having all users executing a single scenario. The results are summarized in Table 4.5a.

We notice that the numbers predicted by the model (first loop of the algorithm) was not as accurate as in the hardware utilization case. Nevertheless, the second loop was were able to find the correct number (column `Usr (m)`). Because the model had to be re-calibrated several times, the number of generated workloads (column `Mx.`) is higher for some stress vectors. However, the total number of workloads generated was 43, which is significantly lower than trying all possible workloads.

Similar to hardware utilization, there is a large range of workloads that use 50% of the threads. This shows the pitfalls of not exploring the whole space, the web

| Str. vec. | Mx. | Usr (e) | Usr (m) |
|---|---|---|---|
| 1 | 3 | 281 | 282 |
| 2 | 10 | 265 | 275 |
| 3 | 4 | 70 | 72 |
| 4 | 2 | 16 | 16 |
| 5 | 4 | 5 | 5 |
| 6 | 6 | 3 | 3 |
| 7 | 9 | 236 | 252 |
| 8 | 10 | 270 | 279 |
| 9 | 4 | 211 | 213 |
| 10 | 5 | 264 | 266 |
| 11 | 7 | 98 | 96 |
| 12 | 4 | 244 | 245 |
| 13 | 9 | 59 | 54 |
| 14 | 3 | 190 | 190 |
| 15 | 3 | 248 | 249 |
| 16 | 3 | 257 | 258 |
| 17 | 6 | 138 | 142 |
| 18 | 6 | 236 | 238 |
| 19 | 6 | 53 | 53 |
| 20 | 12 | 196 | 180 |
| 21 | 6 | 29 | 27 |
| 22 | 8 | 136 | 142 |

**(a)** *Only hardware queues.*

| Str. vec. | Mx. | Usr (e) | Usr (m) |
|---|---|---|---|
| 1 | 6 | 258 | 261 |
| 2 | 4 | 249 | 248 |
| 3 | 7 | 62 | 70 |
| 4 | 2 | 16 | 17 |
| 5 | 2 | 6 | 6 |
| 6 | 3 | 3 | 3 |
| 7 | 4 | 237 | 239 |
| 8 | 4 | 254 | 256 |
| 9 | 3 | 204 | 204 |
| 10 | 5 | 245 | 247 |
| 11 | 2 | 88 | 88 |
| 12 | 5 | 239 | 239 |
| 13 | 2 | 54 | 54 |
| 14 | 4 | 180 | 179 |
| 15 | 3 | 232 | 233 |
| 16 | 4 | 242 | 244 |
| 17 | 8 | 136 | 131 |
| 18 | 9 | 227 | 234 |
| 19 | 5 | 42 | 42 |
| 20 | 4 | 188 | 187 |
| 21 | 6 | 30 | 31 |
| 22 | 8 | 147 | 142 |

**(b)** *Hardware and software queues.*

**Table 4.4:** *The users number that will bring the CPU utilization (on web server or database server) above 50%.*

container can saturate with very few users, if they come in class 6.

*The response time goal.* In the last experiment, we wanted to see the maximum number of users that will ensure that the response time for requests on each scenario does not exceed a certain value. Because the demands for classes vary greatly, we chose a different target value for each class. Our threshold vector is ⟨50, 50, 100, 500, 1500, 5000⟩ for the classes ⟨`buy`, `pay`, `browse 0`, `browse 1`, `browse 2`, `browse 3`⟩ (the times are expressed in milliseconds). Tuning the model for this goal gave us

| Str. vec. | Mx. | Usr (e) | Usr (m) |
|---|---|---|---|
| 1 | 12 | 316 | 144 |
| 2 | 13 | 196 | 144 |
| 3 | 11 | 52 | 69 |
| 4 | 3 | 20 | 22 |
| 5 | 2 | 10 | 10 |
| 6 | 2 | 7 | 8 |

**(a)** *Web container utilization.*

| Str. vec. | Mx. | Usr (e) | Usr (m) |
|---|---|---|---|
| 1 | 9 | 185 | 171 |
| 2 | 13 | 233 | 170 |
| 3 | 9 | 98 | 93 |
| 4 | 4 | 28 | 27 |
| 5 | 2 | 9 | 9 |
| 6 | 2 | 8 | 8 |
| 7 | 15 | 149 | 119 |
| 8 | 15 | 111 | 87 |
| 9 | 15 | 20 | 107 |
| 10 | 2 | 3 | 3 |
| 11 | 3 | 36 | 36 |
| 12 | 3 | 45 | 45 |
| 13 | 7 | 17 | 14 |
| 14 | 4 | 31 | 33 |
| 15 | 11 | 2 | 9 |
| 16 | 5 | 30 | 30 |

**(b)** *Response times.*

**Table 4.5:** *The number of users found when the performance metric is web container utilization and response time for each class.*

16 workload stress vectors. The results are shown in Table 4.5b.

Although the model was not as accurate as for hardware utilization, being necessary several re-tunnes, each stress vector required at most 15 workloads to be investigated (column `Mx.`) before the number of users is found (column `Usr (m)`). Usually the more accurate is the model, fewer workloads need to be generated.

We also noticed that for the stress vector 10 (which is $\langle 0, 0.224, 0.776, 0, 0, 0 \rangle$—22.4% of the users execute scenario `pay` and 77.6% execute `browse 0`) there are necessary only 3 users to get a the response time greater than our threshold. If all users execute `pay` (stress vector 2) there are necessary 170 users, and it all of them execute `browse 0` (stress vector 3), then 93 users are required. The stress vector 10

shows the dramatic effect that the combination of the scenarios have.

Again, we notice that the total number of tested workloads, 119, is much lower than the size of the search space.

### 4.3.3 Complexity of the algorithm

The workload mix space is combinatorial in size. The number of total workload mixes when there are $N$ users and $C$ classes is given by the formula [71]:

$$Mixes_{N,C} = \binom{N+C-1}{C-1} \tag{4.11}$$

The number of classes is known from the begining, but the number of users is not, so we would have to consider that $N$ goes from 0 to $N_{max}$, where $N_{max}$ is the maximum value we allow for $N$. Thus the total number of mixes is:

$$Mixes = \sum_{N=1}^{N_{max}} \binom{N+C-1}{C-1} \tag{4.12}$$

Table 4.6 shows this combinatorial explossion:

| $N_{max}$ | $C$ | $Mixes_{N_{max},C}$ | Total Mixes |
|---|---|---|---|
| 200 | 1 | 1 | 200 |
| 200 | 2 | 201 | 20 300 |
| 200 | 3 | 20 301 | 1 373 700 |
| 200 | 4 | 1 373 701 | 70 058 750 |
| 200 | 5 | 70 058 751 | 2 872 408 790 |
| 200 | 6 | 2 872 408 791 | 98 619 368 490 |

**Table 4.6:** *The size of the workload mix space.*

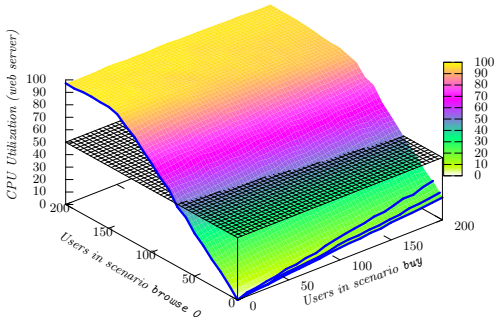
Exploring exhaustively the workload mix space is unfeasible. Our algorithm

explores just a small fraction, without needing $N_{max}$, and finds the minimum number of users that will bring the targeted metric value above the specified treshold.

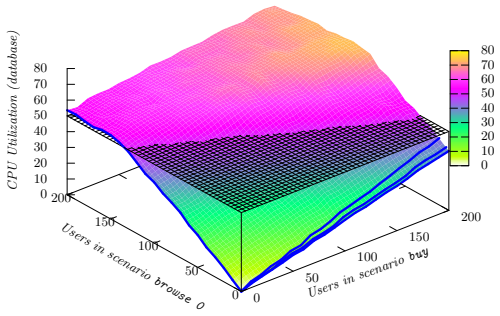
To better understand how the algorithm explores the workload mix space, lets consider only two classes: `buy` and `browse 0`. Figure 4.6 shows how the CPU Utilization (for web and database server), Web Container Utilization and Response Time (for each class) change with the workload mix. The utilizations are shown as percentages, and the response time is expressed in miliseconds. On each plot the threshold is shown as a plane parallel with the $X0Y$ plane, that intersects the $Z$-axis at 50 in Figures 4.6a, 4.6b, 4.6c and 4.6d and 100 in Figure 4.6e (the target was 50% utilization for CPU and web container, 50 ms response time for scenario `buy` and 100 ms response time for scenario `browse 0`).

The model found 4 bottlenecks and the stress vectors are shown with blue lines on the plots. The algorithm investigates only mixes on these lines. In the picture it can be seen that one of these lines will intersect the treshold plane in a point with minimum users among all other intersection points (it is always the vector that follows the $Y$-axis).

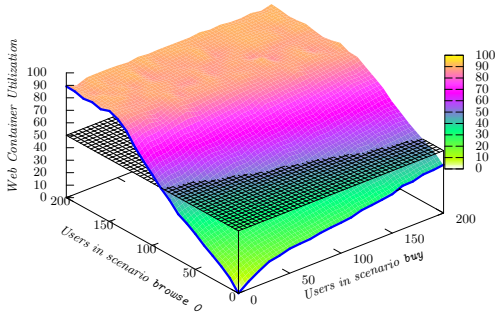
In our experiments, the framework was able to find the intersection point for one stress vector after less than 20 workload mixes were tried. That means our framework can find the overall minimum number of users with less than 80 workload mixes, which is a very low value compared with the size of the search space (20,300 for 2 clases and 200 maximum users).



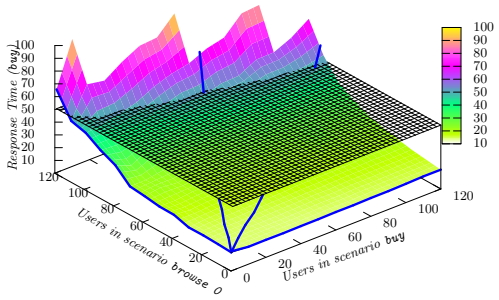

**(a)** *CPU utilization on web server.*



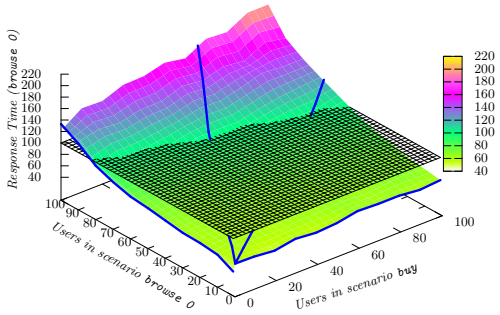

**(b)** *CPU utilization on database server.*




**(c)** *Web container utilization.*




**(d)** *Response time for* `buy`.




**(e)** *Response time for* `browse 0`.

**Figure 4.6:** *On Z-axis is the CPU utilization on the servers (**a** and **b**), web container utilization (**c**) and the response time of the two classes of service (**d** and **e**) when there are $N_1$ users in the class* `buy` *(X-axis) and $N_2$ users in the class* `browse 0` *(Y-axis).*

## 4.4 Related Work

Transactional systems in the context of autonomic computing have been modeled by several authors as regression models or Queuing Network Models (QNM). Dynamic regression models have been described in [92, 93]. Queuing Network Models were described in [94] as predictive models.

Early work in finding bounds on response time and throughput for one dimension of the workloads (one class) was done in [15, 16, 17, 18]. In [91] the authors showed that in multiple workload mixes, multiple resources systems, changes in workload mixes can change the system bottleneck; the points in the workload mix space where the bottlenecks change are called *crossover points*, and the sub-spaces for which the set of bottlenecks does not change are called *saturation sectors*. The same authors, in the same paper, showed analytical relations between the workload mixes and utilization at the saturated bottlenecks as well as analytical expressions for asymptotic (with saturated resources) response times, throughput, and utilization within the saturation sectors. The results were presented for one queuing network layer consisting of hardware resources. In this paper, we considered two layers with the emphasis on the software layer. Moreover, our approach is defined in the context or performance testing.

The paper [20] extended the results from [91] to non-asymptotic conditions (non-saturated resources), and used linear and non-linear programming methods for finding maximum object utilization across all workload mixes. That technique involved only the hardware bottlenecks. In our current approach, we consider the software bottlenecks and we combine the model search for worst case behaviour with a search on the real system.

There is no fully automatic method for building the structure of a performance model, however, there are available tools that can help in building a structure of the performance model [21]. Recent papers, like [22, 23, 24], have shown how to build a tracking filter and a predictive QNM such that the model's outputs always match those of the real system. Performance parameters like the service time, think times, and the number of users can be accurately tracked and fed into a QNM. In our approach, we estimate the demands on the hardware layer using a method similar to [24]

To the best of our knowledge, there is no performance model driven testing approach similar the one presented in this paper. Although there are many model driven performance activities, they do not refer to testing. Many researchers have targeted capacity planning of distributed and client-server software systems and specially the web based ones [25, 95, 26]. Amongst those, many approaches have used the widely recognized queuing models to model web applications at operational equilibrium [25, 26, 27] which has resulted in automated building of measurement based performance models [28, 29] or capacity calculators [30]. Others have tried to model the effect of application and server tuning parameters on performance using statistical inference, hypothesis testing and ranking (e.g. [31, 32]). In a rather different approach some have tried to automate the detection of potential performance regressions, by applying statistics on regression testing repositories [96, 33, 97]. This had enabled developers to identify subsystems that show performance deviations in load tests [98].

All these approaches have contributed to designing scalable systems, building on-demand performance management systems [99, 100, 101] and performance aware

software systems [102].

In this paper our focus was on model driven testing and on finding a method that drives the system towards a target state where performance metrics of interests can be collected. The model is fundamental in analysing the feasible stress space and in driving the system towards the saturation points.

## 4.5 Conclusions

This paper presented an autonomic performance testing method for stress testing web software systems. The systems are modeled with a two-layer Queuing Network Model. The model is used to find the software and hardware bottlenecks in the system and to give a hint about *workloads* that will saturate them. These hints are used as initial workloads on the real system and then in a feedback loop that guides the system towards a stress goal.

The workloads are characterized by *workload intensity*, which is the total number of users, and by the *workload mix*, which is ratio of users in each class of service. By extracting the *switching points* from the model, we are able to compute the *stress vectors* that yield a bottleneck change. Applying a hill-climbing strategy for workload intensity along the stress vectors, we are able to reach the stress goal.

We applied the method to find the workload intensity and workload mix that yields target software and hardware utilization limits or a target response time. The results show that the algorithm is capable to reach the target goal with a small number of iterations and therefore testcases.

Future work includes extending the framework to include more target goals, validate it on large scale software systems and address functional problems uncovered

by stress testing.

# Chapter 5

# Mitigating DoS Attacks Using Performance Model-Driven Adaptive Algorithms

DoS[1] attacks have increased in both volume and sophistication [8]. Attack targets include not only businesses and media outlets but also service providers such as DNS, Web portals, etc. A sophisticated DoS attack can be mounted by attackers without advanced technical skills. There are many advanced attacking toolkits freely available on the Internet [37], including LOIC (low-orbit ion cannon) [38]. It has been used to launch attacks on government sites, the RIAA and MPAA (recording and movie industry associations), and more through coordinated efforts such as Operation Payback [38] and Operation MegaUpload. DoS attacks are motivated by a variety of reasons (financial, political, ideological [39]), but regardless of motivation have similar

[1]A distributed denial of service (DDOS) attack is a subtype of DoS attacks; the more general term is used throughout this paper.

impact: lost revenue, increased expenses, lost customers, and reduced consumer trust.

In this chapter we propose a model-based adaptive architecture and algorithm focused on detecting DoS attacks at the web application level and mitigating them appropriately. A Dynamic Firewall component is added to the standard web application stack; all requests are routed through this firewall. Arriving HTTP[S] requests first encounter a reverse proxy[2], which processes requests based on a set of rules. A decision engine uses a performance model of the application, statistical anomaly detection, and monitoring data from the application to adaptively create, update, and remove rules based on the presence or absence of an attack. Based on the rules, requests are labelled as suspicious or regular. Regular traffic proceeds to the web application as usual, while suspicious traffic is forwarded to an Analyzer component which challenges the end user to verify they are legitimate (for example, using a CAPTCHA test as in [103]).

The major contributions in this work are:

- We introduce a model-based adaptive architecture and algorithm focused on detecting DoS attacks at the web application level and mitigating them appropriately.

- We describe a hybrid approach to detect the beginning of a DoS attack combining a model-based adaptive algorithm and statistical anomaly detection.

- We present an experiment demonstrating the strength of this hybrid approach compared to the previous approach, which relied only on statistical anomaly detection. This extended approach is less sensitive to tuning, and is also capable

[2] A reverse proxy is a type of proxy that retrieves the resources from the server in behalf of the client.

of detecting a type of complex DoS attack that our previous approach could not detect correctly.

- We design experiments comparing more complex statistical anomaly detection approaches, empirically demonstrating that a carefully tuned statistical model could be as effective as our approach in constrained situations, but was sensitive to tuning and could not protect against an unknown attack.

Our adaptive model-based approach is based on mathematical queuing theory and creates an abstract view of the application (rather than relying on an externally-defined baseline) and was able to detect unknown attacks.

This approach improves on the state-of-the-art in several key dimensions. The adaptive architecture is a novel approach to detecting and mitigating DoS attacks. In particular, using a combination of application performance modeling with statistical anomaly detection to establish a set of filtering rules is novel. Iteratively fine-tuning those rules using application- and system-level performance metrics synchronized with the performance model, is also novel; the use of performance models allows us to leverage existing work on constructing performance models. Detecting DoS attacks using a performance model enables the prediction and prevention of application overloading in simulation, rather than waiting for the overload to occur and responding. Prediction also allows for more precisely timed removal of filters, limiting the impact on legitimate traffic. Defining a DoS attack as *any traffic that exceeds the application's ability to handle it* is a straightforward definition that allows us to leverage capacity planning work to address this problem. Moreover, filtering traffic using application-level knowledge at the granularity of individual use case scenarios is more granular than typical mitigation approaches. This also reduces the impact

on legitimate traffic.

To validate this approach, we implemented the Dynamic Firewall and monitored the traffic to a multi-tier J2EE web application. We monitored the request arrival rate (both suspicious and regular), the response time, and the CPU utilization in regular conditions and attack conditions. The attack conditions were both emulated by a workload generator and created realistically using LOIC. We found that under both attack conditions, our approach adapted to block the DoS traffic, restoring response times to expected values within seconds.

To demonstrate the importance of the performance model, we conducted a series of experiments where the rules were generated using various statistical anomaly detection approaches without a performance model. We show the advantage of predictive modelling, the sensitivity of the statistical anomaly approach to the chosen metrics and the construction of the model of "normal" behavior, and that the performance model increases our ability to detect a broader variety of DoS attacks.

The remainder of this chapter is organized as follows. Section 5.1 reviews relevant work. Section 5.2 introduces our adaptive architecture and algorithm. Experiments that showcase the usefulness of this approach compared to anomaly detection are presented in Section 5.3, and the results are discussed in Section 5.4. Section 5.5 concludes the chapter.

## 5.1 Related work

We begin with an introduction to DoS attacks and describe some established methods for DoS mitigation. The approach presented in this paper constructs a performance model; while a full review is out of scope for this paper, Section 5.1.2 briefly introduces

some established achievements in performance modeling.

### 5.1.1 DoS attacks and existing mitigation approaches

A DoS attacker will send many repeated requests that require resources to generate replies. These requests may be low-level TCP requests or higher-level application requests (like GET requests for web pages). The attacker discards the replies, meaning it takes fewer resources to send requests than it does to send responses (this problem is compounded when the target uses SSL; a recently released prototype tool demonstrates a dangerous type of SSL DoS attack [40]). Even with this beneficial ratio, the attacker may not be able to achieve denial of service with a single machine. Distributed Denial of Service attacks harness the power of many distributed attackers to attack a single target; this paper includes DDoS attacks in the term DoS.

The most common DoS attack is a network type of attack. The usual defence is to deploy firewalls and intrusion detection and prevention systems. Firewall rules that implement ingress and egress filtering prevent spoofing attacks that originate on the local network and also prevent incoming traffic from impacting the local network. This impairs the ability of local computers to participate in DoS attacks [37]. Firewalls can also stop TCP-related DoS attacks such as SYN-Floods by implementation of SYN cookies.

DoS attacks which overload computer resources are known to be challenging to defend. Some experts argue the only possible solution for this problem is to improve security for all Internet hosts and prevent attackers from running DoS attacks [37]. An example of these source-end defence schemes is D-WARD [42]. Sachdeva et al. [43] identify a number of problems with source-end defence, principal among them doubt

that such mechanisms will be widely implemented.

Researchers who agree with the challenge of defence at the source suggest defence at the victim site. For example, Kargl suggests using available DoS protection tools augmented with load monitoring tools to prevent clients (or attackers) from consuming too much bandwidth [37]. Other examples include QoS regulation [44] and cryptographic approaches [45]. Sachdeva et al. [43] also identified challenges when defending from the victim network, including the computational expense of filtering traffic, the possibility of the defence tools themselves being vulnerable to DDoS, and incorrectly dropping legitimate traffic. While a variety of other approaches have been suggested (e.g., [46, 47, 48, 49, 50]), the current state of the art does not fully mitigate DoS attacks [51, 104].

### 5.1.2 Performance Models

The use of performance models for detecting DoS attacks, as proposed by this paper, is a novel approach. We propose to use a performance model to analyze the incoming traffic and detecting traffic that moves the system toward its saturation point versus traffic that can be handled without overloading the system.

Any hardware-software system can be modeled by two layers of queuing networks [13, 14]: one that describes the software resources and the other one for the hardware resources. This way, the system becomes a network of resources. Each class of service has a *demand* for each resource, which is the time that resource is needed to complete a single user request.

In multiuser, transactional systems, a *bottleneck* is a resource (software or hardware) that has the potential to saturate if enough users access the system. In general,

the resource with the highest demand is the bottleneck. However, when there are many classes of requests with different demands at each resource, the situation becomes more complex. A change in the *workload mix* (how users are split among the types of service available) may change the bottleneck, or there may be multiple simultaneous bottlenecks. When a bottleneck saturates, the overall performance of the system degrades quickly, and the system may appear unresponsive.

Early work was done to analyze a system from a performance point of view in [15, 16, 17, 18]. In [91, 20] the authors investigated the influence of workload mixes on the performance of the system, how bottlenecks change with the workload mix and when they become saturated. A method to uncover the worst workload mix and the minimum population required to saturate a system is presented in [105].

Some of the parameters for the model cannot always be measured and other methods are required to find the correct values. Also, the monitored data can contain noise that needs to be removed. In previous papers [22, 23, 24] the authors investigated how filters, like Kalman filters [77], can be effectively used with a predictive queuing network model (QNM) so that the model's outputs always match those of the real system. Performance parameters like the service time, think times, and the number of users can be accurately tracked and fed into a QNM.

Capacity planning of distributed and client-server software systems, particularly for web applications, is a common application area; a popular approach is using queuing models to model web applications at operational equilibrium [25, 26, 27] which has led to the automatic construction of measurement-based performance models [28, 29] or capacity calculators [30]. Others have tried to model the effect of application and server tuning parameters on performance using statistical inference,

hypothesis testing, and ranking (e.g. [31, 32]). Another approach automates the detection of potential performance regressions by applying statistics on regression testing repositories (such as Jiang in [33] and related earlier work). This has led to an approach for identifying subsystems that show performance deviations in load tests [98].

Tools have been developed to model and analyze a system from a performance point of view [35, 21, 36]. In [35], the authors present a tool designed to model *software* systems using layered queuing networks. The resources are grouped in layers and the requests move from layer to layer to get service. Once the model is solved, the output contains throughputs and utilizations for the software resources, distributions for the service time, queuing delays, etc.

### 5.1.3 The Optimization Performance Evaluation and Resource Allocator (OPERA)

One example of a queuing network model is OPERA [71], the model used in our approach, which uses layered queuing to build a model of both the hardware and software resources of an application and then reasons about the performance of that application in different environments. Given a description of a system (an application and how it is deployed), and a candidate workload to various components of that system, OPERA can predict the *utilization* of resources (like CPU and disk), *response time* of requests to each component, and *throughput*. This section provides a high-level overview of the approach and its capabilities; a formal specification of the model and algorithms is in [71].

The system is described in terms of topology (*nodes* that have resources and

perform work, the organization and groupings of the nodes, and the *network* that connects the nodes) and the users' interactions with it (*services* are offered by the system and accessed by users, requests move among various services, services are grouped in *classes of service*). The topology is defined in PXL, an XML-based domain specific language[3].

From this description, OPERA will build two queuing networks – one for the hardware layer (that includes hardware resources) and one for the software layer (that includes software resources, software resources grouped into *containers*, and threads). Resources are queuing centers; when a new request arrives to a resource that is not available (the CPU is busy, or the container has all threads in use, etc.), the request is put in a queue and will wait for the resource to become available. The queues are handled on first come, first served basis.

Each resource, either hardware or software, has a *demand* associated with it. The demand is the time (measured in milliseconds or seconds) necessary for that resource to handle a single request. Since requests in each class of service require varying amounts of work from resources, the demands must specified for each class of requests and for each resource (e.g. expected CPU time, expected IO time).

For hardware resources the demands can be either measured (using profiling tools) or estimated (using Kalman filters). For a software resource, the demand is the response time from the hardware layer. OPERA will solve the model for the hardware layer, extract the response times and input them into the software model as demands, and then solve the model for the software layer.

The output from OPERA includes estimates for response time, throughput, and

[3]A detailed specification for the PXL file can be found in [106, 36]. The tool is available online at http://www.ceraslabs.com/technologies/opera.

resource utilization. These estimates can be compared to measured values and, in case they do not match, a Kalman filter adjusts the model demands (see Section 5.2.2.3) to ensure the model's output is synchronized with that of the real system.

## 5.2 Adaptive DoS Mitigation

This section describes our novel approach to detecting and mitigating attacks on HTTP web applications[4]. We consider the term *web application* to mean all resources required to run the user-facing components, most commonly HTTP servers, application servers, and database servers.

To protect an application, we deploy an application-aware Dynamic Firewall to process all incoming requests based on an adaptively managed set of rules. All requests are determined to be either regular traffic and forwarded to the application, or suspicious traffic that is forwarded to an Analyzer component which performs a challenge-response test (e.g. a CAPTCHA [103]) to determine whether or not the suspicious traffic is from a legitimate user and should therefore be sent to the application. Requests are analyzed with high granularity; requests are grouped into classes of requests that represent a usage *scenario* (for example, "browsing an online catalog", or "checking out"). A class of requests is considered suspicious if it would (or does) cause the web application to be overloaded.

Figure 5.1 shows the complete architecture. The Dynamic Firewall is responsible for deciding if the requests to a particular scenario would overload the web application and creating rules to identify and handle these requests (Decision Engine), and for

[4]To simplify the presentation, our discussion will refer only to HTTP, but the general approach is also applicable to HTTPS requests with appropriate certificate management.

processing incoming traffic in accordance with these rules (Reverse Proxy). The Reverse Proxy is a simple context-aware http request router, which redirects legitimate requests to the Web Application and suspicious requests to the Analyzer. Proxy routing is rule-based; the same philosophy as a regular firewall, except that the rules are modified autonomically at runtime by the Decision Engine. The Decision Engine implements an adaptive system, using a two-pronged approach (both a performance model for prediction and statistical Anomaly Detection (AD)) to make decisions, as described in Section 5.2.2. The Analyzer tests the legitimacy of suspicious traffic caught by the rules, using a test to differentiate between human and automated agents (e.g., a CAPTCHA) (Section 5.2.3). First, Section 5.2.1 provides an overview of how incoming requests are processed.

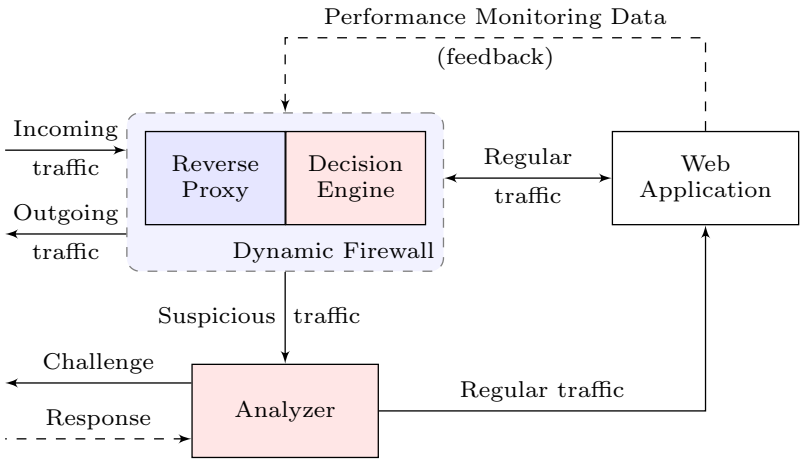



**Figure 5.1:** *DoS detection and mitigation architecture.*

### 5.2.1 Request Processing Overview

Adaptive mitigation of DoS attacks requires an understanding of the baseline behavior of the system under normal, no-attack situations. The Dynamic Firewall creates this

understanding by monitoring the application under normal load (either once deployed or based on development test cases in controlled conditions). Using the gathered measurements, a web application performance profile (WAPP) is calculated based on aggregated metrics about response time, request arrival rate, CPU utilization, etc. The contents and format of this profile are established per-application based on which measurements best capture the normal behavior of the system. The WAPP profile can be constructed automatically, with the permissible ranges of values hand-tuned by an administrator to establish the possible ranges of the performance metrics.

Given this understanding of the base performance of the web application, adaptive DoS attack mitigation requires the correct processing of application requests in three distinct contexts: stable, under attack, and post-attack.

*Stable:* The Decision Engine monitors application and OS performance metrics, constantly synchronizing the performance model with the current state of the system. It also analyzes the state of the web application; if it is statistically similar to the baseline constructed previously, the traffic is considered *regular*. Note that a previously-detected DoS attack may still be proceeding, but the rules created when the attack was detected are preventing that traffic from reaching the web application. The Decision Engine monitors existing rules; each iteration of the adaptation loop tests what would happen if individual rules were removed; if stability would be maintained, the rule can be removed.

Figure 5.2 shows a sequence diagram for requests arriving while the web application is experiencing regular traffic. When a request is received, the reverse proxy passes it to the Decision Engine which configures filtering rules (the rules are similar to a standard firewall). The rules relevant to the request are processed by the reverse

proxy, which forwards the request to the web application or to the Analyzer based on the rules.

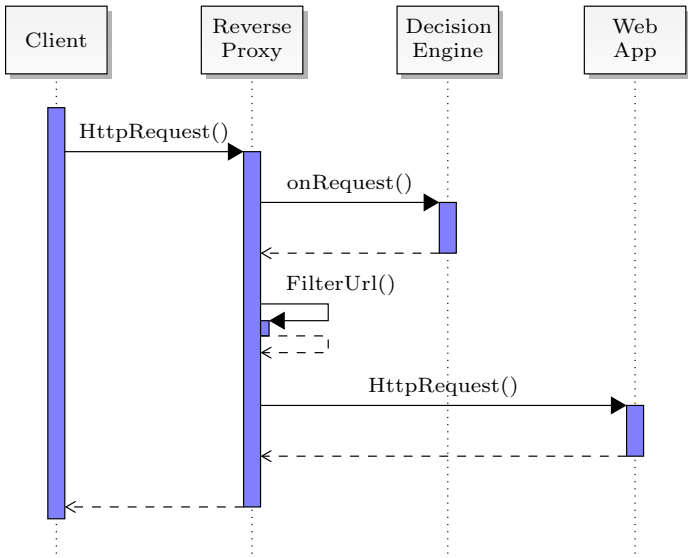



**Figure 5.2:** *Sequence diagram for handling regular traffic.*

*Under Attack:* When the Decision Engine identifies a class of application requests or an application component is under attack, it triggers the creation of new protection rules for the Reverse Proxy. To generate the new protection rules, the traffic collected by the Decision Engine is simulated in the performance model to predict its outcome on the web application. All traffic that is predicted to cause performance degradation is marked as suspicious and corresponding protection rules are added to the proxy (in our proof-of-concept implementation a rule is a regular expression that matches the requested URL, but of course a more robust representation is required in practice.).

Suspicious requests are redirected to the Analyzer for further assessment; typical DoS attack tools are not concerned with the response from the server, and so will not follow redirect requests. This means that the majority of DoS attack traffic will

be stopped at the Reverse Proxy.

*Post Attack:* The Decision Engine detects that the DoS attack is over when all incoming traffic could be handled by the web application, and responds by removing the rules currently diverting traffic to the Analyzer.

### 5.2.2 Decision Engine

DoS attacks often succeed when they overload bottlenecked resources, so the decision engine must detect traffic that has the potential to saturate the system. The Decision Engine implements an adaptive loop to manage the detection and mitigation decision-making, including monitoring the performance metrics from the application and the servers, identifying attack traffic based on statistical anomaly detection and a predictive performance model, using the performance model to predict behavior under given traffic conditions and making decisions accordingly, and creating rules to filter requests as they arrive. Figure 5.3 provides a conceptual overview of the adaptive loop in the decision engine and its place in the overall adaptive system.

The *performance goals* are target performance metrics, such as utilization values, response time, or throughput for a class of request, etc. Those values are derived from the WAPP.

The system is continuously monitored by a *performance monitor*. Data includes *CPU utilization*, *CPU time*, *disk utilization*, *disk time*, *waiting time* (which includes time waiting in critical sections, thread pools, connection pools), and *throughput*. The monitor also collects information on the workload (e.g. *arrival rate* for each class of service) and the system. The collected data is filtered through an estimator for error correction and noise removal. Estimators, such as Kalman filters [77], have

been proven effective in estimating demand [75].

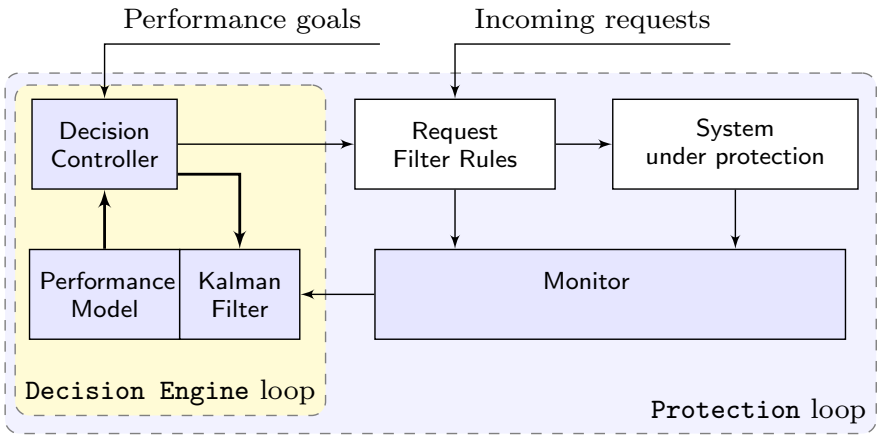



**Figure 5.3:** *Overview of the key components of the Decision Engine.*

The performance data is passed to the *performance model*, which consists of two queuing network layers. The main function of the model is to predict performance indicators if currently filtered traffic were allowed in addition to currently arriving requests. This decision is made at the scenario level (recall a scenario is a class of application requests). For modeling we are using OPERA [36], which is an updated version of the solver APERA [21] developed by one of the authors.

The *decision controller* constructs the rules used by the Reverse Proxy. At each iteration it predicts whether incoming requests would overload the protected web application. Based on the current state of the web application and the *performance goals*, a new class of requests may be filtered or allowed (filtered traffic is redirected to the Analyzer). In each iteration, the controller tries to minimize the number of classes of filtered requests. The controller relies on the performance monitor for information about the monitored environment and on the performance model for predictions.

The controller creates a filtering rule for requests to a scenario when it detects one of the following conditions:

1. The web application will be unable to support incoming traffic based on an estimation from the performance model; or

2. The nature or volume of the requests deviates significantly from the WAPP, and system performance indicators are deviating from predefined thresholds (statistical anomaly detection).

This process allows rapid reaction to prevent system overloading. Our DoS detection process utilizes both a performance model (1) and statistical anomaly detection (2) to create rules, and then iteratively fine-tunes them using the performance model.

The performance-model driven aspect is very efficient as long as the performance model is synchronized with the web application. Synchronization is lost when the theoretical model deviates from the actual application, which occurs due to the general representation of the application as a LQN rather than a heavily tuned simulation: precision is exchanged for general applicability. The detection and repair of lost synchronization is discussed in 5.2.2.3. In contrast, the statistical anomaly detection does not require synchronization with the web application, but its detection capabilities are significantly dependent on the selected measures (e.g. CPU utilization, memory allocation) and the constructed WAPP. Due to these limitations, the AD may not be able to detect traffic that would overload the web application. Filtering based on the combination of the performance model and the statistical anomaly detection improves our ability to prevent the system from overloading.

The algorithms for creating and removing rules, and the algorithm for tuning the model and ensuring synchronization, are described in the following subsections.

#### 5.2.2.1 Filtering Rule Construction Algorithm

As mentioned, the controller tries to detect DDoS attacks using the performance model; when the model is not synchronized, and statistical anomalies are detected, the AD also creates filtering rules. The approach used for AD is based on [107]; the filter construction algorithm when using the performance model is shown in Algorithm 5.1.

**Algorithm 5.1:** Filter Construction Algorithm – algorithm that finds a set of scenarios $\mathbb{C}$ which if filtered would bring the performance metrics to an acceptable level.

**input** : $L^{(u)}$ – the vector that contains the current load on each non-filtered scenario;
**input** : $PM_m$ – the vector of measured performance metrics;
**input** : $err$ – the accepted error for the model estimations;
**input** : $\mathbb{C}^{(u)}$ – the set of all unfiltered scenarios (classes of traffic);
**output** : $\mathbb{C}$ – a set of unfiltered scenarios, $\mathbb{C} \subseteq \mathbb{C}^{(u)}$, that should be filtered.

$\mathbb{C} \leftarrow \emptyset$;
**if** $\mathbb{C}^{(u)} = \emptyset$ **then**
    **return** $\mathbb{C}$;
Use OPERA to compute the estimated performance metrics, $PM_e$, for the load $L^{(u)}$;
**if** $\left|1 - \frac{PM_e}{PM_m}\right| > err$ **then**
    Tune the model for load $L^{(u)}$;
    Compute $PM_e$ with the updated model;
$L \leftarrow L^{(u)}$;
$pm_e \leftarrow PM_e$;
**while** $\mathbb{C}^{(u)} \neq \emptyset$ **and** *( $pm_e$ are **not** acceptable)* **do**
    $C^{tmp} \leftarrow$ **null**;
    **foreach** *scenario* $C \in \mathbb{C}^{(u)}$ **do**
        Use OPERA to compute the estimated performance metrics $pm_e^C$ for load $L - \{L_C^{(u)}\}$;
        **if** $pm_e^C < pm_e$ **then**
            $C^{tmp} \leftarrow C$;
            $pm_e \leftarrow pm_e^C$;
    $\mathbb{C} \leftarrow \mathbb{C} \cup \{C^{tmp}\}$;
    $\mathbb{C}^{(u)} \leftarrow \mathbb{C}^{(u)} - \{C^{tmp}\}$;
    $L \leftarrow L - \{L_C^{(u)}\}$;
**return** $\mathbb{C}$;

It first verifies that the model is synchronized with the system, by checking the estimated values of the metrics for the workload $L^{(u)}$ against the measured values (line 5). If they are not close, the model is adjusted using Kalman filters. The Kalman filter will estimate new values for the model parameters (service demands) such as the model-predicted performance metrics (CPU utilizations, throughput, response times) match those measured [22, 23, 24].

The main loop (line 10) checks the effect of filtering each scenario on the performance of the system (using the model), and selects the scenario that provides the highest increase in performance (line 14). When all have been investigated, the selected scenario is added to the list $\mathbb{C}$ of classes of service to be filtered (line 17). The loop ends when there are no more unfiltered scenarios or when the estimated performance metrics are within acceptable values.

Once a set of classes that should be filtered has been identified, the corresponding rules are added to the Dynamic Firewall. In our experiments, each class can be identified by a regular expression matching the requested URL. Adding a rule to filter the requests belonging to a class translates into adding the regular expression to the firewall. For each request, the firewall will check the URL against the rules, and, if a match is found, the request is marked as suspicious. However, more complex firewall rules can be added.

#### 5.2.2.2 Filtering Rule Removal Algorithm

The second role of the controller is to fine-tune filters or remove them when an attack appears to be over. The filter removing algorithm is shown in Algorithm 5.2.

Again, the algorithm verifies the model is synchronized with the system (line 5),

**Algorithm 5.2:** Filter Removal Algorithm – algorithm to identify existing scenario filters $\mathbb{C}$ that can be unfiltered without overloading the system.

**input** : $L^{(u)}$ – the vector that contains the current load on each unfiltered scenario;
**input** : $L^{(b)}$ – the vector that contains the current load on each filtered scenario;
**input** : $PM_m$ – the vector of measured performance metrics;
**input** : $err$ – the accepted error for the model estimations;
**input** : $\mathbb{C}^{(b)}$ – the set of all filtered scenarios;
**output** : $\mathbb{C}$ – a set of filtered scenarios, $\mathbb{C} \subseteq \mathbb{C}^{(b)}$, that are safe to unfilter.

1 $\mathbb{C} \leftarrow \emptyset$;
2 **if** $\mathbb{C}^{(b)} = \emptyset$ **then**
3  **return** $\mathbb{C}$;
4 Use OPERA to compute the estimated performance metrics, $PM_e$, for the load $L^{(u)}$;
5 **if** $\left|1 - \frac{PM_e}{PM_m}\right| > err$ **then**
6  Tune the model for load $L^{(u)}$;
7  Compute $PM_e$ with the updated model;
8 $L \leftarrow L^{(u)}$;
9 **while** $\mathbb{C}^{(b)} \neq \emptyset$ **do**
10  $C^{tmp} \leftarrow$ **null**;
11  $pm_e \leftarrow$ **null**;
12  **foreach** *scenario* $C \in \mathbb{C}^{(b)}$ **do**
13  Use OPERA to compute the estimated performance metrics $pm_e^C$ for load $L \cup \{L_C^{(b)}\}$;
14  **if** *(* $pm_e =$ **null** *)* **or** *(* $pm_e^C < pm_e$ *)* **then**
15  $C^{tmp} \leftarrow C$;
16  $pm_e \leftarrow pm_e^C$;
17  **if** *(* $pm_e \neq$ **null** *)* **and** *(* $pm_e$ *are acceptable)* **then**
18  $\mathbb{C} \leftarrow \mathbb{C} \cup \{C^{tmp}\}$;
19  $\mathbb{C}^{(b)} \leftarrow \mathbb{C}^{(b)} - \{C^{tmp}\}$;
20  $L \leftarrow L \cup \{L_C^{(b)}\}$;
21  $L^{(b)} \leftarrow L^{(b)} - \{L_C^{(b)}\}$;
22  **else**
23  **return** $\mathbb{C}$;
24 **return** $\mathbb{C}$;

using the specified workload, and tunes it using a Kalman filter if it is not (line 6).

The main loop (line 9) tries to remove filters. For each filtered scenario, it estimates the performance of the system if the scenario were unfiltered. The scenario with the smallest impact on performance is selected (line 14) as a candidate for removal. After

all scenarios have been evaluated, the algorithm checks if the estimated performance metrics when the filter is removed are within acceptable values (line 17). If the test succeeds, the scenario is added to the list $\mathbb{C}$ of scenarios to be unfiltered. The traffic for this scenario is taken into consideration for the next iteration of the main loop. The main loop exits when no scenario is found to be a viable candidate for removal (line 23) or when all scenarios are unfiltered.

**Algorithm 5.3:** Model Tuning Algorithm – algorithm that estimates new demands, when the model goes out of sync with the monitored system.

**input** : $M$ – the performance model used to estimate performance metrics;
**input** : $PM_m$ – the vector of measured performance metrics;
**input** : $L$ – the vector that contains the current workload that generated $PM_m$;
**input** : $D^{in}$ – the vector that contains the current demands for resources;
**input** : $err$ – the accepted error for the model estimations;
**input** : $max$ – the maximum number of iterations;
**output** : $D$ – a vector with demands such that the estimated and measured performance metrics are close to each other.

1 Initialize the Kalman Filter;
2 $D \leftarrow D^{in}$;
3 Use model $M$ to compute the estimated performance metrics, $PM_e$, for $L$ and $D$;
4 Initialize the Kalman Filter sensitivity matrix $H$ with zeroes;
5 **while** $\left|1 - \frac{PM_e}{PM_m}\right| > err$ **and** *max not reached* **do**
6 Update Kalman Filter sensitivity matrix, $H$, for model $M$ when using demands $D$ and workload $L$;
7 $D \leftarrow KalmanFilterEstimate(H, D, PM_e, PM_m)$;
8 Use model $M$ to compute the new estimated performance metrics, $PM_e$, for workload $L$ and new demands $D$;
9 **return** $D$;

#### 5.2.2.3 Tuning the model

When the predicted values from OPERA and those measured by the performance monitor are sufficiently dissimilar (a tuneable parameter), the model is considered *desynchronized* from the monitored system and new demands for resources are necessary. To find new demands we use Kalman Filters, which are known to be effec-

tive [22, 23, 24].

The algorithm, shown in Algorithm 5.3, is an iterative one. Each iteration executes the algorithm presented in [23] to estimate the demands (function `KalmanFilterEstimate`). We briefly summarize the algorithm here; [23] contains a comprehensive formal description of Kalman Filters and an analysis of their performance.

Initialization of the Kalman Filter (line 1) for the computations that will follow begins with setting the internal matrices to their initial values. The sensitivity matrix $H$ contains the sensitivity of observations (measured metrics) to parameters (demands). If we consider that the model, $M$, is a function of demands and workloads that produces a vector of performance metrics, then

$$H = \frac{\partial M}{\partial D}$$

When this algorithm finishes, a vector with new values for demands is found and these values will be used by the model until a new synchronization is required.

### 5.2.3 Analyzer

Suspicious traffic is redirected to the Analyzer, which is responsible for making the final decision. Our approach is inspired by the process presented by [103]: it presents a CAPTCHA test that must be passed before the request is identified as legitimate. The test is required for each request, to prevent the attacker from passing the CAPTCHA test and then triggering an automatic attack. A pre-filter drops all requests believed to have malicious intent; requests are considered malicious when a request from the same source has failed or not answered the CAPTCHA within a pre-defined time

period. The sequence diagram for the Analyzer when the request is not malicious is shown in Figure 5.4.

Once an end-user has successfully solved the CAPTCHA, they are temporarily whitelisted for a configurable length of a time at a configurable rate of requests. This limits user frustration at solving CAPTCHAs while preventing an attacker from solving a single CAPTCHA and then launching an automated attack.

The redirection process is a straightforward HTTP redirect (status code 302), which for page load requests will be handled transparently by the user's client (web browser). Asychronous requests that use HTTP as a transport layer (e.g. AJAX) can be handled differently. For example, when returning the page with the CAPTCHA query, a special header can be included, then client-side scripting used to detect the presence of this header and prompt the user to enter the CAPTCHA answer.

## 5.3 Experiments

In this section, we present experiments showing the successful mitigation of DoS attacks (Section 5.3.2) and demonstrating the benefits of using a performance model in DoS mitigation (Section 5.3.3). We begin with the experimental setup.

### 5.3.1 Experiment Environment

In all experiments we have used a *bookstore* application that we have developed using J2EE technology. The *bookstore* application emulates an e-commerce web application with the following usage scenarios:

- `marketing` – browse reviews and articles;

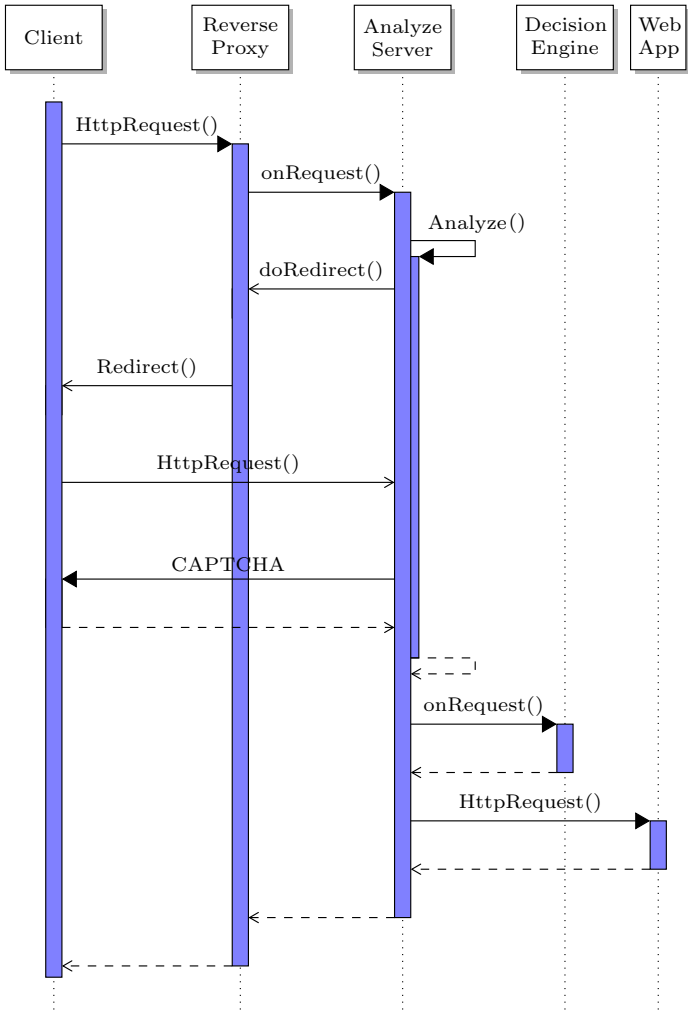



**Figure 5.4:** *Sequence diagram for traffic redirected to the Analyzer.*

- `product selection` – search the store catalog and compare product features;
- `buy` – add items to the shopping cart;
- `pay` – proceed to checkout the shopping cart;
- `inventory` – inventory management such as buy/return items from/to the supplier; and

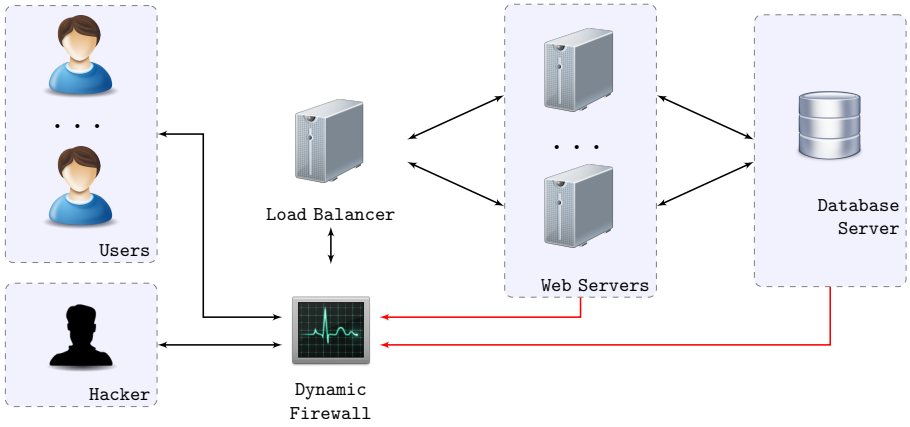


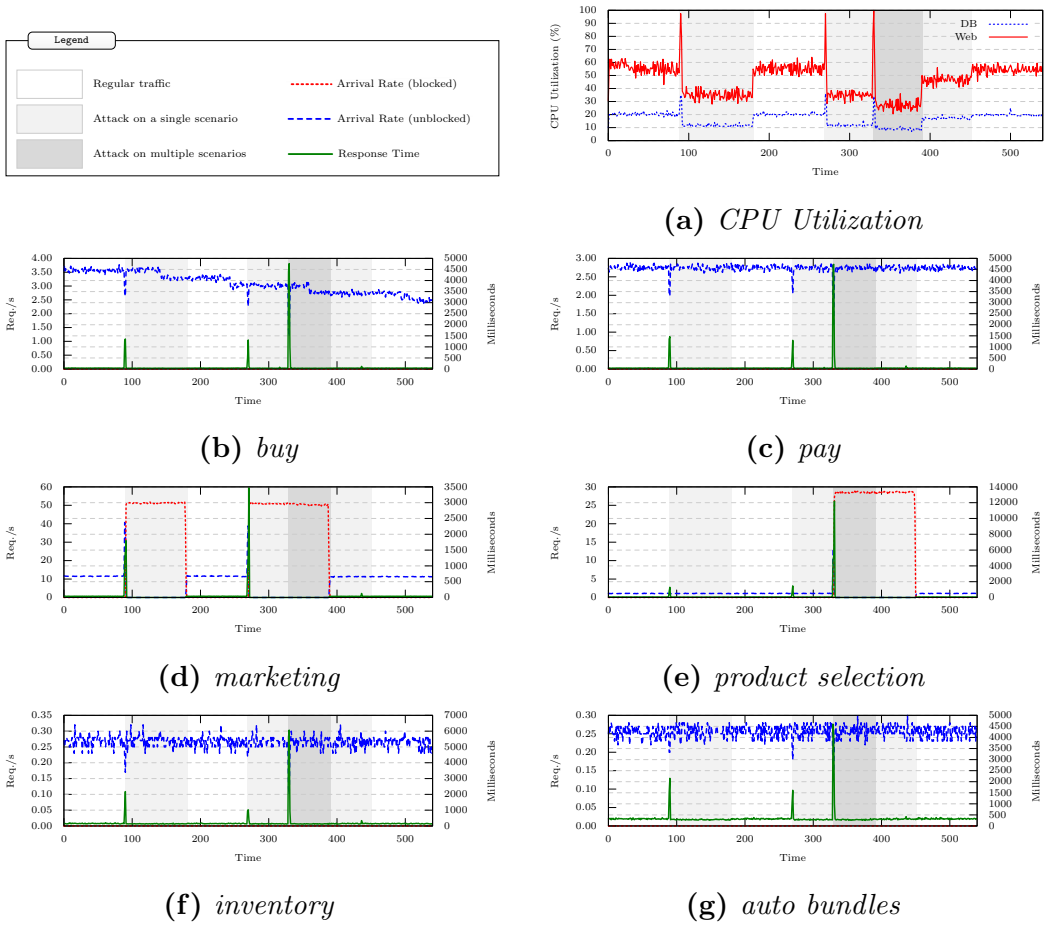

**Figure 5.5:** *The cluster used for experiments.*

- `auto bundles` – upselling / discounting system.

Each scenario performs a different mix of select, insert, and delete SQL commands. We deployed *bookstore* to three Windows XP machines: one database server (MySQL) and two application servers (Tomcat). A fourth machine hosted a workload balancer (Apache 2) to distribute the incoming web requests to the application servers. Figure 5.5 shows our deployment architecture. Once deployed, we gathered data under regular conditions to establish the WAPP. The profile for this application was based on CPU utilization of the web host and database host (expected to be between 0 and 70%), and on the response time of the application (established per scenario; for some scenarios 5 second response time is normal, while others are expected to be as low as 1 second).

Between the users and the web application, we deployed a DoS mitigation implementation. The primary focus of the experiments is our Dynamic Firewall implementation, but we deploy three other implementations for comparison purposes:

- **Dynamic Firewall** – as described in this paper, using a two-layer queuing




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *pay*

**(d)** *marketing*

**(e)** *product selection*

**(f)** *inventory*

**(g)** *auto bundles*

**Figure 5.6:** *Experiment with emulated DoS attack, using the performance model.*

network performance model combined with a Kalman filter to establish filtering rules (with a statistical anomaly detection approach running in parallel), and iteratively fine-tuning these rules based on the same performance model.

- **Dynamic Firewall (Original)** – like the Dynamic Firewall, but with the rules initially created by statistical anomaly detection only (from [108]).

- **AD-CPU** – using statistical Anomaly Detection to establish a set of filtering rules then iteratively fine-tuning the rules based on CPU utilization measure-

ments.

- **AD-CPUAR** – using statistical Anomaly Detection as above, then fine-tuning based on a combination of both CPU utilization and Arrival Rate measures.

To validate our approach, we conducted a series of experiments where we used a popular DoS attack tool (LOIC) to launch denial-of-service attacks on our sample web application[5]. We employed the four DoS mitigation implementations and monitored the response of each implementation to the incoming requests for the six web application scenarios. The evaluation is focused on the key contribution of this work, namely the decision-making on arriving traffic; therefore, while traffic is redirected to an Analyzer component, that component is not explicitly included in these experiments.

To assess the efficacy of our approach, we monitor the request arrival rate and CPU utilization metrics. The desired behavior is that traffic to the attacked scenarios entirely redirected to the Analyzer for the duration of the attack, and a reduced impact on CPU Utilization for both the database and web servers. For each experiment, there are time periods of regular traffic and time periods with an ongoing DoS attack (noted in the figures with varied backgrounds). We tracked response time (green, solid line on right y-axis), arrival rate of requests that are unfiltered (blue, dashed line on left y-axis), and arrival rate of requests that are redirected to the Analyzer and its CAPTCHA test (red, dotted line on left y-axis).

[5]In some experiments, we use a workload generator that mimics the behavior of LOIC

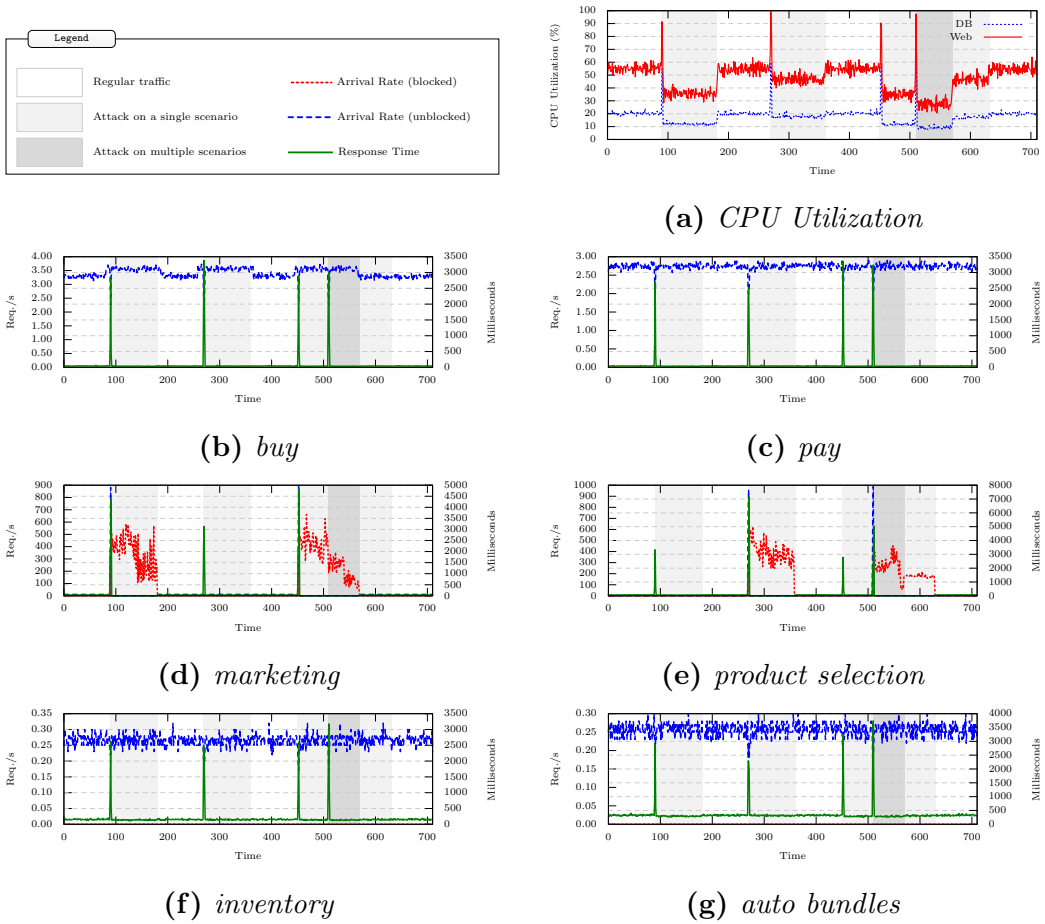
### 5.3.2 Efficacy of Adaptive DoS Attack Mitigation

We first examine the efficacy of our Dynamic Firewall approach in mitigating DoS attacks.

**Experiment 1.** This experiment demonstrates the efficacy of the complete Dynamic Firewall solution in two contexts. In the first context, a workload generator simulated three attacks: one attacking the `marketing` scenario, then a second attack on the `marketing` scenario that overlaps with the third attack on the `product selection` scenario. In the second context, the LOIC was used to launch a similar attack. The expected outcome is that the model-based adaptive algorithm effectively detects and mitigates the attacks targeting the web application, for both the artificial and the "real" attack.

*Results:* The results of Experiment 1 for the first context are shown in Figure 5.6 (with the three metrics for each of the scenarios in Figures 5.6b-g). The results show that traffic to the attacked scenarios is detected and redirected for further analysis. There is a momentary jump in incoming traffic requests before the attack is detected, then a consistently large number of filtered requests. When the attack ends, the mitigation rules are removed and normal traffic resumes. For all of the scenarios, response time jumps quickly in the seconds before the attack is detected and mitigated. Response time quickly returns to normal once the mitigation action is implemented. Figure 5.6a shows the CPU utilization on the application servers and the database. There are three spikes which correspond to the DoS attacks. Normal load was restored after the malicious traffic was filtered; based on the performance model, the filter removing algorithm first removed the filter for the marketing scenario and then for product selection.

The results for the second context, the LOIC attack, are similar (Figure 5.7): the mitigations successfully limited the impact of the DoS attack on the web application. We note in passing that minimal differences were seen between the LOIC tool and the emulation using a workload generator. □




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *pay*

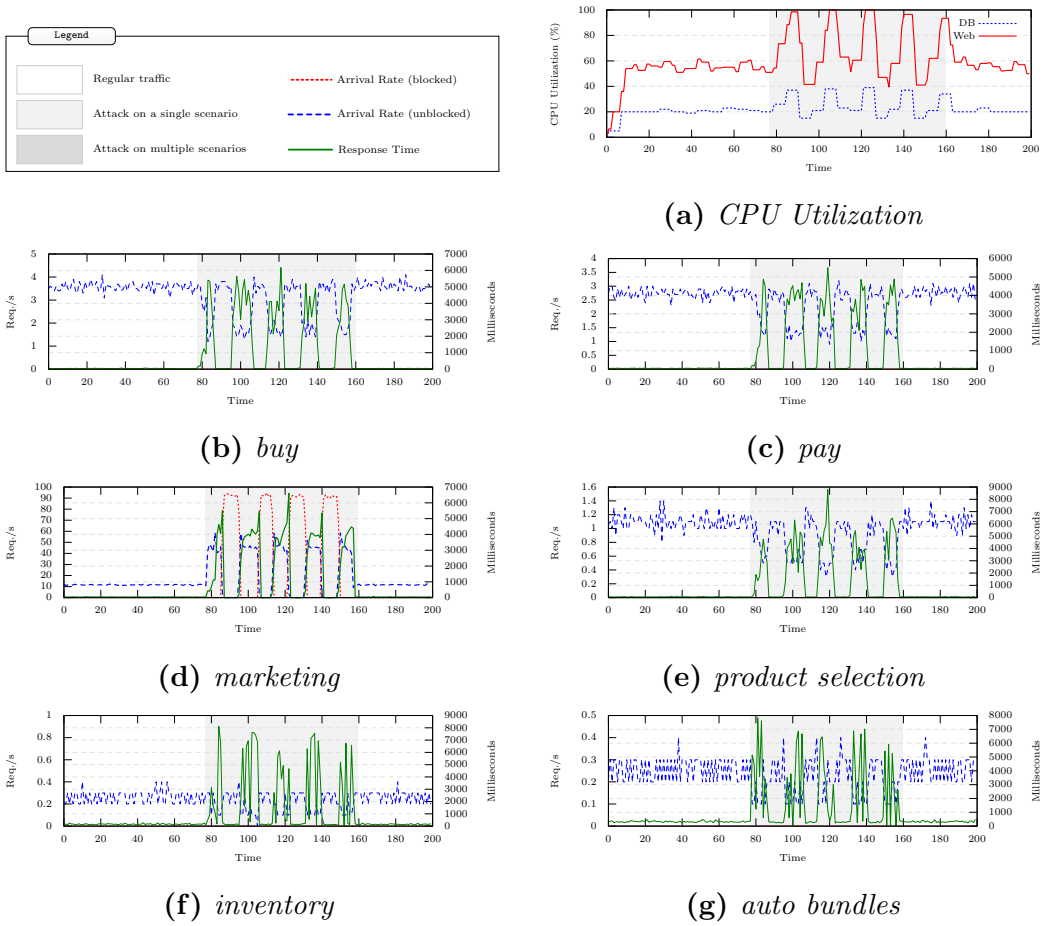
**(d)** *marketing*

**(e)** *product selection*

**(f)** *inventory*

**(g)** *auto bundles*

**Figure 5.7:** *Experiment with LOIC, using the performance model.*




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *pay*

**(d)** *marketing*

**(e)** *product selection*

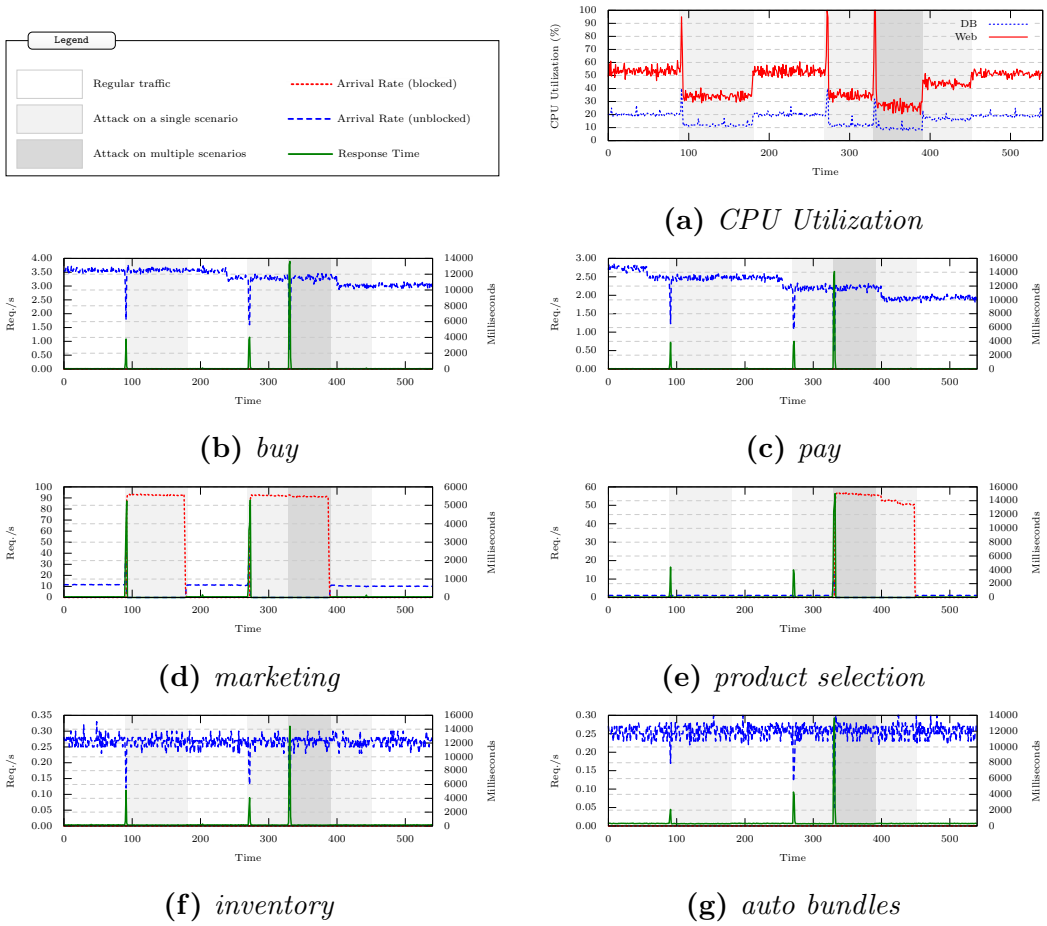
**(f)** *inventory*

**(g)** *auto bundles*

**Figure 5.8:** *Experiment with emulated DoS attack, without the performance model (using AD-CPU).*

### 5.3.3 Importance of the Performance Model

There are several elements to our effective DoS mitigation approach, the most important being the performance model. To demonstrate its importance to the overall solution, we conducted a series of experiments with and without the performance model to examine the benefits over approaches that employ statistics about the behavior of the web application. In general, approaches that rely on behavior statistics are a) often difficult to generalize to address different variations of DoS attacks, and




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *pay*

**(d)** *marketing*

**(e)** *product selection*

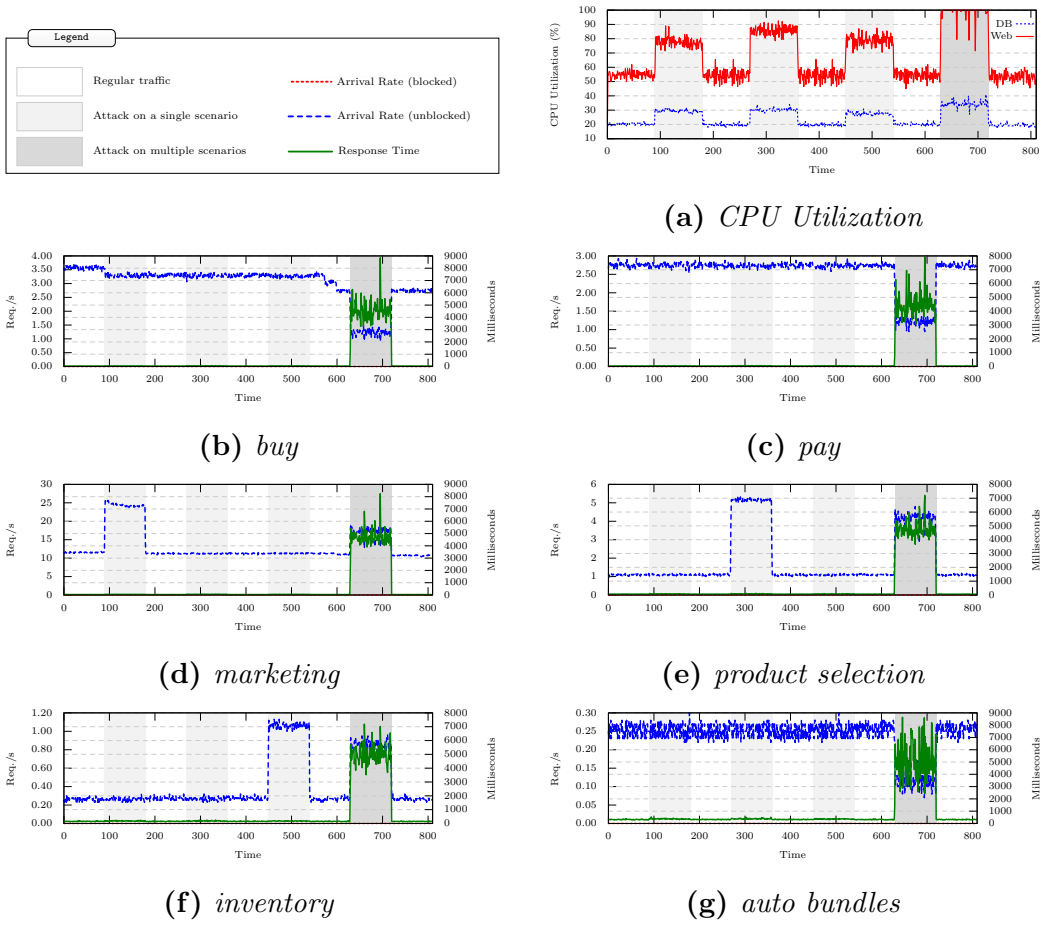
**(f)** *inventory*

**(g)** *auto bundles*

**Figure 5.9:** *Experiment with emulated DoS attack, showing anomaly detection tuned using CPU utilization and Arrival Rate (AD-CPUAR) performing similarly to when a performance model is used.*

b) utilize historical information, which leaves them vulnerable when new behavior is encountered.

In Experiment 2, we show how the mitigation fails when a statistical model that considers CPU Utilization (AD-CPU) is used to create filters. In Experiment 3, we show that manual tuning effort with the statistical model used in Experiment 2 (adding an additional metric, arrival rate, to create AD-CPUAR) can improve its




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *pay*

**(d)** *marketing*

**(e)** *product selection*

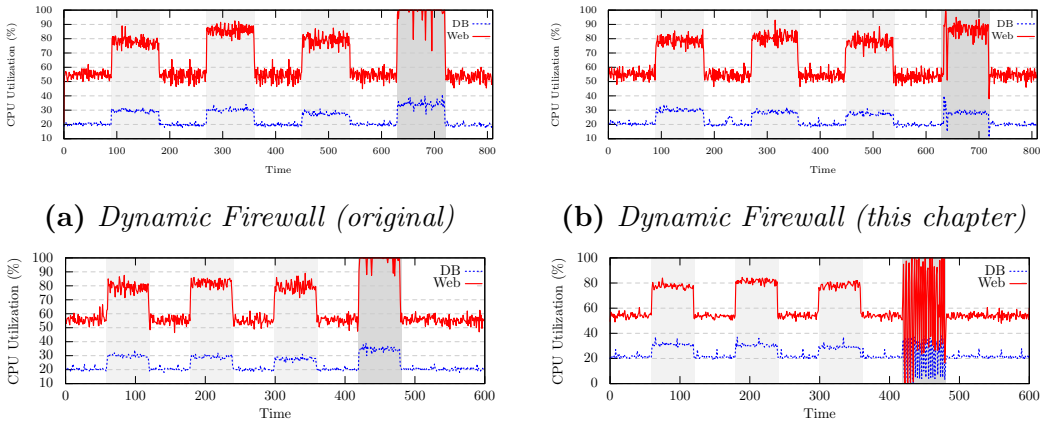
**(f)** *inventory*

**(g)** *auto bundles*

**Figure 5.10:** *Experiment emulating an advanced DoS attack (no mitigation).*

performance, to the point where it successfully mitigates attacks of the style used in Experiments 1-3. Experiment 4 shows how an attack of a more advanced style is not detected by either of the statistical anomaly detection approaches, while the performance model successfully mitigates the attack without additional tuning.

**Experiment 2.** In this experiment, we examine the impact of the performance model on the overall solution by employing the same approach to mitigating DoS attacks, except that the decision to remove rules is made based on statistical anomaly detection



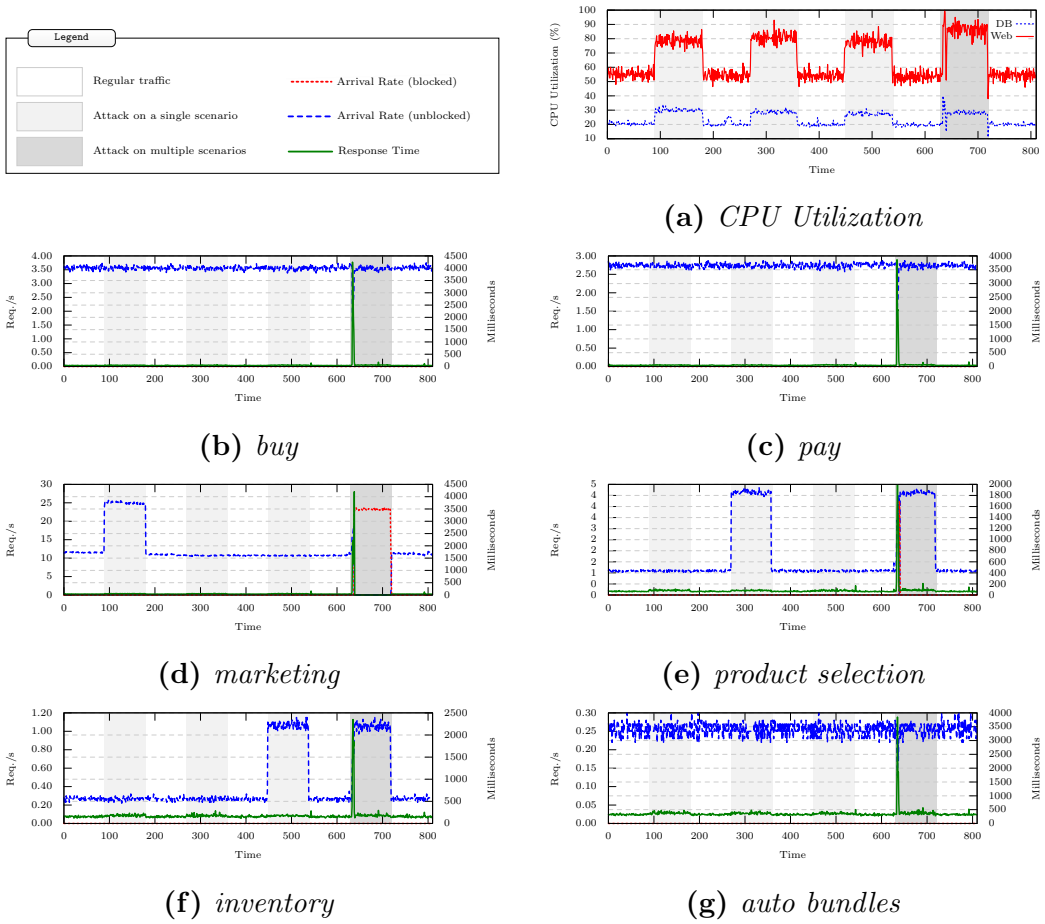

**(a)** *Dynamic Firewall (original)* **(b)** *Dynamic Firewall (this chapter)*

**(c)** *AD-CPU* **(d)** *AD-CPUAR*

**Figure 5.11:** *CPU Utilization of the web application during an advanced DoS attack for four mitigation approaches.*

using CPU utilization metrics (AD-CPU) instead of the complete performance model. We emulated a single DoS attack on one scenario, `marketing`. We expect a similar ability to detect the start of an attack, but impaired ability to identify the end of an attack.

*Results:* Figure 5.8 shows the results, again showing the three metrics for each usage scenario. The algorithm is shown repeatedly filtering and resuming suspicious traffic. Without the performance model, traffic is detected as returning to normal prematurely because the algorithm cannot predict the influence of currently filtered traffic on the overall system performance. Although only one scenario (`marketing`) is targeted, all of the scenarios experience substantially degraded performance with response times degraded by an order of magnitude. The CPU utilization plot (Figure 5.8a) shows that the application servers reached 100% utilization, a sign of being badly overloaded. □




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *pay*

**(d)** *marketing*

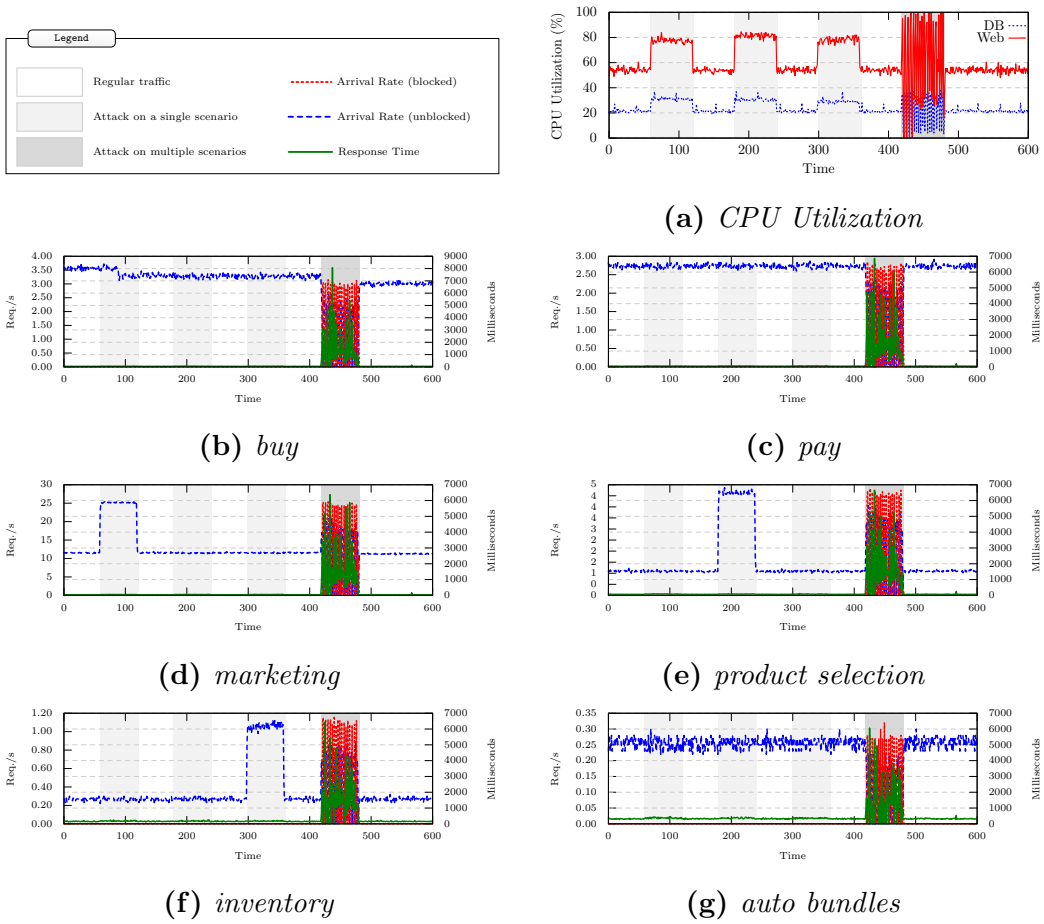
**(e)** *product selection*

**(f)** *inventory*

**(g)** *auto bundles*

**Figure 5.12:** *Details of DoS mitigation during an advanced attack, for the Dynamic Firewall using a performance model.*

**Experiment 3.** We extended the statistical anomaly approach from Experiment 2 by adding a second metric; because the effectiveness of the statistical approach depends on the selection of metrics, we used our knowledge of the deficiencies identified in the previous experiment and used a combination of CPU utilization and request arrival rate (AD-CPUAR). We used the same three-attack model: one attack on the `marketing` scenario, then a second attack on the `marketing` scenario that overlaps with the third attack on the `product selection` scenario. The request arrival rate




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *pay*

**(d)** *marketing*

**(e)** *product selection*

**(f)** *inventory*

**(g)** *auto bundles*

**Figure 5.13:** *Details of DoS mitigation during an advanced attack, when no performance model is used (AD-CPUAR).*

was measured at the reverse proxy *before* the filtering rules were applied, which allows the statistical model to incorporate some information about what would happen if the filtering rules were removed. We expect this incorporation of very basic prediction will be sufficient to address this style of DoS attack, and so expect results similar to Experiment 1.

*Results:* Figure 5.9 shows the correct behavior, namely the filtering of traffic to the affected scenario while the remaining scenarios were unaffected. We have demon-

strated the ability to tune a statistical model to correctly respond to this type of DoS attack. □

**Experiment 4.** This experiment demonstrates the non-generality of the behavior-based statistical models, where the same statistical approach may successfully mitigate several DoS attacks while failing to mitigate others. We tested four approaches: Dynamic Firewall (the approach in this paper), Dynamic Firewall (Original, from [108]), AD-CPU, and AD-CPUAR. We consider an advanced DoS attack that loads multiple scenarios with traffic that could be handled individually, but not collectively: no individual scenario is overloaded. To simulate this case, we coordinated three traffic generators. In the pre-attack phase, the traffic was increased to the `marketing` scenario, then returned to normal levels. Then, we increased traffic to the `product selection` scenario. After returning traffic to normal, traffic to the `inventory` scenario was increased and returned to normal. Finally, an attack was launched on all three scenarios simultaneously.

*Results:* The behavior of the application with no DoS mitigation deployed is shown in Figure 5.10 as a baseline. The three spikes from the pre-attack phase are visible, but the web application did not significantly deviate from the WAPP and was able to process the load. When the attack launched, the traffic levels peaked simultaneously, and the application server could no longer handle the load: we see all scenarios impacted by the overload, with high response times and high CPU utilization.

To compare an overview of the results for each of the four approaches, consider the CPU utilization graphs in Figure 5.11. Only Dynamic Firewall shows CPU Utilization within acceptable ranges; the details of the Dynamic Firewall experiment run are shown in Figure 5.12. Using the performance model, it was able to detect

scenarios that would cause application overload and redirects their traffic through the Analyzer. Two of the scenarios were filtered (marketing and product selection), which was estimated to be sufficient to meet desired performance levels. We then see that the system adapted the redirection rules during the attack; during the iterative fine-tuning, it was estimated that a scenario's traffic could be unfiltered without impacting performance beyond acceptable levels, and the product selection traffic was unfiltered. This experiment demonstrates how the defense mechanism protects the web application while reducing user inconvenience.

The only other approach to take any action at all in response to the attack is shown in Figure 5.13, where we see all scenarios (even those not under attack) being repeatedly filtered and unfiltered. We discuss possible explanations in the Discussion section. □

## 5.4 Discussion

Experiment 1 demonstrated our ability to mitigate DoS attacks, both those emulated by a workload generator and those using the LOIC.

An interesting observation from our results is that under normal conditions, arrival rate decreases as the response time increases (see Figure 5.6b), while under attack conditions the arrival rate and response time increase together (see Figure 5.6d). Higher response time yields more "think time" from the user, because the time in between two requests includes the waiting time for the reply. This is a natural behaviour when the requests are sent by human operators. However an attacker will typically send requests regardless of the response from the web server.

One of the main advantages of our approach is that while some scenarios are

redirected through the Analyzer, the rest are functioning normally, with a response time on the order of tens of milliseconds. Another observation is that the restoration is done smoothly, without oscillations or churn, which indicates the filters were not removed prematurely.

In the Dynamic Firewall, the performance model works in tandem with statistical anomaly detection. Experiments 2 and 3 examined what would happen if statistical anomaly detection (a common approach to DoS attack detection) was asked to function without the performance model. We demonstrated two types of statistical anomaly detection (AD-CPU and AD-CPUAR), one where rules were fine-tuned based on CPU utilization (i.e. it was safe to stop filtering traffic once CPU utilization was low enough) and another based on CPU Utilization and Arrival Rate (i.e. once CPU utilization was low enough and arrival rate had decreased, it was safe to stop filtering some traffic). The latter approach was effective at mitigating one type of DoS attack due to an improved ability to detect the end of an attack, but proved ineffective otherwise. These experiments also demonstrated the importance of selecting the correct metrics for statistical anomaly detection, and correctly tuning the thresholds and metrics.

Experiment 4 tested all four approaches, showing that only the approach described in this paper was capable of detecting and mitigating this type of DoS attack. However, the behavior of all four approaches merits further examination. The Dynamic Firewall (Original) and AD-CPU methods did not detect the DoS attack. Since both methods detect DoS attacks based on threshold models for each scenario, neither could detect a DoS attack on a combination of scenarios. This shows the disadvantages of using statistical anomaly detection to establish filtering rules, as models are

sensitive to tuning and parameterization, and may successfully detect some DoS attacks but not others. The basic issue is that statistical anomaly detection is based on an established baseline of application behavior, which is captured by an application under certain conditions. Generalization of statistical models of behavior for complex web applications is still an open question.

The Dynamic Firewall successfully detects the more advanced DoS attack (Figure 5.12) due to the generality of the performance model. The performance model allows us to estimate application reaction to specific input, which enables the prediction of application performance degradation before the degradation actually occurs. This predictive element is what is required to detect DoS attacks and mitigate without negative impacts.

The AD-CPUAR approach successfully detected that application is undergoing a DoS attack (Figure 5.13), but only after application performance was already degraded. It was unable to identify the malicious traffic accurately, and therefore was constantly changing the filtering rules, filtering and unfiltering scenarios whether they were under attack or not. This explains the rapid cycling of CPU utilization between 0 and 100%, as traffic is filtered and unfiltered repeatedly. This is a common issue with approaches that are not capable of prediction; the statistical anomaly detection relies on only historical and current information. By making use of historical, current, and estimated future information, the performance-model driven approach is able to respond smoothly.

All experiments demonstrate the clear advantages of using the performance model for detecting the beginning and end of DoS attacks, and the potential for using an adaptive algorithm for optimizing rules on the fly.

## Threats to Validity

The experiments conducted used realistic web applications and an actual DoS attack tool. However, with only several experiments using the complete approach there is room for additional validation.

The accuracy of the performance model is an important factor in the accuracy of this mitigation approach.

A false positive is when traffic is detected as an attack incorrectly. Because our approach does not block traffic outright but instead forwards to a CAPTCHA test, that traffic is not lost. However, the test may be annoying to users. There were no false positives in our experiments.

A false negative is when malicious traffic is not detected. In our approach, attacks are only detected when the performance of the application suffers; any malicious traffic that does not have a negative impact will not be detected, but is by definition not a true DoS attack. As our approach filters types of traffic until the application's performance is acceptable, false negatives will not impact the application.

Rather than blocking traffic outright, our approach relies on the use of a test only human users will pass (e.g a CAPTCHA). For our focus of user-facing web applications, this is sufficient to avoid dropping legitimate traffic outright. An extension to this work would consider similar tests to differentiate legitimate automated traffic from malicious automated traffic, for example pre-shared keys or other trust negotiation mechanisms.

This approach is intended to address a popular type of DoS attack, a "fast" application-aware attack where the traffic levels increase sharply. This may not be the best approach to mitigate "slow" application-aware attacks where the traffic

increases gradually over time; we have not evaluated the performance for this type of attack. As mentioned, we assume existing techniques are in place to defend against attacks not at the application level.

When we compare the effectiveness of our approach to an identical approach that excludes the performance model, we use a threshold model that considers selected performance metrics when making decisions. The selection of metrics used — in our case, CPU utilization or combination of arrival rate, but potentially also including response time, arrival rate, throughput, etc. — will impact the behavior of the system. However, regardless of the metrics chosen, they describe only the current state. Because they are not capable of prediction, there is an inherent disadvantage to using only threshold models when compared to our combined threshold and performance model approach. Though threshold models may perform as well as our combined approach on certain cases, these models would not work as well across a variety of cases and scenarios.

A DoS attack on one scenario does have an impact on the performance metrics of the other scenarios, before the attack is mitigated. Because the performance of a scenario is a factor in creating filtering rules, this may result in incorrectly filtered traffic to scenarios not under attack. This traffic will have to go to the Analyzer and the legitimate users will need to pass a CAPTCHA test.

## 5.5 Conclusions

We have demonstrated an adaptive architecture, an algorithm, and an implementation that effectively detects and mitigates application-aware DoS attacks. The approach, using a performance model and predicting the impact of traffic to create filters to

shape that impact until it is at a level the web application can handle, was able to restore normal response times to a web application that was experiencing a DoS attack. It did so efficiently at a use case scenario level of granularity that reduced the impact on legitimate traffic.

# Chapter 6

# Model-Driven Elasticity DoS Attack Mitigation in Cloud Environments

The use of software-defined infrastructure enables the addition or removal of resources (computation, storage, networking, etc.) to or from a deployed web application at run-time in response to changes in workload or in the environment. Central to this *elasticity* is the use of mechanisms that autonomically decide when to make these changes. Many approaches have been proposed and tested (see for example a recent survey [109]), including reactive approaches that establish thresholds or elasticity policies which determine when changes will be made (e.g., [110, 111, 112, 113]) and proactive approaches that attempt to anticipate future requirements using techniques like queuing models [114, 115], simulation-generated state machines [116], or reinforcement learning [117]. The focus is typically on meeting a desired service level by ensuring provisioned resources are sufficient to handle a workload, perhaps

while minimizing the total infrastructure cost [115].

The typical assumption of elasticity mechanisms is that all traffic arriving at the application is desirable. This is not always the case. For example, an application-level denial of service (DoS) attack has many of the same characteristics of an increase in legitimate visitors, especially a low-and-slow DoS attack [118], but represents undesirable load on the application. Such attacks are increasing in volume and sophistication [8], due in part to freely available tools [37, 38]. A common response to a denial-of-service attack at the application layer is to add resources to ensure the service remains available (e.g. [37, 44, 48]), which resembles elasticity. However, deploying sufficient resources to handle a major DoS attack is expensive [43], with little return on investment. Another example is a cost of service attack, where the goal is not to deny service but to increase the cost of offering a service [119, 120]; or heavy traffic from an online community (the so-called Slashdot Effect) that does not generate revenue.

In this chapter, we propose, implement, and evaluate a unified approach to enabling elasticity and mitigating DoS attacks. Rather than view DoS attempts as malicious traffic (in contrast to legitimate traffic), or even an evolved definition of "any workload beyond our capacity" [121], we define DoS traffic to be any segment of workload we cannot handle *while still providing value to the organization.* This perspective offers the opportunity to view self-management as a business decision based on a cost-benefit analysis of adding resources: if there is benefit (e.g. increased sales, ad impressions, profit, brand reputation, etc.) that exceeds the expected cost, then add resources; otherwise, manage the traffic. Workload is regarded not as malicious or legitimate, but rather as either (potentially) undesirable or desirable.

We describe three primary contributions of this work:

- an adaptive management algorithm for choosing which portions of a workload need additional resources and which portions represent undesirable traffic and should be mitigated;
- adapting a layered queuing network (LQN) model to cloud environments to enable proactive cost-benefit analysis of workload; and
- an implementation and a series of experiments to evaluate this approach in the Amazon EC2 IaaS cloud environment.

Our algorithm examines portions of the workload and assesses whether incoming traffic is desirable or undesirable. This decision is based on a runtime software quality metric called the cloud efficiency metric [120], which at its most basic calculates the total cost of the software-defined infrastructure and calculates the ratio to the revenue generated by incoming traffic (though in the general case, value can be defined very broadly). Traffic considered undesirable is handled as described in previous chapter, where instead of being discarded it is forwarded to a *checkpoint*. At this checkpoint, a challenge is issued and the user is asked to verify that they are a legitimate (and valuable!) visitor (for example, using a CAPTCHA test as in [103]). The management is completely autonomic; this avoids the known problems with the complexity of manually tuning threshold-based elasticity rules [110].

Estimating the cost-benefit potential requires a proactive approach that takes measurements from the deployed infrastructure and makes short-term predictions. Our overall approach does not prescribe which mechanism should be used; we have chosen to illustrate our approach using a LQN model solver called OPERA. We

describe the challenges in using OPERA to predict cloud behavior in practice. We present experimental results demonstrating how OPERA diverges from reality over time due to the unpredictable variability of cloud services [122], describe the modifications required to account for unexplained delays, and a second set of results that demonstrate with the modifications the LQN remained synchronized with the actual performance of the real cloud system.

We implemented our algorithm (§6.3), deployed a sample e-commerce application protected with our updated protection/elasticity autonomic manager, and tested response to a several attack scenarios: DoS alone, increase in customer traffic alone, and both combined (the common case where a site becomes more popular but also attracts negative attention). Our results (§6.4) show that the application is protected from all forms of surging traffic by adding servers or mitigating undesirable traffic, as appropriate. We also identify a limitation of our approach: the reaction time is slow enough that a temporary backlog of requests can be created, which skews calculations and leads to the temporary mitigation of desirable traffic until the model recovers. We present an example of this limitation.

## 6.1 Methodology

The goal of our approach is to treat desirable traffic (which generates business value) differently than undesirable traffic (which consumes resources disproportionate to the value created). This novel broad view of elasticity better reflects business objectives, while also addressing issues that have historically been dealt with separately. To achieve this goal, the autonomic manager must be capable of differentiating between the two, and adding or removing resources to ensure desirable traffic encounters

sufficient quality of service without over-spending, while routing undesirable traffic through an additional checkpoint. In this section, we introduce an algorithm for an adaptive manager with these capabilities.

The overall model of the approach resembles a standard feedback-loop, with an adaptive manager accepting monitoring data, using a predictive model to inform decision-making, and executing decisions autonomically using a deployment component capable of adding and removing resources. The managed system is a standard three-tier web application.

The behavior of the adaptive manager is described in Algorithm 6.1. At each iteration, a new set of metrics is observed from the managed system and its environment, including current workload, current performance, and current deployment information (which includes any ongoing traffic redirection or scaling activities). Some of this information is provided for each distinct class of traffic. For example, traffic accessing features related to browsing an e-commerce catalog might be grouped into a single class of traffic; similarly, features related to the checkout process might get their own class. These classes (sometimes called usage scenarios) allow traffic to be treated more granularly than simply looking at overall traffic to the application. A class of traffic corresponds to classes of services used in Layered Queuing models (§6.2.1).

Included in the set of performance metrics is the current cost efficiency (CE) [120], a runtime software quality metric that captures the ratio of the benefit derived from the application to the cost of offering the application:

$$CE = \frac{\text{application benefit function}}{\text{infrastructure cost function}}$$

**Algorithm 6.1:** Decision Algorithm – The algorithm used by the adaptive manager to choose appropriate actions for the managed system.

**input** : $\mathbb{C}^u$ – the set of unaltered traffic classes;
**input** : $\mathbb{C}^f$ – the set of all classes of traffic redirected to a checkpoint;
**input** : $L^u$ – the vector of current load on unaltered classes of traffic;
**input** : $L^f$ – the vector of current load on each class of traffic redirected to a checkpoint;
**input** : $M_m$ – the vector of measured performance metrics;
**input** : $svr_{cur}$ – the number of current web servers;
**input** : $svr_{max}$ – the maximum number of web servers that can be allowed to run;
**input** : $err$ – the accepted error for the model estimations;
**output** : $\mathbb{A}$ – the deployment plan.

```
1  Use LQM to compute the estimated performance metrics, M_e, for the load L^u;
2  while |1 − M_e/M_m| > err do
3      D ← Kalman(M_m, L^u, LQM);
4      M_e ← LQM(D, L^u);
5  svr_ce ← the maximum number of servers that can be added and still be cost effective;
6  A ← {do nothing};
7  if M_m violates SLOs then
8      svr ← min(svr_max, svr_cur + svr_ce);
9      n ← CalculateServersToAdd(L^u, M_m, svr_cur, svr);
10     if n > 0 then
11         𝔸 ← {add n web servers};
12     else
13         ℂ ← TrafficClassesToRedirect(L^u, M_m, err, ℂ^u);
14         𝔸 ← {redirect traffic classes ℂ};
15 else
16     set in the model the number of web servers to svr_cur + svr_ce;
17     ℂ ← TrafficClassesToRestore(L^u, L^f, M_m, err, ℂ^f);
18     if ℂ ≠ ∅ then
19         svr ← svr_cur − 1;
20         ℂ_tmp ← ∅;
21         while ℂ_tmp ≠ ℂ do
22             svr ← svr + 1;
23             set in the model the number of web servers to svr;
24             ℂ_tmp ← TrafficClassesToRestore(L^u, L^f, M_m, err, ℂ^f);
25         if svr − svr_cur > 0 then
26             𝔸 ← {add svr − svr_cur web servers} ∪ {stop redirecting traffic classes ℂ};
27         else
28             𝔸 ← {stop redirecting traffic classes ℂ};
29     else
30         n ← CalculateServersToRemove(L^u, M_m, svr_cur);
31         if n > 0 then
32             𝔸 ← {remove n web servers};
33 return A
```

Due to size constraints, the details of the cost efficiency metrics are not included in this paper, but the reader can find an in-depth presentation in [120]. To summarize, the **cost function** must capture the real cost of offering a given application on the cloud for the given period (e.g., per hour). This function excludes other costs related to the application, e.g. the cost of development, the cost of goods sold, and customer support. It is not a measure of overall profitability; rather, it captures current infrastructure costs. The **benefit function** must capture the benefit the application provides to the organization. Typically, organizations have mechanisms for assessing this benefit at least at the macro-level. The benefit may come from many sources: revenue, advertising, brand awareness, customer satisfaction, number of repeat customers, or any number of business-specific metrics. For example, a denial of service attack on an e-commerce website would reduce the value of incoming traffic (as fewer visitors would be able to make purchases), which would reduce the overall cost efficiency. If an adaptive manager were in use and were to add additional resources to handle the DoS traffic, the current value would be maintained, but the cost would increase: the overall effect would be the same.

OPERA is used to estimate a set of performance metrics (CPU utilization, response time, throughput), which are compared to the measured set of performance metrics to ensure the model is still synchronized with the system (line 2; if not, it is re-tuned using Kalman filters [24]).

On line 7, we test compliance with service-level objectives (SLOs); if any measured performance metrics are non-compliant (perhaps due to a decrease in cost efficiency, or an increase in response time), a remedial action must be taken (logic regarding cool-down times to avoid thrashing is omitted for clarity). The remedial action is chosen

from two options (adding servers because the traffic has value or redirecting traffic to a checkpoint because it does not). To decide on which action is appropriate, we call a function which uses the predictive model to estimate the impact of adding one or more web servers, up to a maximum cap on the number of servers. If adding web servers is estimated to bring performance metrics in compliance with the SLO, this solution is executed and the chosen number of servers is added (line 9). Performance metrics include the cost efficiency metric, which means that if additional traffic does not add business value, adding servers will increase the cost and therefore lower cost efficiency. In this case the approach will be rejected and 0 will be returned. The algorithm will then consider redirecting some traffic to a checkpoint instead. The algorithm considers each class of traffic separately. For example, it might be appropriate to redirect the traffic of users who access only free services, while preserving the traffic of those who pay a monthly subscription fee. Or we might give priority to the class of traffic that includes the actual checkout process, versus the customer discussion forums. The function TrafficClassesToRedirect determines which class of traffic should be redirected. The complete definition is provided in [121], but conceptually the LQN model is used to produce a set of performance metric estimates based on blocking various classes of traffic. The set returned consists of the classes that produce the best performance metrics (including cost efficiency). Traffic to these classes are redirected to a checkpoint for verification of legitimacy; this checkpoint also serves as a *speedbump* for legitimate traffic.

In contrast, if measured performance complies with the SLOs, the algorithm checks to see if we can restore classes of traffic (or if we could restore classes of traffic if we added additional servers, line 17) or if we can reduce the number of servers.

The model is used to predict performance measures (including cost efficiency) under each possible action. If the model estimates that performance metrics would remain in compliance with the SLO after restoring traffic classes, these classes are restored; similarly for removing one or more servers (line 30). It is important to note that servers can be removed even while traffic is being redirected; the elasticity function is focused on the desirable traffic, not on overall traffic. The number of servers to remove is calculated as the largest possible reduction before the model estimates SLOs would be violated.

The algorithm will terminate returning a set of actions (which may be a null set), specifying which traffic classes to redirect (or restore) and specifying the number of servers to add (or remove).

This general approach allows an administrator to specify their expected SLOs, to define a benefit function, and to define classes of traffic; however, they are not expected to write procedural rules or detailed policies. The adaptive manager is responsible for deciding both what action to take (managing traffic or adding/removing resources) and the magnitude of that action (how much traffic to manage, how many resources to add/remove). The inclusion of a general cost efficiency metric allows the adaptive manager to make decisions that reflect business objectives, rather than going to great expense to handle large amounts of traffic.

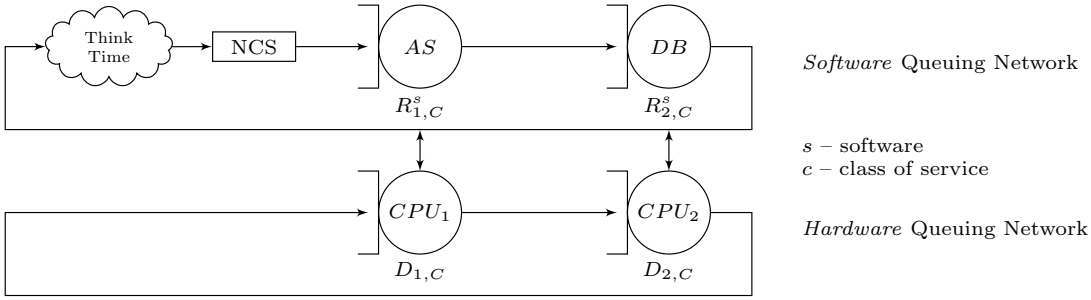



**Figure 6.1:** *Software and hardware layers in a LQN of a 2-tier web system.*

## 6.2 A Layered Queuing Network for Cloud Environments

A layered queuing network model (LQN) is at the heart of our methodology. While our algorithm is general, we implement it using a particular layered queuing network (named OPERA) which we have used successfully to model transactional web applications deployed on hardware infrastructure. In the process of validating its accuracy in cloud environments, we found that over time it diverged from reality, and that for some values (such as modeled response time) it was consistently below the actual values. This section describes using a LQN to model an application on the cloud.

### 6.2.1 Previous model

For a transactional web application such as those examined in this paper, the user interaction with the system is modeled using *classes of services* (or simply *classes*), a service or a group of services that have similar statistical behavior and have similar requirements. When a user begins interacting with a service, a *user session* is created, and persists until the user logs out or becomes inactive. We define $N$ as the number

of active users at some moment $t$; these users can be distributed among different classes of services. For a system with $C$ classes, we define $N_c$ as the number of users in class $C$, thus $N = N_1 + N_2 + \cdots + N_C$. $N$ is also called *workload intensity* or *population* while combinations of $N_c$ are called *workload mixes* or *population mixes*.

Any software-hardware system can be described by two layers of queuing networks [13, 14]. The first layer models the software resource contention, and the second layer models the hardware contention. To illustrate the idea, consider a web based system with two software tiers, a web application server (WS) and database (DB) server (see Figure 6.1). Each server runs on dedicated hardware, which for the purpose of this illustration we will limit to CPUs only ($CPU_1$ and $CPU_2$ respectively). The hardware layer can be seen as a queuing network with two queues, one for each hardware resource. The software layer also has two queues, one for the WS process and another for the DB process, which queue requests waiting for an available thread (or for *critical sections*, *semaphores*, etc.). The software layer also has a *Think Time* centre that models the delay between requests that replicate how to model how long a user waits before sending their next request.

Each resource has a *demand* (or *service time*, i.e. the time necessary for a single user to get service from that resource) for each *class*. If in this example there are two classes of service, there will be four demands for the hardware layer: each class will have a demand for each CPU. The service times (demands) at the software layer are the *response times* of the hardware layer. In our case, for class $C$, they are $R^s_{1,C}$ and $R^s_{2,C}$, and they include the demand and the waiting time at the hardware layer (we use the upper script $s$ to denote software metrics that belong to the software layer). Ideally, hardware demand is based on measured values; however, this is impractical

for CPUs because of the overhead imposed by collecting such measurements. In our approach the CPU demands are estimated using Kalman filters. Once the model is created, it is iteratively tuned, also using Kalman filters.

This model has been used to inform a variety of adaptive systems, including implementing elasticity policies [115] and mitigating DoS attacks [121, 123]. In earlier work, Zheng et al. [24] present a method for tuning model parameters for a web application. Properly tuned models have been shown to accurately estimate performance metrics [24].

### 6.2.2 OPERA in the Cloud

Despite the proven track record of this model, when we switched from modelling a private data center to modelling the public cloud (Amazon AWS EC2), we noticed significant differences between the model's estimates and the actual observed performance metrics.

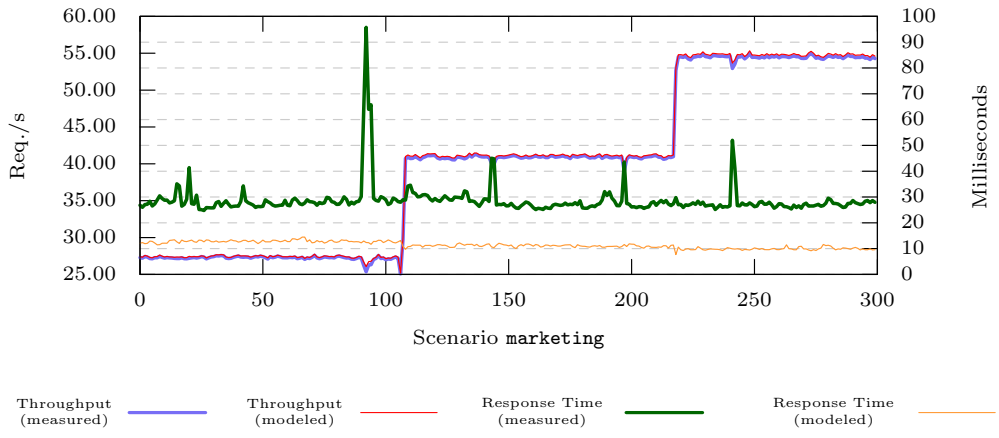



**Figure 6.2:** *Observed versus estimated values for 2 key performance metrics, showing the* `marketing` *scenario only.*

To illustrate the issue, we deployed a sample web application (see §6.4) and generated an increasing workload to a single class of service, `marketing` scenario. Figure 6.2 shows the measured and modeled response time (right Y-axis) and throughput (left Y-axis) when the workload is increased. The picture shows that the model is well-synchronized with the system for throughput, but not for the response time; the measured response time is almost three times the estimated one. As the workload increases, the synchronization procedure manages to tune the model for the throughput, but does not improve the accuracy of predicted response time.

Figure 6.3 shows the CPU utilization for the database server and the application server, both observed and predicted. The distance between the observed and predicted values is noticeable, and diverges further as workload increases, demonstrating that this metric is also not synchronized between the model and reality.

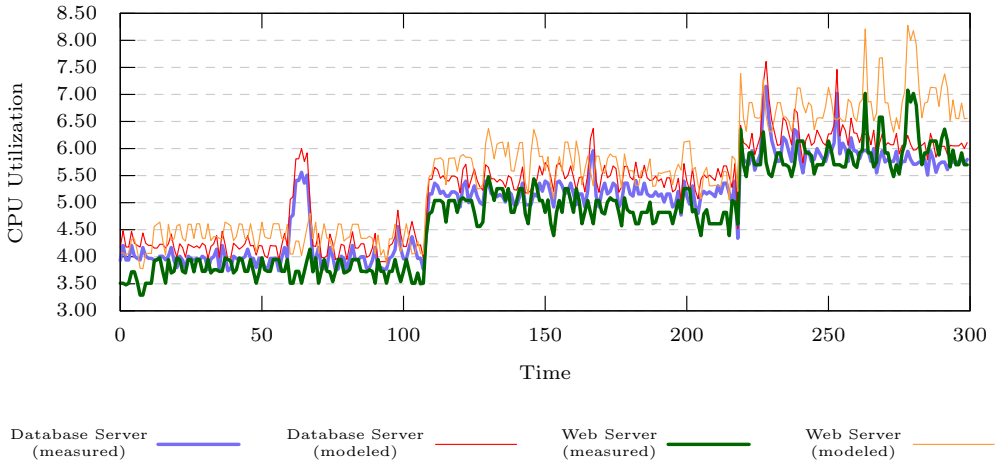



**Figure 6.3:** *The CPU utilization for the* `marketing` *scenario; the model is not sychronized with observed measurements.*

LQN models are well-established approaches to predicting the performance of web applications; after ruling out measurement errors, we concluded our particular

model is missing an unknown factor related to the known variance in EC2 [122, 124]. In order to improve the estimated metrics and reduce the modelling error overall, we added a "cloud delay centre" designed to capture all the undocumented work or delay. The cloud delay queue is shared by all classes of traffic, it has no limit, it exists at the software level, and it is the first queue encountered by incoming requests.

Following this change, we again tested synchronization and obtained substantially better results §6.4. The modified LQN is used throughout the remainder of this paper to make decisions about workload, to inform adaptive systems, and to implement elasticity policies; it demonstrated close alignment with measured values throughout this process. Other approaches that use LQN models to predict cloud behavior but do not validate their models against an actual cloud (e.g. [125],[126]) may wish to adopt a similar solution.

## 6.3 Implementation

We have implemented our adaptive management algorithm to manage applications deployed to the Amazon EC2 public cloud infrastructure, using a framework that can automatically deploy a topology in EC2 by launching all instances required (application servers, database, load balancer), install the required software on each machine (e.g. Tomcat and all dependencies, the web application and its libraries, etc.), and configure each application (for example, add the application servers to the load balancer). The framework provides an API that accepts requests to modify the deployed topology – for example, if the request is to horizontally scale the web tier, the framework can launch an instance for the application server, install all required software, and add the new server to the load balancer. The framework is

general enough to allow us to install and configure any type of application in a linux environment, once the deployer provides installation and management scripts.

Our implementation uses Amazon CloudWatch detailed monitoring to acquire performance metrics from the deployed instances. As discussed in [124], these metrics are delayed by one minute from when they are recorded. Our implementation runs the main algorithm, and acquires Cloudwatch metrics, once per minute. The arrival rate, throughput, and response time for each class of traffic are acquired from a reverse proxy on the load balancer at the same time[1]. This reverse proxy monitors incoming requests and assigns them to the appropriate traffic class (based on the URL); the classification rules can be modified at runtime if necessary. The administrator of an application is responsible for defining their traffic classes based on their business logic. Measuring response time at the load balancer allows us to focus on the component of user-experienced response time that we can control. While end-to-end response time will be higher and more variable, adapting to slow user connections is outside the scope of this paper.

## 6.4 Experiments

To evaluate the contributions of this paper, we performed a set of experiments. The first examines our modified LQN to assess its ability to synchronize with the managed application so its estimates have predictive value. Experiments 2-4 examine a sample web application.

[1]Initially, our framework used SNMP and JMX on the deployed VMs and application servers; however, these monitoring tools did not capture performance metrics for individual classes of traffic. Additionally, some values are measured incorrectly (such as CPU utilization) when running on a virtual machine in a cloud environment.

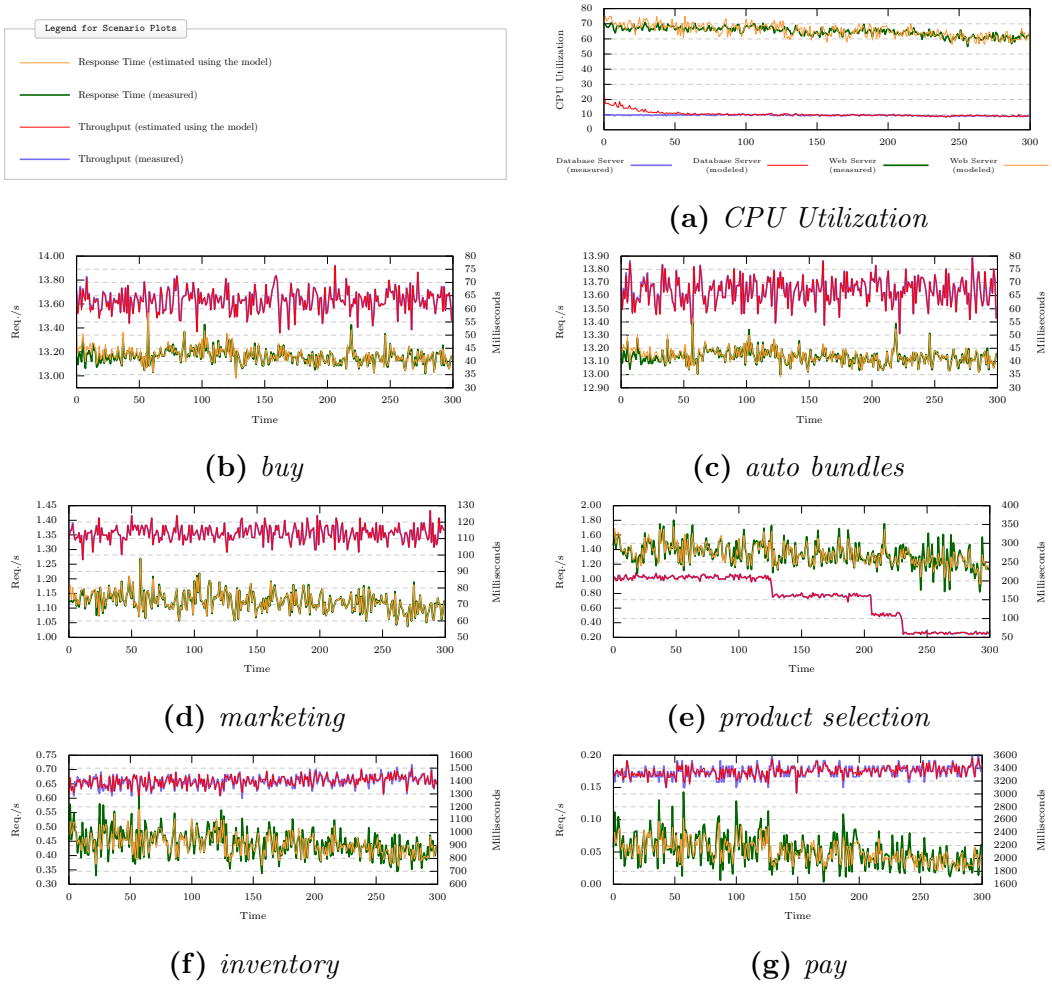
In all experiments we have used a *bookstore* application that we have developed using Java EE, which emulates an e-commerce web application. We defined six classes of traffic: `marketing` (browsing reviews and articles); `product selection` (searching the store catalog, product comparisons); `buy` (add to cart); `pay` (complete checkout); `inventory` (inventory tracking); and `auto bundle` (an upselling / discounting system). The workload used is generated randomly by a workload generator using an unevenly weighted mix of the 6 classes. Each class of traffic has a different performance impact on the deployed application.

Each experiment starts with a deployed topology, with a single application server in the web tier. The web tier is scaled horizontally by adding and removing resources. The traffic filtering approach described in [121] is used to refer undesirable traffic to a captcha to serve as the checkpoint.

### 6.4.1 Experiment 1: Synchronizing the model with public cloud resources

To verify the ability of our LQN to synchronize with reality in a cloud environment, we generated a constant workload and compared the estimated performance metrics to actual measurements at each sampling interval (Figure 6.4). We generated workload for all scenarios, though not all are shown due to space constraints. At each iteration or sampling interval, we measure the arrival rate for each scenario. We feed this workload into the model, and then solve the model to calculate the estimated metrics. If the error between measured and estimated values exceeds a specified threshold (10% in our case), we run the Kalman filters on the model in order to find more accurate values for the model parameters (this process is called *tuning the model*).

At $t = 0$, the estimated CPU utilization numbers (Figure 6.4a) for the database server are almost double the measured values. Within 25-50 iterations, the Kalman filter settles on accurate model parameters, and the difference between measured and estimated values was around 1%. Importantly, once synchronization is achieved, it is not lost. Before we added the Cloud Delay Center, this synchronization was never achieved.




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *auto bundles*

**(d)** *marketing*

**(e)** *product selection*

**(f)** *inventory*

**(g)** *pay*

**Figure 6.4:** ***Experiment #1.*** *Comparing estimated values to measured values for a selection of traffic classes for consistent workload.*

In plots 6.4b-6.4g, we show the measured response time (green line, using right Y-axis) and measured throughput (blue line, using left Y-axis) for two classes of traffic. After some initial error, the estimated values and measured values also remain within several percentage points, even through peaks and valleys.

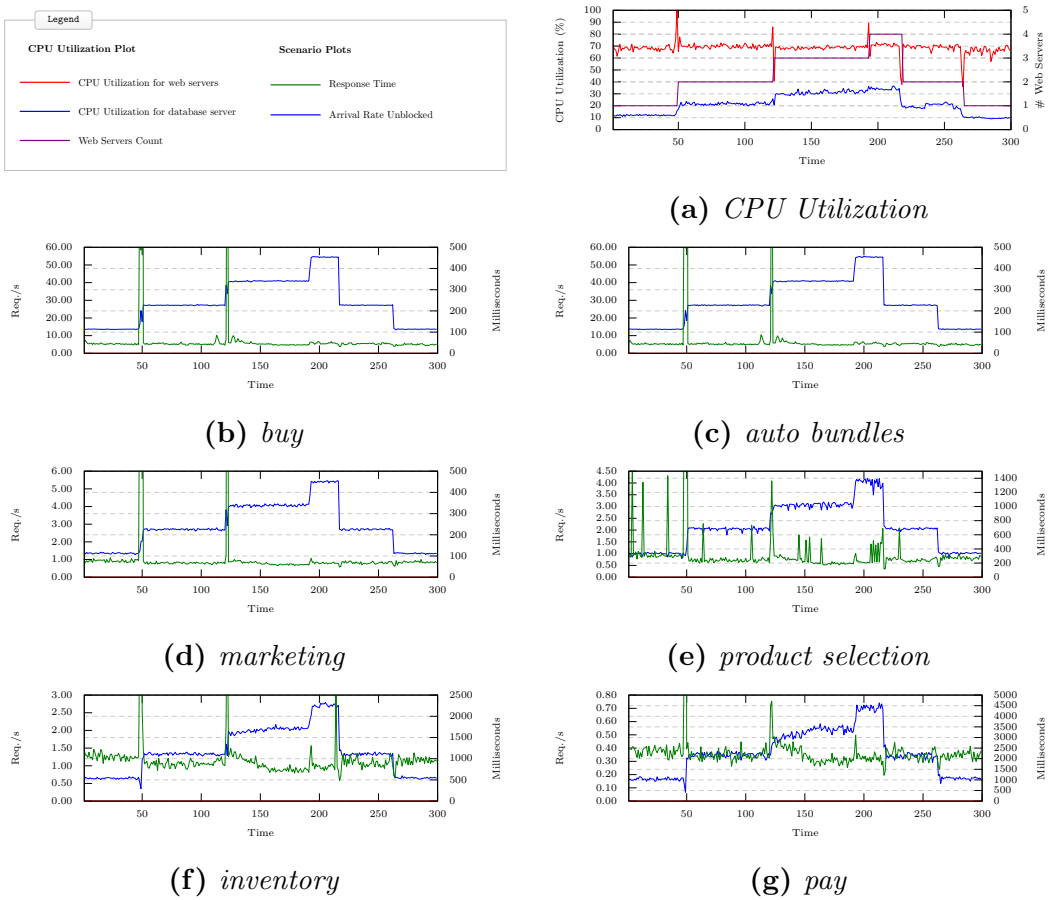
We conclude from this data that the modified LQN has addressed the challenges of modelling a cloud environment by treating the variability of the cloud as a delay center in an LQN, and modifying the demand on that delay center using Kalman filters to account for unpredictable variability.

### 6.4.2 Experiment 2: Elasticity in the public cloud

This experiment examines the adaptive management algorithm's ability to provide elasticity when overall traffic increases and decreases. We use a simplified cost metric to calculate the cost of our EC2 deployment, and use the volume of the `pay` class of traffic as our benefit function. This is roughly analogous to prioritizing checkout activity over other activities. Application-level DoS attacks are not expected to generate traffic to this traffic class, because reaching the checkout page usually requires user interaction, a valid account, and valid credit card numbers[2]. The workload mix remained constant over the experiment, and therefore so did the cost efficiency metric.

Figure 6.5 shows the measurements obtained during this experiment. The workload generated for each scenario is captured (approximately) as the arrival rate (blue line in figures 6.5b–6.5g, on the left Y-axis). We started with a baseline workload; at iteration 50, we increased the workload. The adaptive manager added a new

[2]Of course, our focus is the general approach and not optimal selection of the benefit function.




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *auto bundles*

**(d)** *marketing*

**(e)** *product selection*

**(f)** *inventory*

**(g)** *pay*

**Figure 6.5:** ***Experiment #2.*** *Increasing and decreasing the workload resulted in the addition/removal of servers, while maintaining key performance metrics at acceptable levels.*

web server (purple line in figure 6.5a, on the right Y-axis). When the workload is increased, there is a brief spike in the CPU utilization metric for the web servers (red line in figure 6.5a, on the left Y-axis), but also on the response time for each scenario (green line in figures 6.5b–6.5g, on right Y-axis). This violated the SLO, and caused the addition of a server because the cost efficiency metric remained within an acceptable range.

After workload increases at iteration 120 and again at 190, again a new server

is added each time. Notice that after adding new servers, the CPU utilization for the web servers and the response time for the various classes of traffic have about the same values as before. This suggests that the number of servers added each time (estimated using the model) was appropriate to maintain SLOs.

At iteration 220, we dropped the workload sharply. The algorithm decided that two web servers could be safely removed. Again, the performance metrics remained acceptable following this removal.

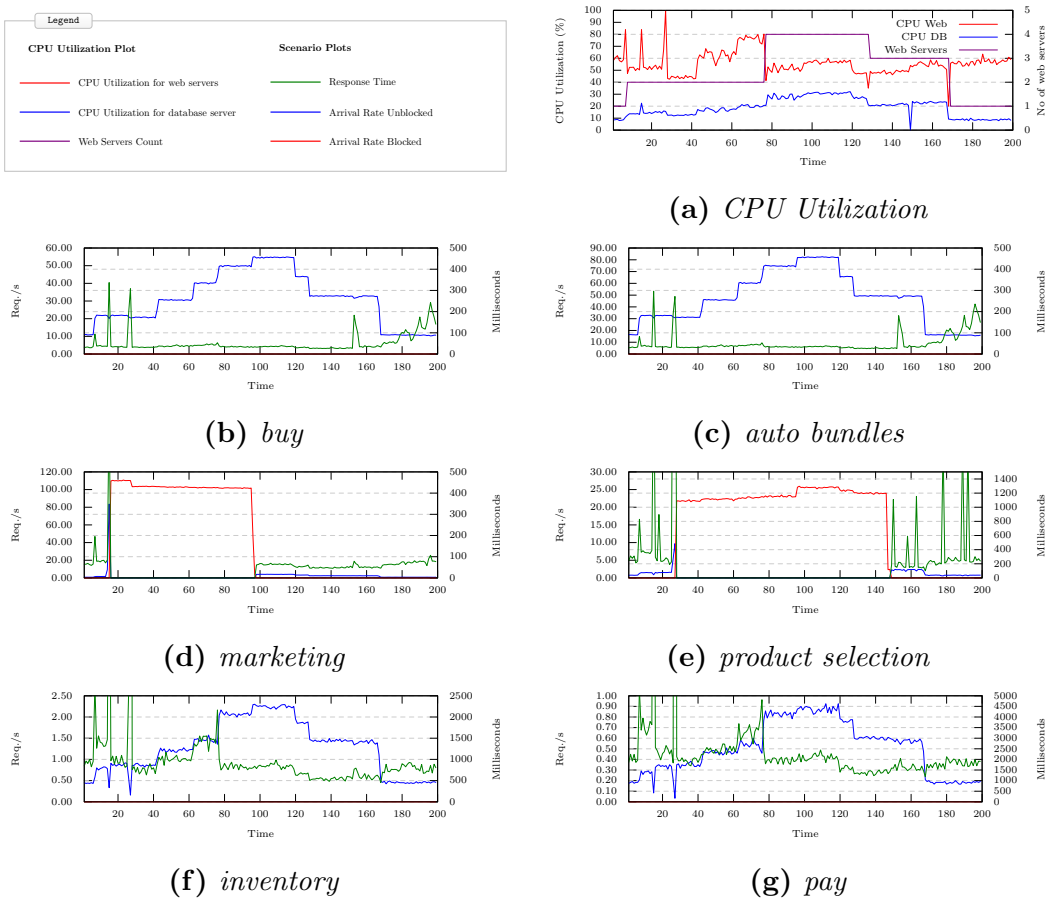
The spikes in some of the measurements are largely due to delays in actuated new servers, leading to some backlog of requests and temporarily higher response times (off the graph, in some cases).

### 6.4.3 Experiment 3: Elasticity while mitigating DoS attacks

This experiment tests one of our key contributions: achieving elasticity to maintain SLOs for our desirable traffic, while detecting and redirecting undesirable traffic. To emulate a DoS attack, we dramatically increased the volume of traffic generated to one (or more) of the traffic classes. Our measurements from this experiment are shown in Figure 6.6.

At iteration 4, we increase the workload across all traffic classes, and a new virtual machine is added. Performance degrades across all performance metrics in all classes, including the average CPU utilization. However, the addition of the new server restores performance metrics to acceptable levels.

At iteration 15 a first attack starts on a URL in the traffic class `marketing`. The arrival rate abruptly jumps from around 1.5 requests per second to close to 100 requests per second. The degradation in performance is immediate. Our algorithm




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *auto bundles*

**(d)** *marketing*

**(e)** *product selection*

**(f)** *inventory*

**(g)** *pay*

**Figure 6.6:** ***Experiment #3.*** *Overlapping DoS attacks on two traffic classes; the algorithm mitigated these attacks while adjusting the number of VMs for the remaining workload.*

provides the new metrics and workload levels to the LQN. The algorithm contemplates adding additional servers, but the increased cost is not offset by an increase in benefit, as the marketing traffic class does not generate revenue directly, and does not contribute to the benefit function. It instead determines it is necessary to redirect some traffic classes to a checkpoint and, after solving the model for various possible redirection schemes, determines (correctly) that the best course of action is to filter the requests on `marketing`.

A few iterations later (iteration 27), we emulated a second simultaneous attack on the `product selection` traffic class. Using a similar process, the algorithm once again identified the scenario under attack and redirected traffic to a checkpoint.

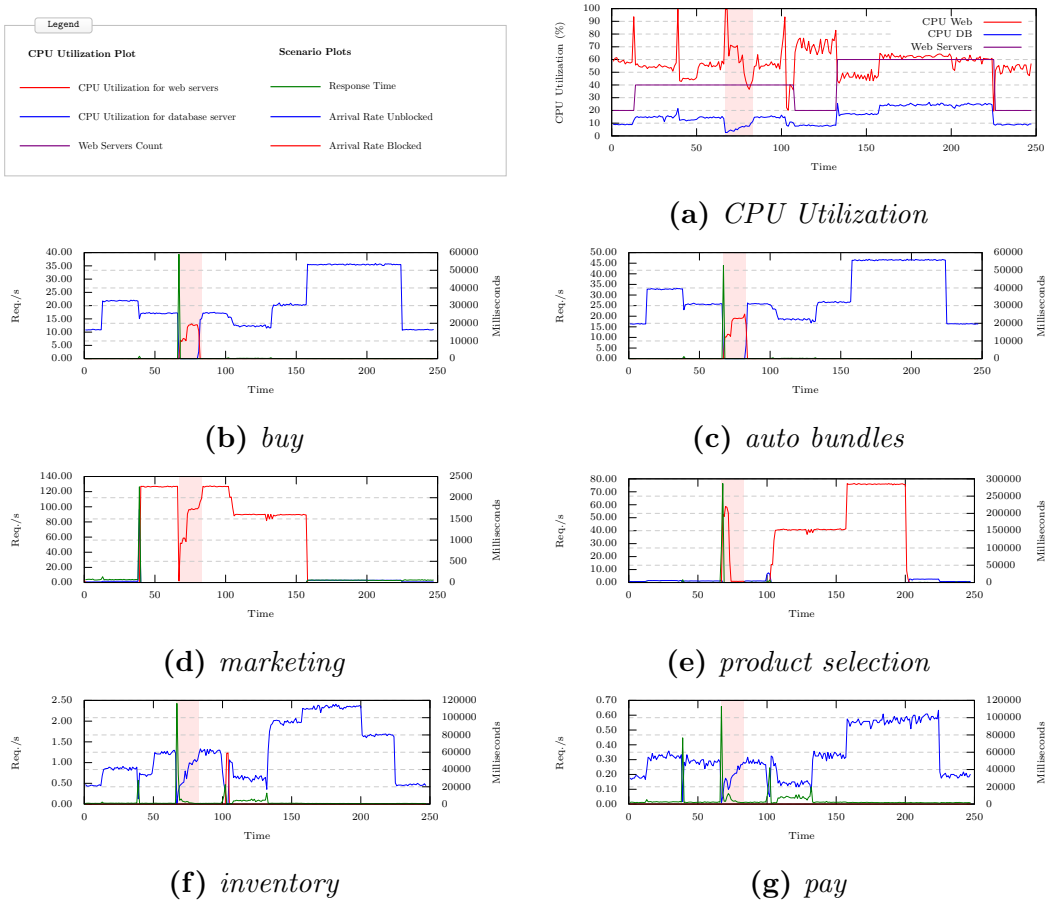
While these two attacks continued, the workload to the other classes increased for unrelated reasons. We can see response time (particularly for `inventory` and `pay`) increase. Once an SLO is violated, the algorithm decides two servers will be necessary to handle the continuing increase in desirable traffic (see purple line in figure 6.6a). This decision appears to be the correct one, as all performance metrics return to satisfactory levels (without being over provisioned).

At iteration 97, the first DoS attack (on `marketing`) is stopped, and the algorithm stops redirecting traffic for that class of traffic. By iteration 120, the temporary increase in desirable traffic also resides, and at iteration 130 a web server is removed leaving the web cluster with three machines. Note there is still an ongoing DoS attack, and removing resources is counter-intuitive, but again the performance metrics suggest this was the correct decision as they are maintained at acceptable levels. When the second attack finishes, and the last redirection is halted, the system performance metrics stay within the SLOs. We do see an increase in some response times and had the experiment continued we expect the algorithm may have added an additional server.

This experiment shows that our algorithm is able to assure the elasticity of the web application and make good decisions to achieve SLOs and maintain cost efficiency even while the application is under one or more attacks.

### 6.4.4 Experiment 4: A limitation of the implementation

The previous experiments demonstrated the strength of our approach and our algorithm. However, there is a limitation: the reaction time is slower than is sometimes required to respond appropriately. This problem can be addressed by adding a statistical model that responds rapidly using a heuristic (as in [121]). The measurements from this experiment are shown in Figure 6.7.




**(a)** *CPU Utilization*

**(b)** *buy*

**(c)** *auto bundles*

**(d)** *marketing*

**(e)** *product selection*

**(f)** *inventory*

**(g)** *pay*

**Figure 6.7:** ***Experiment #4.*** *A DoS attack causes overcorrection due to a slow reaction, and additional classes are blocked.*

When the first DoS attack starts on `marketing`, the system identified the problem

and started redirecting traffic. At iteration 67, a second attack causes a severe and rapid degradation of performance metrics across all traffic classes and servers. Due to delays between iterations, the system is unprotected and the internal queues are filled. Our algorithm analyses the data and decides that two more classes need to be protected: `buy` and `auto bundle`. The decision is correct in that it restores SLOs, but of course incorrect since no malicious traffic is targeting these classes.

When the scenario under attack is resource-intensive, it will take multiple iterations to process the backlog of DoS requests that made it through before we started directing traffic. Although traffic to `inventory` and `pay` is not redirected, there is a significant drop in the arrival rate and an increase in response time. The drop in arrival rate is normal behaviour, because normal users will not make a new request to the server until they receive the response from their previous request.

## 6.5 Related Work

There are many approaches in achieving elasticity. Companies such as Amazon, Azure, RightScale, and Rackspace offer pre-defined rule-based autoscalers. The application owners manually define rules (often threshold based) for triggering scaling out/in actions. Then, at runtime, the autoscalers monitor the application performance metrics and evaluate them against the rules. When a rule condition becomes true, the system executes the rule's action such as adding or removing VM. Some researchers argue that specifying good threshold based rules is a challenging task [110, 127].

A potential weakness of pre-defined rule system is the thrashing effect when the system constantly adds or removes VM due to a fast changing workload. To address this problem, Iqbal et al. [128] combine rule-based scaling with statistical predictive

models. Xiong et al. [129] demonstrated that an application analytic model could be used as an efficient method to identify the relationship between a number of required servers, workload and QoS.

Many researchers have designed and developed elastic algorithms without considering the cost factor; however, Han et al. [130] propose an elastic algorithm which considers both the cost and performance. An elastic algorithm will identify the application tier to scale in order to resolve the QoS issue while keeping the overall deployment cost as low as possible. A queuing analytic performance model is utilized for identification of the inefficient application tier.

The main weakness of the above approaches is that all user requests are considered desirable to the application owner. This may not be true in actual deployment environments. Many researchers and practitioners agree that DoS attacks are one of the biggest threats in the today's security landscape. Our novel approach distinguishes between desirable and undesirable traffic using cost efficiency metrics that consider not only the cost of the infrastructure, but also the business value of the workload.

## 6.6 Conclusion

This paper presented a model-driven adaptive management architecture and algorithm to scale a web application, mitigate a DoS attack, or both, based on an assessment of the business value of workload. The business value is measured through an efficiency metric as a ratio between the revenue and cost. The approach is enabled by a layered queuing network model previously used to model data centers but adapted for cloud. The model accurately predicts short-term cloud behavior, despite cloud variability over time. We evaluated our approach on Amazon EC2 and demonstrate

the ability to horizontally scale a sample web application in response to an increase in legitimate traffic while mitigating multiple DoS attacks, achieving the established performance goal. We also showed the limitation of the approach which can be overcome through further work.

# Chapter 7

# Model Identification Adaptive Control for Software Systems

The growth in software complexity and the increase in its operational and evolution cost has led to the need for self-managing or autonomic systems [131], and later to the field of self-adaptive software systems. In essence, a self-adaptive system senses the changes in the operating conditions and in the environment and adjusts its structure and behavior to meet its goals in the presence of those changes. A reference MAPE architecture [131], that consists of Monitoring, Analysis, Planning and Execution components, allows the design and the implementation of an Autonomic Manager that manages the software system.

Despite major progress in the field, there are still substantial challenges in designing and implementing self-adaptive systems. Limited efforts have been invested in developing an engineering methodology and formal mathematical foundations, which makes the development and the verification of the self-adaptive systems tedious and time consuming. A systematic method would primarily enable the automation in

designing self-adaptive systems, but also the effective verification of the automatic adaptation cycle. A design space methodology has been proposed by Brun et al. [132] to guide the designer along several dimensions, including identification, observation and control. However, this effort is subject to the peculiarities of individual interpretations and implementations.

Control theory has been proposed as a foundation formalism. The control theory is based on a formal model of the system, specified in a canonical form, from which a controller (or Autonomic Manager[1]) is synthesized based on the goals of the system. Initial steps in using control theory as foundation for designing and implementing self-adaptive systems have been studied in [133] for web control and memory management or in [132] for admission control. These approaches assume a *static linear model* of the system and a *static controller*. This static assumption limits the efficiency of the controller because software systems are highly dynamic and volatile; their models change at runtime and over time the controller may be based on the wrong assumptions. The design and implementation of adaptive systems become even harder for applications deployed in the cloud. The lack of transparency and control on the environment adds to the uncertainty of the models and impedes the design of the autonomic manager. As a result, most of the tools and frameworks available in industry for designing autonomic systems [134, 135, 136] are simple rule-based systems ("ON condition, DO action") that leaves the practitioner to do all the hard work in designing, implementing and verifying adaptive systems.

In this chapter, we propose a robust adaptation architecture and method: model identification adaptive control, MIAC. MIAC identifies the non-linear model of the

[1]The terms "controller" and "Autonomic or Adaptive Manager" will be used interchangeably hereinafter.

software system at runtime and then, dynamically, synthesizes the controller based on that model and the goals of the application. The model is a layered queuing model (LQM) that captures the quantitative relationships between different software and hardware components. The model is based on two layers of queues and is a type of a model known to software performance engineers. We focus on performance and cost, two important qualities of the software deployed in cloud but the method can be extended to any quantifiable property of the software. The non-linear LQM is then linearized around different operational points, to obtain a *control theoretic model*. The structure of the control theoretic model is different than the LQM and captures the causality among the control points (things we can automatically change at runtime) and the goals of the system. The control model is then used to study the *properties of the system* and to *synthesize and tune* a runtime *controller* for the software system.

This chapter makes four distinct contributions towards the advancement of the domain of self-adaptive software systems on the cloud.

- We introduce the *model identification adaptive controller* (MIAC). To the best of our knowledge, the proposed blend of software specific performance models and control theoretic approach is original.

- We create the *mathematical apparatus* that comes along with the MIAC architecture. This enables a degree of formalization that allows us, on one hand, to easily and systematically design controllers of different purposes by experimenting with their parameters (inputs, outputs etc.). On the other hand, it also allows us to easily verify the effectiveness of the controller.

- The third contribution is the ability of the controller to provide *multi-*

*dimensional* adaptation strategies. In practice, this means that the adaptive manager is able to control a number of the system's outputs (e.g. response time and throughput), monitor a number of the system's state metrics (e.g. CPU and memory consumption, disk I/O etc.) and, eventually provide a complex adaptation strategy by simultaneously changing a number of the system's configuration parameters (e.g. number of threads and servers, network bandwidth etc.).

- The final contribution is specifically on the linearization of the LQM model. Conventional methods linearize LQM once in the beginning and then retain the same linear model throughout the system's lifecycle. However, due to the nature of software, the linear model is bound to become outdated, in which case the error in prediction would become exaggerated and the adaptive system would often fail. In our method, the novelty is that we *linearize multiple times*; when the error of the linear model exceeds a particular threshold, we relineariaze and calibrate the controller accordingly.

The remainder of the chapter is organized as follows: Section 7.1 introduces the background and related work. Section 7.2 presents the proposed architecture and the proposed method. Section 7.3 presents the experimental results that validate our approach. The conclusions are presented in Section 7.4.

## 7.1 Background and Related Work

Adaptive systems have got a great deal of attention lately and there have been several road map books [137, 138] that identify the importance of feedback loops and their

design. At the conceptual level, the feedback loops follows MAPE architecture [139], but at the implementation levels there are many variations. A prevalent one is based on control theory [132] and has been used for some time. Hellerstein et al. [92] introduced several case studies where control theory has been used for controlling the threading level, memory allocation or buffer pool sharing in commercial products such as IBM Lotus Notes and IBM DB2. Other researchers and practitioners have published results on control theory for computer power control [140], thread and web cluster management [141], admission control, video compression [142]. In the above examples, authors used feedback loops to control quantitative metrics from the categories of performance, cost, energy, and availability. Controllers are static, created and tested at design time. Our method builds and tunes the controller at runtime and as far as we know this is the first attempt in the realm of software.

Control theory approaches revolve around three main stages for designing and analyzing a feedback loop: *defining a linear and static model* of the controlled system, *analyzing the properties* of the model, *designing a static controller*. Defining the *model* starts with identifying the *outputs* that need to be controlled, $y$; the *commands*, $u$, that can control the outputs of the system; and the internal *state variables*, $x$ ($x$ can be seen as a link between $u$ and $y$). Although the performance in software is more accurately modeled with non-linear models, many authors use linear models due to their simplicity. For example [92, 143, 144, 142, 145] use linear models. Because the models are linear and valid only around the linearization point, the controller designed based on the linear model will likely be valid around that linearization point as well and will not reject a large spectrum of perturbations. This is a well known problem in the control of non-linear systems and often control switch approaches [146]

are employed to switch between many statically designed controllers. In this paper we consider the system non-linear, represent it with a non-linear model and then linearize it at runtime, around the operational points. In this case, the system is modelled as series of linear models and controllers.

One of the most important reasons for having a model is to study the properties of the system it models and then to design the controller. In control theory, the significant properties are *stability*, *controllability*, *observability*. *Stability* is probably the the main *raison d'tre* of control theory. In simple terms, stability means that for bound inputs (commands), the system will produce bound state and output values. Examples of stability studies in control of software and computing systems have been presented in [92, 145]. *Observability* is the property of the model that allows finding, or at least estimating, the internal state variables of a system from the output variables. This property is important from pragmatic point of view. In a real system, it is impossible, hard or impractical to measure all state variables. On the other hand, the commands and outputs are easier to measure. Examples of how to use observability and how estimate software performance parameters for applications deployed across multi-tiers and using Kalman filters are presented in [147]. Other authors have used other techniques to estimated runtime model parameters [148, 149] on the same assumption that the system was observable. The concept of controllability (or reachability) describes the possibility of driving the system to a desired state, i.e. to bring its internal state variables to certain values [131]. This property ensure that we can design a controller. We use the concept of observability in our paper when we estimate the performance model in cloud. In this aspect we extend [147] for cloud environment. Our goal is to achieve controllability across a large design space and

for any model and controller we design and to make the system stable. Although one can synthesize empirical controllers [150, 136], we follow classic control theory to build optimal controllers [137, 151].

Models can be composed, that is, if they are connected in series, parallel or through a feedback loop, a resultant composed model can be computed from the individual models. If the models are linear, then the resultant model is linear, too. In [152], the authors showed, on web service performance case studies, how web services models can be composed in series and in parallel. It is interesting to note that by composing web service models, the properties of the individual models (stability, controllability, observability) are not necessary transferred to the composed model. In this paper, we use the composition of linear models when analyzing the properties of the feedback loop.

## 7.2 Model identification adaptive controller

For the purpose of this paper, we consider the software system as grey box, as shown in Figure 7.1. This implies that while we don't know all details about the internal structure of the deployed software system, we have access to its internal *state variables* ($x$). State in the context of cloud-deployed software may refer to performance measurements, usually gathered by a monitoring system, such as CPU or memory consumption, disk I/O, and so on. In addition to the state, the *model* representing the system also includes *outputs* ($y$) that need to be controlled and the *commands* ($u$) that can control the outputs of the system). The outputs of the system refer to the observable and measurable results of the system's execution, for example its response time, deployment cost, throughput, availability etc., and upon which we

can understand whether the system operates normally or not. The outputs are usually captured in the system's Service Level Agreement (SLA). The commands refer to any adaptive and corrective actions we may take to rectify abnormal situations and bring the system back to healthy operation levels. These actions may include changing the bandwidth, the number of web or database servers in the cloud topology, or the number of corresponding threads in the deployed application. Finally, we have the perturbations ($p$), corresponding to changes in the environment that are unknown and uncontrollable.

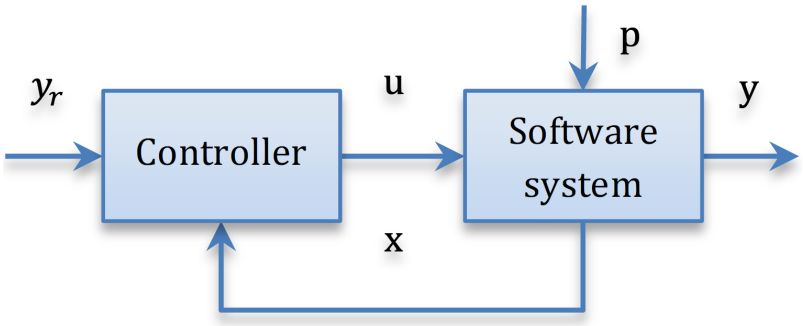



**Figure 7.1:** *State feedback control.*

Assuming we have a software system model in place, along with the controllability property, it is suggested that if we wanted to keep $y$ at a predefined value $y_r$ (the goal), we can do that by computing a command $u$ (*feed-forward control*). In practice, this is not possible for two main reasons: (a) the model is an inaccurate representation of the real system, (b) there are perturbations, $p$, that affect the system. A feedback loop implies adding a new software component, a *Controller* (or Adaptive Manager), that is fed back with state information $x$ from the system.

The controller and the feedback will compensate for modelling errors and for perturbations. The controller can also stabilize unstable systems and achieve additional

quality control criteria. The main difficulty for the above architecture is identifying the model and then synthesizing the controller that compensates for perturbations and achieves the goals.

The method we propose here is based on control theory, more specifically on synthesizing a controller, using linear quadratic regulators [153] (LQR), based on a model of the system. Given a system, LQR takes as inputs an objective function and the state of the system and computes/optimizes the inputs $u$ in the system. LQR is a static regulator, so we extend this methodology to deal with dynamic time variant non-linear models that better characterize software systems. The following sections present the general methodology and architecture and then present the design of the controller.

### 7.2.1 Overview

To deal with the uncertainty, we propose a runtime model identification adaptive controller. Its architecture is illustrated in Figure 7.2. The flow of data and control is further presented in Algorithm 7.1. The algorithm requires as input three sets $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{U}$. $\mathcal{X}$ contains the names for the system parameters that are to be monitored and will represent the state of the system. $\mathcal{Y}$ contains the outputs of the system, which will determine whether the system operates normally or not. $\mathcal{U}$ contains the set of resources or parameters, e.g. number of threads or servers, which we can change to bring the system back to a healthy state. Sets $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{U}$ are *nominal* sets, meaning they only specify *what* is to be included, based on which the actual measurement vectors $x$, $y$, $u$ are generated. Apart from these three sets, the algorithm requires as input the original non-linear model, $LQM_0$, so that we know how the performance

of the system is modelled and based on what parameters.

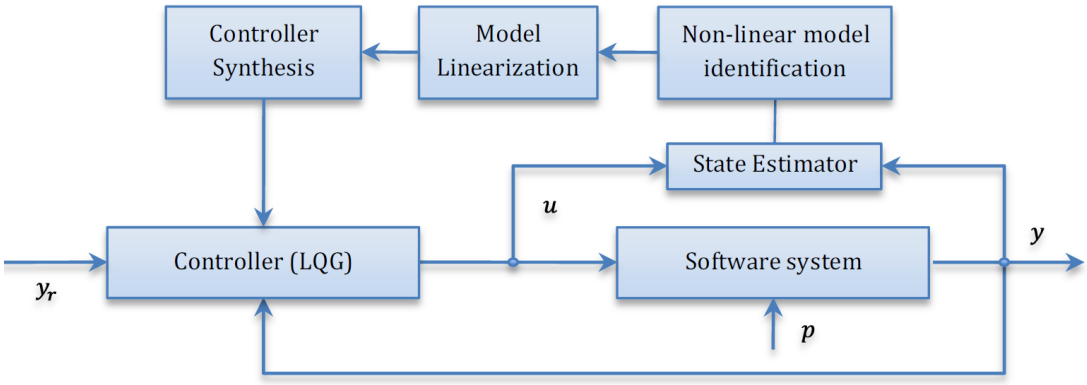



**Figure 7.2:** *Model Identification Adaptive Control Architecture.*

**Algorithm 7.1:** Model Identification Adaptive Controller (MIAC)

**input** : $\mathcal{X}$ – the set of system state parameters to be monitored;
$\mathcal{Y}$ – the set of system outputs
$\mathcal{U}$ – the set of possible commands
$LQM_0$ – the original non-linear model

1 **while** *TRUE* **do**
2  Extract current state $\rightarrow [x_c, y_c, u_c]$;
3  **if** *linear model not accurate* **then**
4  **if** *non-linear model not accurate* **then**
5  Rebuild non-linear model $\rightarrow LQM$ ;
6  Set $[x_{op}, y_{op}, u_{op}] = [x_c, y_c, u_c]$;
7  Linearize $LQM \rightarrow [A, B, C, D]$;
8  Design optimal controller for linear model $\rightarrow [K, k_r]$;
9  Controller produces adaptive commands $\rightarrow \Delta u$ ;

In step 1 of the algorithm, and illustrated in Figure 7.1, it can be seen that the monitoring and control of the system by MIAC is a closed loop. In step 2, we extract the current state of the system. Vector $x_c$ includes measurements for the system's current state as they come from a monitoring service, vector $y_c$ contains the current measurements for the system's outputs and vector $u_c$ is the current configuration of the system, in terms, for example, of the software configuring parameters and the

cloud resources of the deploying topology. In step 3, we check whether the current linear model is still accurate. If not, then, we activate the upper loop of Figure 7.1. In step 4, we check if our non-linear model is still in sync. The reason for this check is that if we have to redefine our linear model, it has to be based on an accurate non-linear model. If the non-linear model is out of sync, then, in step 5, we rebuild an $LQM$, using the parameters from $LQM_0$. The details of this step are outlined in Section 7.2.2. Having established that we need to redefine our linear model, in step 7, we set a new operational point $[x_{op}, y_{op}, u_{op}]$ (further definition at Section 7.2.3), which is the current state of the system as extracted in step 2. In step 8, the $LQM$ is linearize, a process which is detailed in Section 7.2.3. Based on the linear model, in step 9, we design an optimal controller, as described in Section 7.2.4. The controller is now ready to operate in the lower loop of Figure 7.1. By monitoring the deployed system and comparing its behavior against a set of goals $y_r$, in step 11, the controller can issue a set of commands $u$ to rectify any problematic situations.

### 7.2.2 Non-linear performance models in clouds and their identification

In our previous work, we have shown that performance of software systems deployed in clouds is highly non-linear [108, 105]. In this section, we will reiterate and summarize our previous findings to better illustrate and motivate the construction of the non-linear model of $LQM$.

Consider a three-tier application, such as the one depicted in Figure 7.3, deployed in cloud (e.g., Amazon EC2). Such an application has one or more database servers, a set of worker application servers (or web servers) and a load balancer to distribute

the user requests among the workers.

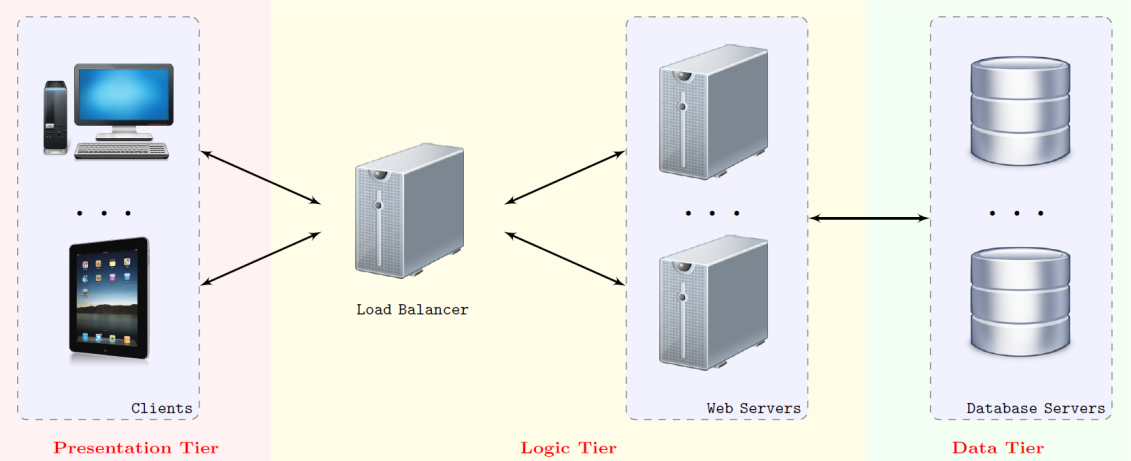



**Figure 7.3:** *A typical three-tier application.*

Performance of such applications is in general modelled as a layered queuing model (LQM), with one layer representing the software and one layer modelling the hardware. Mathematically $y(t) = LQM(w, u, x)$, where $y$ is the output, $w$ is workload and $x$ is the vector state that can represent all the intrinsic relationships between software elements. In general, given a software application deployed on an infrastructure, building a structure of the LQM model and then tuning its parameters at runtime is feasible and accurate.

Figure 7.4 shows how the response time of the application server depends on the number of users accessing the system. The data has been gathered by repeatedly generating requests for the same URL simulating a linearly increasing number of users. The web application has been deployed on a cluster with a single application server (Tomcat) and a single database server (MySQL). The plot shows that when there are approximately 30 users, the response time starts to increase abruptly; this is due to the CPU for the application server that is close to saturation point.

For these experiments, we have modelled the response time (i.e. the output $y$ of the system) as an LQM over the number of users (the workload $w$). The estimates of the model are shown with dashed line in the plot from Figure 7.4. As it can be seen, the model estimates the measurements for the two metrics almost perfectly (less than 1% of error in average).

When the operational conditions change (a change in the topology, the workload intensity or the structure of the workload), the model becomes outdated and need to be retuned. In the case of our model, this tuning happens with the use of Kalman filters [121]. Prebuilding all the models at the design time is infeasible because of the large number of possible changes and combinations of changes. Therefore, runtime retuning of the model is more efficient and, therefore, preferable. The retuning phase and the overall rebuilding of the LQM occurs in step 5 of Algorithm 7.1.

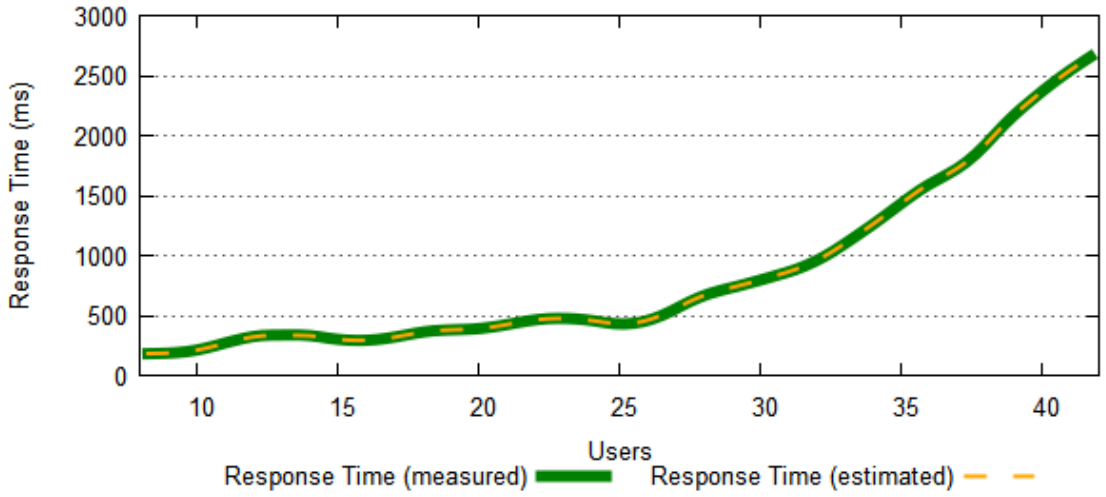



**Figure 7.4:** *Response time when the deployed topology contains only one application server.*

In order to examine the effect of change, we repeated the experiment, using two application servers and keeping everything else the same (Figure 7.5). This deployment is able to handle up to 60 users while maintaining a response time

under 500 milliseconds. At around 60 users the average CPU utilization of the two application servers is close to 100% and this results in a sharp degradation of the response time. By constantly tuning the model with Kalman filters, we were able to always have very accurate estimations of the performance metrics (dashed line in Figure 7.5).

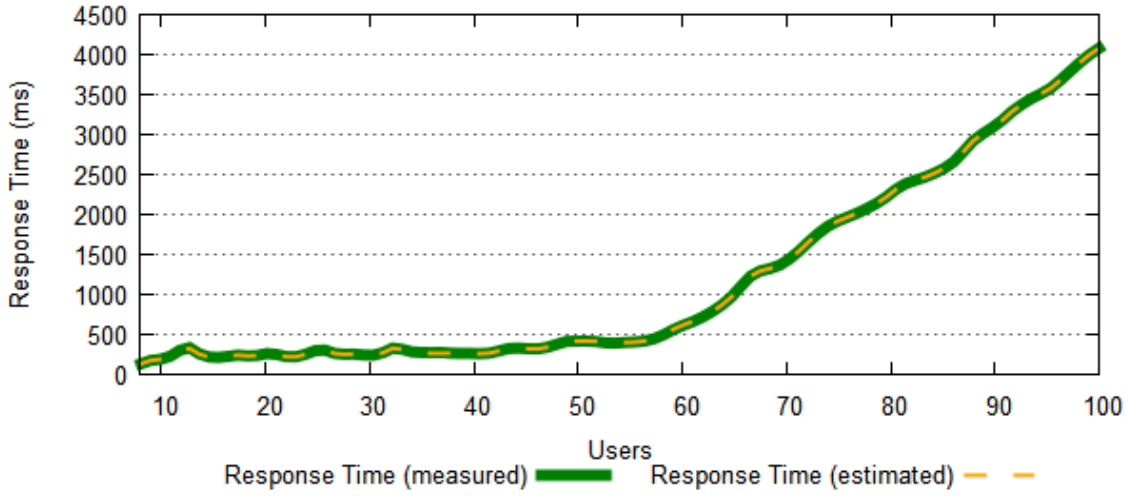



**Figure 7.5:** *Response time when the deployed topology contains two application servers.*

#### 7.2.2.1 Modelling under uncertainties

When an application is deployed on the cloud, the cloud's management system usually has to perform extra work to maintain and coordinate the deployment. Although this extra work is reflected in the measured performance metrics, its source cannot always be identified [122, 154] and, thus, it is considered an *uncertainty*.

There could be two types of uncertainties that can affect the accuracy of the application model in cloud and therefore the efficiency of the controller: parametric uncertainties and unmodelled dynamics. The parametric uncertainties refer to both parameters of the model (such as service times, number or probabilities of calls

between different components of the software, communication delays, etc.) or the intensity and mix of the workloads. The unmodelled dynamics refer to structural deficiencies of the model, that is missing components and queues that we have not knowledge of. The latter are very important in cloud where we do not have complete knowledge of the deployment environment.

To account for unmodelled dynamics, we add two sub-models to the application model as seen in Figure 7.6: a serial sub-model, made of a queuing centre, $\Delta s$, and a parallel sub-model, made of another queuing centre, $\Delta p$. The serial model $\Delta s$ will account for the delays in the application requests processing, due to additional proxies in the cloud. The parallel model $\Delta p$, will account for speed ups at higher loads, due to possible caching or heterogeneity of the cloud resources. The parameters of those queuing centres, such as service times and visit probabilities (for the parallel model) are unknown at the design time and they will be identified at runtime together with other parameters of the model. As a result, the architecture of the LQM model will consider the model for the deployed application in the cloud, as well as these two additional queuing centres. To deal with the parameter uncertainities, we identify the LQM parameters at runtime, using Kalman filters [121]

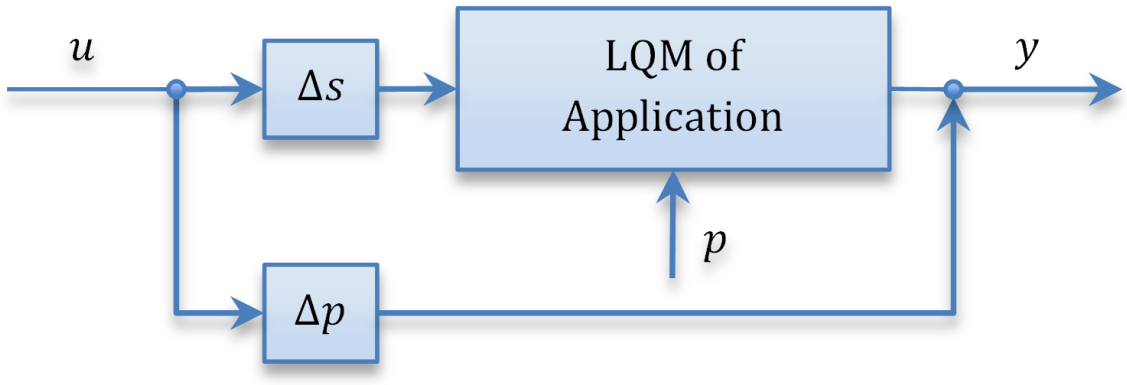



**Figure 7.6:** *Modelling structural uncertainties*

Finally, to deal with workload uncertainties, we design a robust controller. Instead of focusing on a single possible adaptive action, we consider a vector of available commands $u$ that have the advantage of being able to adjust the application and its deployment over large variations of workload. We discus the design of such a controller in details in the following sections.

### 7.2.3 Linearizing and discretizing the models

In general, a controller can be designed based on a linear model of the software system. We can extract such a linear model, if we observe the behavior of the system around an operational point $[x_{op}, y_{op}, u_{op}]$. If we focus closely to the operational point, we can linearly approximate the system's behavior from the non-linear model. We can take points close to the operational point by applying the corresponding *deltas* (i.e. small differences) from the following equations[2]:

$$\begin{cases} y(t) = y_a(t) - y_{op} \\ x(t) = x_a(t) - x_{op} \\ u(t) = u_a(t) - u_{op} \end{cases}$$

where $a$ denotes actual values and $op$ denotes values at the operational point.

Using these equations to find points close to the operational point, we can define a discrete-time linear system described by:

$$\begin{cases} x(t+1) &= Ax(t) + Bu(t) \\ y(t) &= Cx(t) + Du(t) \end{cases} \tag{7.1}$$

[2]We consider to be working in discrete time.

where $A$, $B$, $C$ and $D$ are matrices; $y$, $u$ and $x$ are vectors. When the matrices are constant in time, the system described by Equation 7.1 is a linear time invariant (LTI).

Although there is a number of available linearization techniques, in the context of this work, we use linear regression. First, we select an operating point (OP) and we take a number of measurements around OP from the LQM. Then, using regression we obtain the linear model around the OP. Given that linear regression is simple and efficient, we chose it in order to create models that may be valid for points further than OP. Other linearization techniques focus too much around OP. In our case, we can afford to sacrifice some of the accuracy, since according to our method the linear model will at some point have to be updated around a new OP.

### 7.2.4 Designing the controller

As it has already been mentioned, a feedback controller can compensate both for modelling errors and for perturbations. The feedback can be taken from the state variables (*state feedback*) or from the output (*output feedback*). The state feedback is the most effective, since assumes an understanding of inner workings and dependencies of the software system. Output feedback is more practical since output variables are easier to measure. The controller can vary from a simple constant matrix to a complex set of differential (or discrete) equations and can be synthesized to achieve some design goal, such as: stability of the entire system, perturbations rejection across wide ranges (robust control), or optimization. While the controller design goals can be diverse, there are standard design techniques that have well defined procedures and solutions and they are applied a wide range of domains, including

software control.

In our work, we employ *design for optimization* and the *Linear Quadratic Regulator* (LQR) technique to design a feedback controller. In design for optimization, a set of weighted goals are captured in an objective function. Most commonly, the goals are defined as a performance index across the state and command variables in form of a quadratic function (Equation 7.2 [131]), with the objective of finding the command $u$ that minimizes this quadratic function subject to the system in Equation 7.1. The weight matrices $Q_1$ and $Q_2$ penalize the state variations or the cost of adaptive commands, respectively. The construction of the weight matrices depend on the domain on which we apply the controller.

$$J = \sum_{0}^{\infty} x^T Q_x x + u^T Q_u u \tag{7.2}$$

An optimal feedback controller will find the $u$ that minimizes $J$. The optimization problem has the following solution [131]:

$$u = -Kx + k_r y_r \tag{7.3}$$

where $x$ is the system's state as defined earlier, $y_r$ is the goal for the output, $K$ is the feedback gain matrix and $k_r$ is steady-state factor.

The feedback gain matrix guarantees that the system will remain stable, in the form of $y = 0$, meaning that the output of the system will remain close to the operational point ($y_a(t) = y_{op}$). Since our goal is to bring, in fact, the output towards its desired state (i.e. $y = y_r$), we need another factor, which is $k_r$.

$K$ is calculated with LQR as:

$$K = -Q_2^{-1} B^T P x$$

where $P \in \mathbb{R}^{n \times n}$ is a positive definite, symmetric matrix that satisfies the *Riccati equation* [131]:

$$PA + A^T P - PBQ_2^{-1} B^T P + Q_1 = 0$$

Based on $K$, we can calculate $k_r$ by solving the following equation [131]:

$$1 = C(A - BK)^{-1} B k_r \tag{7.4}$$

Eventually, the set of commands issued to the system to adapt itself is given by:

$$u_a(t) = u_{op} + u$$

In the context of our work, we have already defined $x$, $u$ and $y_r$. $Q_1$ is the penalty matrix for any deviations in the system's performance from the normal or desired state. $Q_2$ is practically the cost of the resources in the cloud or reconfiguring the parameters of the deployed software. Variations within the two matrices indicate a difference in the significant of parameters. For example, we may deem a performance metric more important than another if the cost for a deviation of the former is higher than that of the latter. Equivalently, we may be more inclined to employ one resource over another if its cost is lower. Eventually, the combination of the two matrices will dictate our adaptation strategy. For example, if we know that a particular resource is more capable in handling a state parameter, which we deem important (i.e. high

cost for deviation), we may overlook the resource's potential high cost in order to adjust the system to its healthy state.

A significant advantage that feedback controllers offer is that they can give us an optimal, in terms of efficiency and effectiveness, set of adaptive actions both automatically and fast. Alternatively, we would have to simulate and evaluate every possible combination of states and adaptive actions (i.e. $x$ and $u$), probably over multiple dimensions, in terms of performance parameters and types of resources, before we can find the optimal adjustment.

## 7.3 Experimental studies

To validate the feedback controller designed in the previous section and illustrated by Figure 7.2 and its effectiveness in taking adaptive actions according to Algorithm 7.1, we have conducted a series of experiments. We deployed a *bookstore* application, developed using J2EE technology, on multiple Linux (Ubuntu) virtual machines, hosted on Amazon EC2. The application performs various SQL commands (select, insert, update). In the initial topology, the database server (MySql) was deployed on one instance, the web application server (Tomcat) was deployed on two instances, while a fourth instance was acting as a load balancer (Apache 2) to distribute the incoming web requests between the application servers. The deployment closely resembles the architecture from Figure 7.3. To automate the deployment process and management of the topology (adding/removing instances, changing configuration of instances, monitoring the instances), we have used the Hogna platform [155].

For this particular deployment, we have already commented on the non-linear relationship between the workload and CPU utilization and response time, regardless

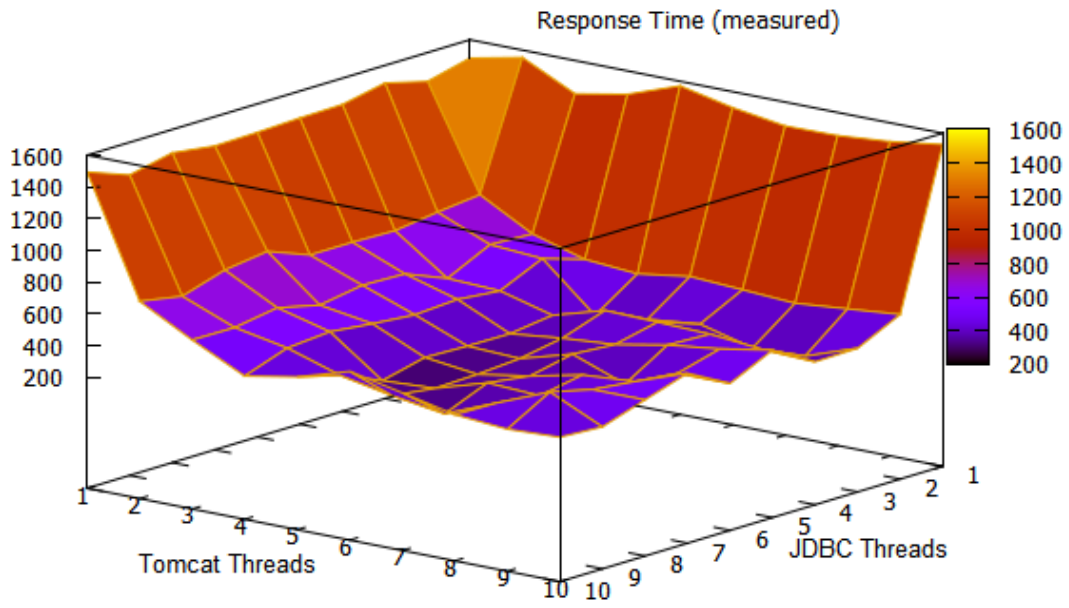
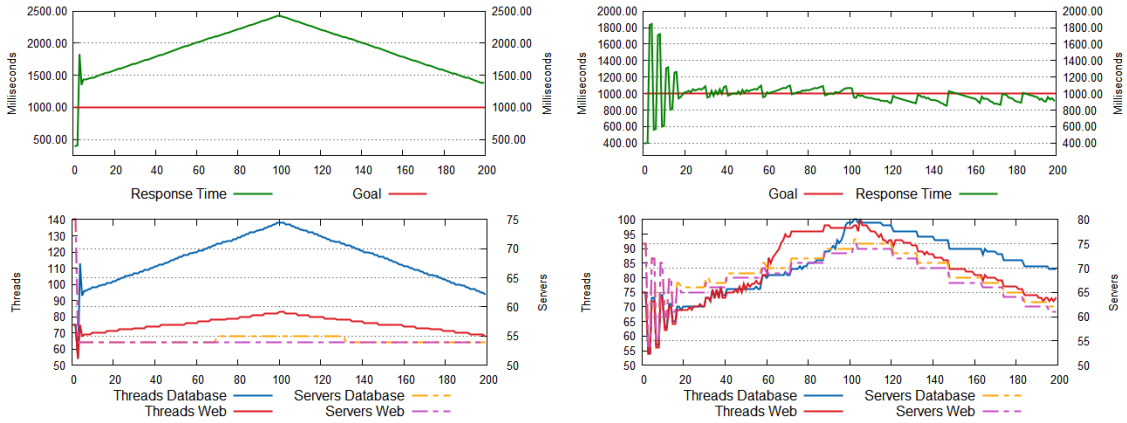



**Figure 7.7:** *Response time (milliseconds) as a function of number of threads, with constant workload.*

of the underlying topology, i.e. whether we have one web server (Figure 7.4) or two (Figure 7.5). In addition, Figure 7.7 shows that non-linearity also holds in a multidimensional model with respect to the number of available threads and servers.

As input to our algorithm and in order to design the controller, we need to define the set of commands $\mathcal{U}$, the set of monitored state parameters $\mathcal{X}$ and the set of controlled system outputs $\mathcal{Y}$. We define the command points for our system as $\mathcal{U} = [S_d, T_d, S_w, T_w]$, where $T_d$ is the number of threads for database servers and $T_w$ for web servers; $S_d$ is the number of database servers and $S_w$ is the number of web servers. The state vector, $x$, contains the response time of the web application, the same as the output vector, $x$ (see Equation 7.1). For the rest of our experiments,




**(a)** *Without re-linearization.* **(b)** *With re-linearization.*

**Figure 7.8:** *Behaviour of the system when the goal for Response Time was set to 1000 ms.*

when we refer to specific values of these vectors, we will note them with their lower case representation, i.e. $u$, $x$ and $y$.

We construct a Layered Queuing Model (LQM) using the OPERA tool [36] to track the behavior of the system. When the LQM becomes inaccurate, we apply a retuning procedure based on Kalman filters that will calculate the model parameters, so that the model matches the measured metrics.

To linearize the LQM, we have used the `LinearRegression` function from the package `optim` [156] in Octave to calculate the matrices $A$ and $B$ from Equation 7.1.

We have also chosen $C = [1]$ and $D = 0$. The reason for this is because we picked response time to represent both the state and the output of the system. From Equation 7.1, we have that $y(t) = Cx(t) + Du(t)$. Given that $y = x$, it is derived that $C = [1]$ and $D = 0$.

For the linear-quadratic regulator (LQR), we have used the implementation offered by Octave's package `control` [157]. The parameters for the `lqr` function were the

matrices $A$ and $B$ identified during linearization and the following weight matrices $Q_1$ and $Q_2$:

$$Q_1 = [1] \qquad Q_2 = \begin{bmatrix} 100\,000 & 0 & 0 & 0 \\ 0 & 1\,000 & 0 & 0 \\ 0 & 0 & 100\,000 & 0 \\ 0 & 0 & 0 & 1\,500 \end{bmatrix} \tag{7.5}$$

The `lqr` function calculated the matrix $K$ as discussed in Section 7.2.4. To calculate $k_r$, we assumed that all of its four components were equal and applied the Equation 7.4:

$$k_r = \frac{1}{b_1 + b_2 + b_3 + b_4} \times \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \times (1 - (A - BK))$$

where $b_i$ are the components of $B$:

$$B = \begin{bmatrix} b_1 & b_2 & b_3 & b_4 \end{bmatrix}$$

$K$ and $k_r$ form our controller, and allow us to calculate a command by applying the Equation 7.3.

For the first experiment (Figure 7.8), we have set the goal for the response time to be 1000 ms. As for the workload, we start with 18000 users, linearly increasing up to 23000 and then linearly decreasing again up to 18000 users. Figure 7.8a shows

the behaviour of the system when using a conventional LQM, liniarized only at the beginning of the experiment, thus creating only one controller ($K$ and $K_r$ from Equation 7.3), while Figure 7.8b shows the behaviour when the LQM is relinearized every time its error (difference between the measured response time and the estimated one using the linear model) exceeds the threshold of 100 ms.

The experiment shows that when we linearize often (we have built 31 linear models for the whole duration of the experiment), we manage to stabilize the system and maintain the response time close to the desired value. Also, the value of $J$ (Equation 7.2) in this case was approximately $21 \times 10^6$, which is significantly smaller than the value obtained with a single linearization: $17 \times 10^9$. Considering that the goal of the controller is to minimize $J$, this shows that multiple linearization is a significantly better model. Figure 7.8a shows that the controller fails to maintain the response time close to the goal when the workload fluctuates.

Running more experiments with different goals (at 300 ms, 500 ms and 700 ms) produced similar results: the relinearization of the LQM model enabled the controller(s) to maintain a response time close to the desired goal, while doing a single linearization at the beginning of the experiment generated poor results.

In the second experiment, we wanted to evaluate the behaviour of the system in the presence of irregular workload, i.e., when the workload suddenly increases or decreases non-monotonically. The results are summarized in Figure 7.9. The bottom plot shows the the shape of the workload. The goal for the response time in this experiment was set to 700 ms and as it can be seen in the Figure, the controller is able to stabilize the system around this goal. In fact, the controller helps the system achieve an average response time of 703 ms (standard deviation of 73.37).

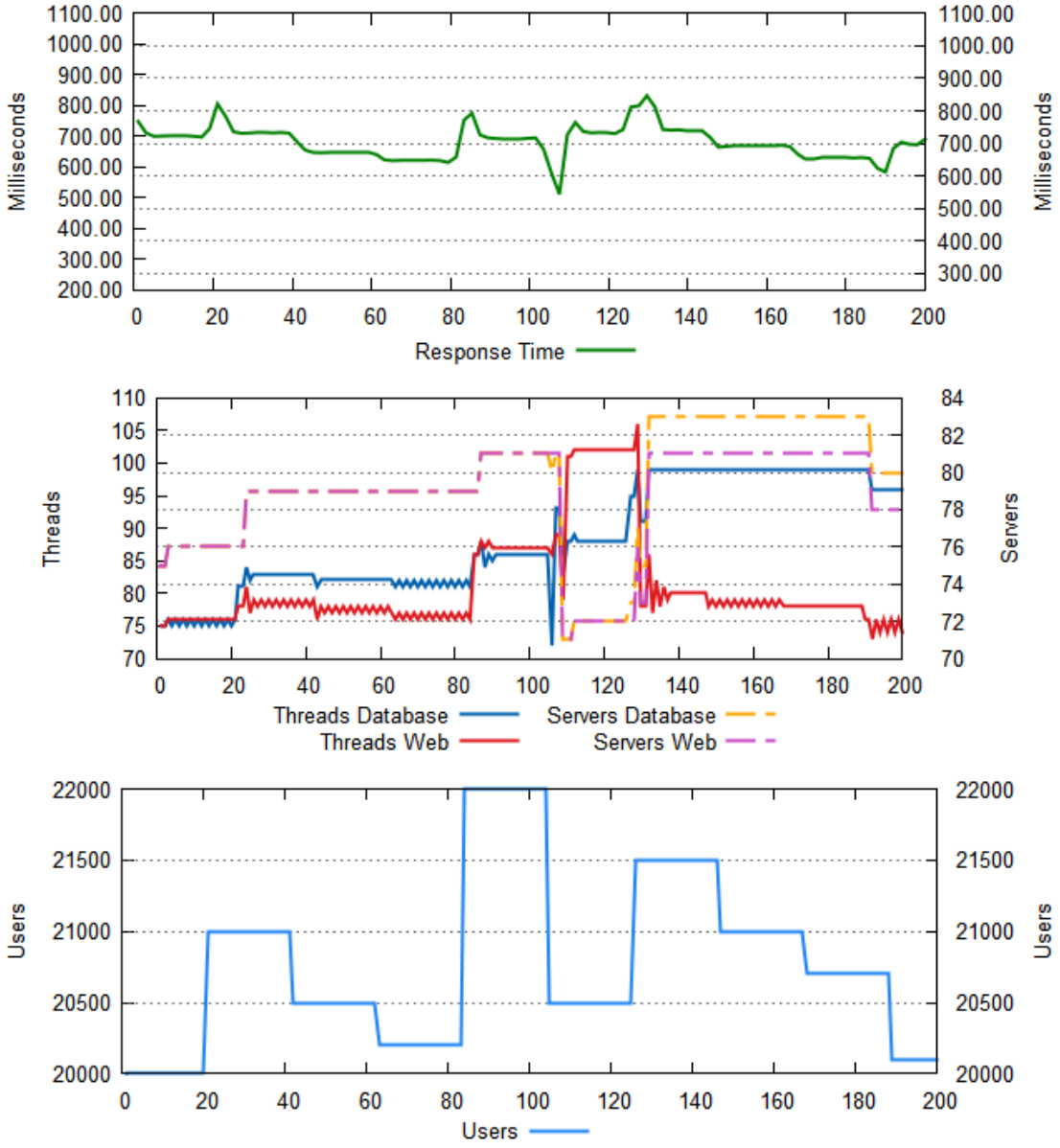


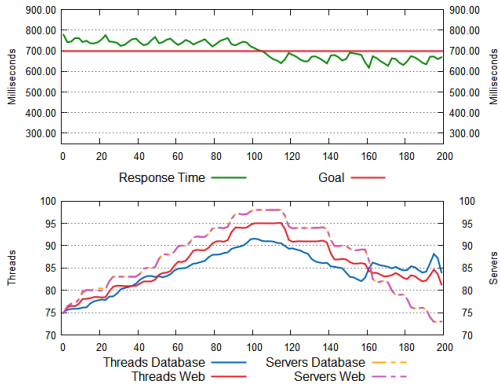

**Figure 7.9:** *Behaviour of the system when the goal for Response Time was set to 700 ms, and the workload increasing/decreasing suddenly.*

In order to evaluate the influence of the weight matrices $Q_1$ and $Q_2$ on the

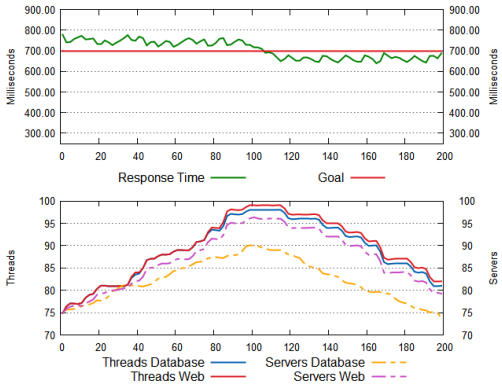
behaviour of the controller, we have set an experiment with the following matrices:

$$Q_1 = [1] \qquad Q_2 = \begin{bmatrix} 1\,000 & 0 & 0 & 0 \\ 0 & 100\,000 & 0 & 0 \\ 0 & 0 & 1\,500 & 0 \\ 0 & 0 & 0 & 100\,000 \end{bmatrix} \tag{7.6}$$

We set the response time goal to be 700 ms, and then increased the workload linearly and then decreased it, also linearly. In Figure 7.10a is shown the behaviour of the system when the matrices from Equation 7.5 (prefer to add servers) are used, while Figure 7.10b shows the results for matrices from Equation 7.6 (prefer to add threads).




**(a)** *Using the $Q_1$ and $Q_2$ from Equation 7.5.*




**(b)** *Using the $Q_1$ and $Q_2$ from Equation 7.6.*

**Figure 7.10:** *The effect of the weight matrices $Q_1$ and $Q_2$ on the behaviour of the controller.*

In both situations, the system was stable and the goal has been maintained. Using the matrices from Equation 7.5, the value for $J$ (Equation 7.2) was approximately $27 \times 10^7$; the utilization of the weights from Equation 7.6 resulted in a value for $J$ of

almost half, in the range of $14 \times 10^7$.

Figure 7.11 shows the error of the linear model with respect to the LQM. During the experiment, the workload has been kept constant, and we varied only number of web servers. Also, the number of threads and database servers were constant. The green line is the response time as estimated with the LQM. The non-linear model has been linearized at the begining of the experiment, when there were 100 web servers. Then we started removing servers, one at a time. The blue line shows the response time estimated using the linear model generated at the begining, while the red line shows the response time when the LQM is linearized as often as necessary. When we are left with only 75 web servers, the linear model diverges too much from the LQM, and a re-linearization is triggered (see that the red and blue lines diverge, sign that a new linear model has been built). This new linear model is valid when the topology has between 75 to 55 web servers and a new linearization is required at 55. By linearizing often, we maintain a linear model that closely matches the non-linear one. Note that the red line follows the green one, while the blue line (estimations done with the original linear model) diverges significantly.

Each background color in the plot shows the range of validity of a linear model. A change in the background color signals that a new linearization has been performed. Also, note that as the response time grows faster, re-linearization frequency increases.

### 7.3.1 Quality of control parameters

When the goal of the controller is the optimization, the quality of control is judged by how close to global optimum the controllers bring the system. In case of a robust control, the quality is measured by the insensitivity to changes in the managed

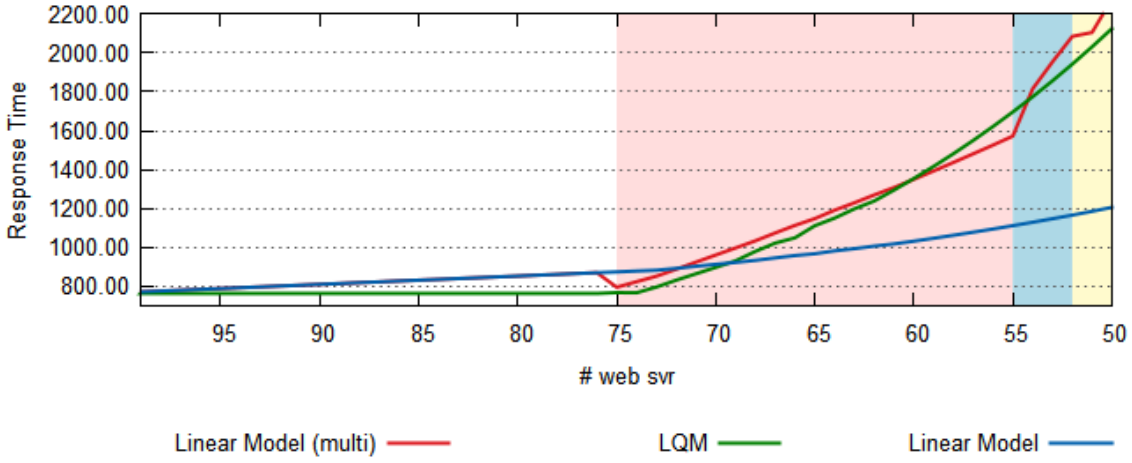



**Figure 7.11:** *The errors of the linear model.*

element and environment and high tolerance to perturbations.

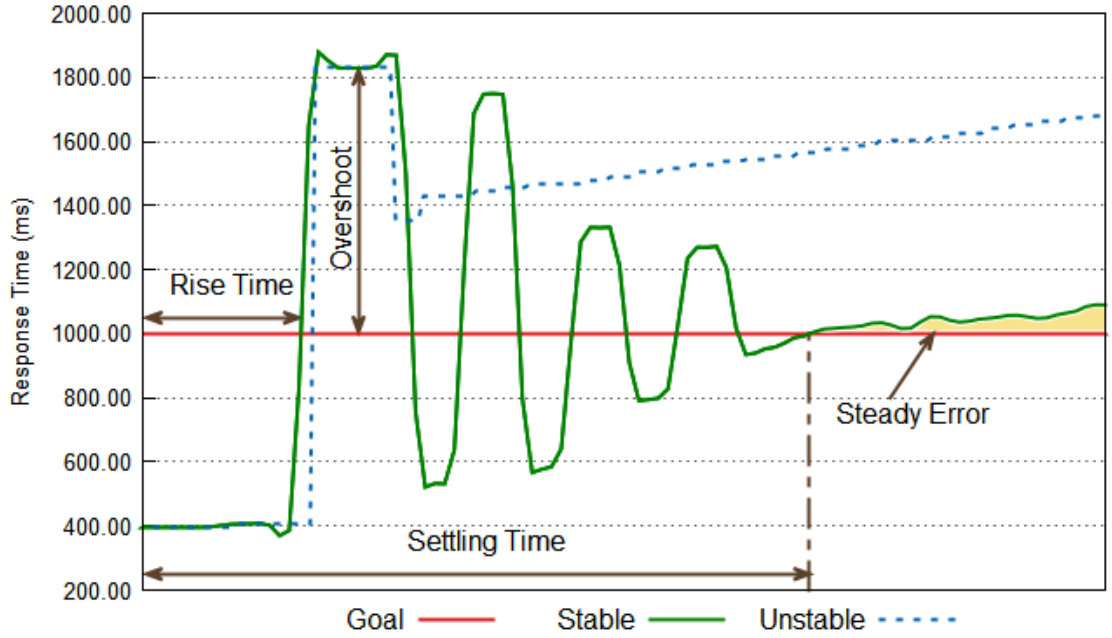



**Figure 7.12:** *Quality metrics of an autonomic system*

Many controller goals can be reduced to setpoint control (like the one in Figure 7.12) and, in this case, the quality of control can be measured with several

metrics: *rise time*, *overshooting*, *settling time*, *steady error* [158, 159]. Figure 7.12 illustrates the behavior of the response time of the web service cluster from the first experiment when perturbed by a workload change. The figure actually shows the beginning of the experiment until the controller was able to stabilize its behavior. Let's assume that response time is setpoint to the horizontal continuous line in Figure 7.12 representing the goal of 1000 ms. The perturbation in workload, that affects the system at the origin of the *time* axis, starts to increase the response time of the system (*rise time*). A good controller will compute a command $u$ (the number of database and web servers and threads in the cluster) that will bring the response time back to normal in a short time (*rise time*) by increasing the number of resources in the cluster. Very likely, the system will become over-provisioned, but the *overshooting* must be small since it can affect other attributes (like cost). There will be some oscillations around the setpoint value, but after a time (*settling time*), preferably short, the response will stabilize. In Figure 7.12, there is also an example of unstable scenario, where the once-linearized controller cannot reach the target response time.

### 7.3.2 Threats to Validity

The effectiveness of the proposed method depends on the accuracy of the LQM performance model. In some circumstances, the LQM may take longer to synchronize with the deployed system in the cloud, and this may impede the linearization process and by extension the effectiveness of the controller.

Our approach assumes that the system is controllable. When the system is controllable (the rank of the matrix $[A \quad AB \quad A^2B \quad \cdots \quad A^{n-1}B]$ is $n$, where $n$ is the size of matrix $A$), the Linear Quadratic Regulator will build the controller successfully.

In our experiments, the system was controllable, as the matrices $A$ and $B$ produced by the linearization satisfy the controllability condition for all linearization points. In future work, we are planning to investigate conditions, in which matrices A and B are uncontrollable for software systems, and propose guidelines for avoiding such situations when designing adaptive system. Currently, if such a condition arises, we recommend changing sets $\mathcal{X}$, for state and $\mathcal{U}$ for commands, if matrices A and B are uncontrollable.

As far as our experiments are concerned, there a few threats to validity. The application domain of the proposed methodology is supposed to be cloud environment and multi-tier transactional applications. However, as it can be seen from our experiments, having to deploy dozens or hundreds of virtual machines in a public cloud can be extremely costly. For this reason, to validate at scale, we relied on some experiments on simulations, which may affect the validity and more importantly the generalization of our experiments. Nevertheless, our simulations are based on realistic settings including the web application, the underlying infrastructure and the behavior of the whole deployment. In the future, we plan to perform additional experiments on real deployments. The validity of our results may also be affected by the assumptions we made on the structure of the model, for example, how we decided on the parameters of $x$, $y$ and $u$ and how we defined the weight matrices $Q$. In the context of our experiments, we believe that these parameters were defined realistically and within reason, but we plan to expand on them and perform sensitivity test to investigate how their definition affects the results.

## 7.4 Conclusions

In this work, we proposed a model identification adaptive controller as a management system to monitor and maintain applications on cloud environments. Our experiments have shown that the use of control theory in an Adaptive Manager for cloud applications performs exceptionally well and can produce a robust and effective controller. Additionally, the mathematical background of control theory allows us to systematically design and verify such adaptive management systems. Our method is capable on operating on a multidimensional level both with respect to the goals that are to be achieved, as well as the adaptive actions. The proposed controller performs better than previous methods thanks to the concept of multilinearization, which allows the controller to readjust itself in order to better monitor the system and produce more efficient adaptive actions. In the future, we plan to evaluate more multidimensional controllers with respect to the goals and study their performance in a real cloud setting. Moreover, we plan to perform extensive sensitivity tests concerning the construction parameters of the controller.

# Chapter 8

# Conclusions

The complexity of the software systems has increased significantly over the last years and it reached levels that makes it challenging to manage them. For applications deployed in cloud, the management is even more difficult because of the resource sharing environment, that results in unrelated applications influencing each other.

Model based autonomic managers are becoming an attractive solution to handle the complexity and the highly dynamic environment where the applications reside. The autonomic managers can react to change (in environment, in workload, in utilization patterns) much faster than a human operator can.

In this thesis, we have addressed several major research questions related to engineering adaptive systems for web applications: (1) what are the bottlenecks in a web application and what type of workload will saturate them, (2) how to model a web application deployed in cloud and handle the cloud variability so the model stays accurate and useful, (3) how to find the resource types and quantities that need to be added / removed so the goal can be maintained, (4) and how to use models in order to protect web applications against Denial of Service attacks which is a major

threat on today's Internet.

We have presented autonomic systems that can make decisions regarding deployment and management of web applications at various stages of their life: at design time when the best achitecture (based on the Service License Agreement) is selected, after deployment when the bottlenecks and worst workload mixes are uncovered, and at runtime when the application is subject to normal fluctuations in usage patterns but also a target for malicious traffic. All methods and algorithms we have developed make use of an autonomic manager, and incorporate performance models that capture the essence of the managed web application.

The major original contributions we propose are:

A. a model-based method to explore the workload space in order to uncover and saturate the bottlenecks of a deployed web application;

B. a performance model for web applications deployed in clouds capable to handle cloud variability;

C. a robust adaptation architecture and method (Model Identification Adaptive Control) capable to synthesize a controller at runtime that will provide the adaptation strategy based on a goal;

D. model-based adaptive architectures and algorithms focused on detecting DoS attacks at the web application level and mitigating them appropriately;

Specifficaly, in **contribution (A)** (Chapter 4) we show how to efficiently uncover the bottlenecks and workloads that saturates them. The method uses an analytical representation of the software system, a two-layer queuing model that captures the hardware and software contention for resources.

The model is used to uncover the existing bottlenecks in the system and provides hints about the workloads that will saturate them. Using a feedback loop to guide the system towards a stress goal, we find the exact workloads that will overload it. The model is automatically tuned, using on-line estimators that finds the model parameters.

The workloads are characterized by *workload intensity*, which is the total number of users, and by the *workload mix*, which is ratio of users in each class of service. By extracting the *switching points* from the model, we are able to compute the *stress vectors* that yield a bottleneck change. Applying a hill-climbing strategy for workload intensity along the stress vectors, we are able to reach the stress goal.

We applied the method to find the workload intensity and workload mix that yields target software and hardware utilization limits or a target response time. The results show that the algorithm is capable to reach the target goal with a small number of iterations and therefore testcases.

In **contribution (B)** (Chapter 6) we present experimental results demonstrating how the model diverges from reality over time due to the unpredictable variability of cloud services, describe the modifications required to account for unexplained delays, and we present a second set of results to show that, after the modifications, the model remained synchronized with the actual performance of the real cloud system.

Specifically, we introduce a new resource in the model, a "cloud delay centre", designed to capture all the undocumented work or delay. The cloud delay queue is shared by all classes of traffic, it has no limit, it exists at the software level, and it is the first queue encountered by incoming requests. The rationale for this is that any undocumented work is not related to the web application itself, but to the cloud

environment and should affect all classes equally.

The validation tests were executed in real-world conditions, on Amazon EC2, giving a greater confidence in the results.

In **contribution (C)** (Chapter 7) we show how control theory can be used by an autonomic manager. We described the details for implementation of a model identification adaptive controller (MIAC) using a combination of performance and control models. We show that our approach can account for uncertainty and modelling errors and efficiently adapt a cloud deployment.

Our experiments have shown that the use of control theory in an Adaptive Manager for cloud applications performs exceptionally well and can produce a robust and effective controller. Additionally, the mathematical background of control theory allows us to systematically design and verify such adaptive management systems.

The proposed method is capable on operating on a multi-dimensional level both with respect to the goals that are to be achieved, as well as the adaptive actions. The controller performs better than previous methods thanks to the concept of multilinearization, which allows the controller to readjust itself in order to better monitor the system and produce more efficient adaptive actions. In the future, we plan to evaluate more multidimensional controllers with respect to the goals and study their performance in a real cloud setting.

In **contribution (D)** (Chapters 5 and 6) we introduce a novel adaptive architecture to detecting and mitigating (D)DoS attacks. We show how we can use a combination of application performance modeling with statistical anomaly detection to establish a set of filtering rules, then how to iteratively fine-tune them using application- and system-level performance metrics synchronized with the performance

model.

The approach is enabled by a layered queuing network model adapted for cloud as described in contribution (B). The model accurately predicts short-term cloud behavior, despite cloud variability over time. We evaluated our approach on Amazon EC2 and demonstrate the ability to horizontally scale a sample web application in response to an increase in legitimate traffic while mitigating multiple DoS attacks, achieving the established performance goal.

## 8.1 Limitations and Future Work

In all contributions stated above one of the most important factors is the accuracy of the model. The model has a direct impact to the quality of the results of the methods presented. An inacurate model will lead to poor identification of the bottlenecks and workloads that saturate them (**contribution (A)**), creation of a wrong controller that should drive the adaptation strategy (**contribution (C)**) and improper classification of malicious traffic (**contribution (D)**).

In order to preserve the accuracy of the model we use a procedure to re-synchronize the model with the system. In some circumstances, the LQM may take longer to synchronize, and this may result in longer reaction times of the autonomic manager.

In **contribution (C)** (Chapter 7) we assume that the system is controllable. In our experiments the system was controllable. In future work, we are planning to investigate conditions that make a system uncrollable and propose guidelines for avoiding such situations when designing adaptive system.

Also, since some of the experiments relied on simulations, future work will include validation of the proposed method for systems deployed on cloud that run under

realistic conditions.

In **contribution (D)** (Chapters 5 and 6) the goal was to protect a web application from (D)DoS attacks by accurately discriminate between desirable traffic and undesirable traffic.

The experiments show that in certain situations, the mitigation strategy is filtering segments of non-malicious traffic (altough the misidentified traffic is quickly restored). The decision is correct in that it restores SLOs, but of course incorrect since it doen't involve malicious traffic. This is a limitation that we will address in the future work.

A false positive is when traffic is detected as an attack incorrectly. Because our approach does not block traffic outright but instead forwards to a CAPTCHA test, that traffic is not lost. However, the test may be annoying to users. There were no false positives in our experiments.

A false negative is when malicious traffic is not detected. In our approach, attacks are only detected when the performance of the application suffers; any malicious traffic that does not have a negative impact will not be detected, but is by definition not a true DoS attack. As our approach filters types of traffic until the applications performance is acceptable, false negatives will not impact the application.

Rather than blocking traffic outright, our approach relies on the use of a test only human users will pass (e.g., a CAPTCHA). For our focus of user-facing web applications, this is sufficient to avoid dropping legitimate traffic outright. An extension to this work would consider similar tests to differentiate legitimate automated traffic from malicious automated traffic, for example pre-shared keys or other trust negotiation mechanisms.

This approach is intended to address a popular type of DoS attack, a "fast"

application-aware attack where the traffic levels increase sharply. This may not be the best approach to mitigate "slow" application-aware attacks where the traffic increases gradually over time; we have not evaluated the performance for this type of attack. As mentioned, we assume existing techniques are in place to defend against attacks not at the application level.

# Bibliography


[1] P. Horn, "Autonomic computing: IBM's Perspective on the State of Information Technology," 2001.

[2] S. Joines, R. Willenborg, and K. Hygh, *Performance Analysis for Java Websites*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010. [Online]. Available: http://dx.doi.org/10.1007/s13174-010-0007-6

[4] X. Geng and A. B. Whinston, "Defeating distributed denial of service attacks," *IT Professional*, vol. 2, no. 4, pp. 36–41, July 2000. [Online]. Available: http://dx.doi.org/10.1109/6294.869381

[5] P. Zaroo, "A survey of ddos attacks and some ddos defense mechanisms," *Advanced Information Assurance (CS 626)*, 2002.

[6] K. Arora, K. Kumar, and M. Sachdeva, "Impact analysis of recent ddos attacks," *International Journal on Computer Science and Engineering*, vol. 3, no. 2, pp. 877–883, 2011.

[7] D. Kaur, M. Sachdeva, and K. Kumar, "Recent ddos incidents and their impact," *International Journal of Scientific & Engineering Research*, vol. 3, no. 8, 2012.

[8] R. Dobbins, C. Morales, D. Anstee, J. Arruda, T. Bienkowski, M. Hollyman, C. Labovitz, J. Nazario, E. Seo, and R. Shah, "Worldwide Infrastructure Security Report," http://www.arbornetworks.com/dmdocuments/ISR2010_EN.pdf, Arbor Networks, Tech. Rep., 2010.

[9] V. Durcekova, L. Schwartz, and N. Shahmehri, "Sophisticated denial of service attacks aimed at application layer," in *ELEKTRO, 2012*. IEEE, 2012, pp. 55–60.

[10] "Low orbit ion cannon (loic)," http://en.wikipedia.org/wiki/Low_Orbit_Ion_Cannon.

[11] B. Subraya and S. Subrahmanya, "Object driven performance testing of web applications," in *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, 2000, pp. 17–26. [Online]. Available: http://dx.doi.org/10.1109/APAQ.2000.883774

[12] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to linear regression analysis*. John Wiley and Sons, 2012, vol. 821.

[13] J. A. Rolia and K. C. Sevcik, "The method of layers," *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689–700, 1995. [Online]. Available: http://dx.doi.org/10.1109/32.403785

[14] D. A. Menascé, "Simple analytic modeling of software contention," *SIGMETRICS Performance Evaluation Review*, vol. 29, no. 4, pp. 24–30, 2002. [Online]. Available: http://doi.acm.org/10.1145/512840.512844

[15] J. Zahorjan, K. C. Sevcik, D. L. Eager, and B. I. Galler, "Balanced job bound analysis of queueing networks," in *SIGMETRICS '81: Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 1981. [Online]. Available: http://doi.acm.org/10.1145/800189.805475

[16] D. L. Eager and K. C. Sevcik, "Performance bound hierarchies for queueing networks," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 99–115, 1983. [Online]. Available: http://doi.acm.org/10.1145/357360.357363

[17] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[18] M. Reiser and S. S. Lavenberg, "Mean-value analysis of closed multichain queuing networks," *J. ACM*, vol. 27, no. 2, pp. 313–322, 1980. [Online]. Available: http://doi.acm.org/10.1145/322186.322195

[19] G. Balbo and G. Serazzi, "Asymptotic analysis of multiclass closed queueing networks: multiple bottlenecks," *Performance Evaluation*, vol. 30, no. 3, pp. 115–152, 1997. [Online]. Available: http://dx.doi.org/10.1016/S0166-5316(97)00005-9

[20] M. Litoiu, J. Rolia, and G. Serazzi, "Designing process replication and activation: A quantitative approach," *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1168–1178, 2000. [Online]. Available: http://dx.doi.org/10.1109/32.888630

[21] APERA, "Application Performance Evaluation and Resource Allocator (APERA)," 2009. [Online]. Available: http://www.alphaworks.ibm.com/tech/apera

[22] M. Litoiu, M. Woodside, and T. Zheng, "Hierarchical model-based autonomic control of software systems," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, 2005. [Online]. Available: http://doi.acm.org/10.1145/1082983.1083071

[23] M. Woodside, T. Zheng, and M. Litoiu, "The use of optimal filters to track parameters of performance models," in *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*. Washington, DC, USA: IEEE Computer Society, 2005, p. 74. [Online]. Available: http://dx.doi.org/10.1109/QEST.2005.40

[24] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with extended kalman filters," in *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2005, pp. 334–345.

[25] D. A. Menascé and V. A. F. Almeida, *Capacity planning for Web performance: metrics, models, and methods*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1998.

[26] ——, *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[27] N. J. Gunther, *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[28] D. Thakkar, A. E. Hassan, G. Hamann, and P. Flora, "A framework for measurement based performance modeling," in *WOSP '08: Proceedings of the 7$^{th}$ international workshop on Software and performance*. New York, NY, USA: ACM, 2008, pp. 55–66. [Online]. Available: http://doi.acm.org/10.1145/1383559.1383567

[29] H. Gomaa and D. A. Menascé, "Performance engineering of component-based distributed software systems," in *Performance Engineering, State of the Art and Current Trends*. London, UK: Springer-Verlag, 2001, pp. 40–55.

[30] D. Thakkar, "Automated capacity planning and support for enterprise applications," Master's thesis, Queens University, 2009.

[31] G. Imre, T. Levendovszky, and H. Charaf, "Modeling the effect of application server settings on the performance of j2ee web applications," in *TEAA'06: Proceedings of the 2nd international conference on Trends in enterprise application architecture*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 202–216.

[32] M. Sopitkamol and D. A. Menascé, "A method for evaluating the impact of software configuration parameters on e-commerce sites," in *WOSP '05: Proceedings of the 5$^{th}$ international workshop on Software and performance*.

New York, NY, USA: ACM, 2005, pp. 53–64. [Online]. Available: http://doi.acm.org/10.1145/1071021.1071027

[33] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *Software Maintenance, 2009 (ICSM 2009). IEEE International Conference on.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 125–134. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2009.5306331

[34] H. Malik, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, "Using load tests to automatically compare the subsystems of a large enterprise system," in *Proceedings of the 2010 IEEE 34$^{th}$ Annual Computer Software and Applications Conference*, ser. COMPSAC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 117–126. [Online]. Available: http://dx.doi.org/10.1109/COMPSAC.2010.18

[35] G. Franks, P. Maly, M. Woodside, D. C. Petriu, A. Hubbard, and M. Mroz, "Layered Queueing Network Solver (LQNS)," 2012. [Online]. Available: http://www.sce.carleton.ca/rads/lqns/

[36] OPERA, "Optimization, Performance Evaluation and Resource Allocator (OPERA)," 2013. [Online]. Available: http://www.ceraslabs.com/technologies/opera

[37] F. Kargl, J. Maier, and M. Weber, "Protecting web servers from distributed denial of service attacks," in *Proceedings of the 10th International Conference on World Wide Web*, ser. WWW '01. New York, NY, USA: ACM, 2001, pp. 514–524. [Online]. Available: http://doi.acm.org/10.1145/371920.372148

[38] J. Roman, B. Radek, V. Radek, and S. Libor, "Launching distributed denial of service attacks by network protocol exploitation," in *Proceedings of the 2nd international conference on Applied informatics and computing theory*, ser. AICT'11. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2011, pp. 210–216.

[39] E. Zuckerman, H. Roberts, R. McGrady, J. York, and J. Palfrey, *Distributed Denial of Service Attacks Against Independent Media and Human Rights Sites*. Publisher Open Society Institute, 2010.

[40] The Hacker's Choice, "THC SSL DOS," http://thehackerschoice.wordpress.com/2011/10/24/thc-ssl-dos/, 2012.

[41] Y. Chung, "Distributed denial of service is a scalability problem," *CoRR*, vol. abs/1104.0057, 2011. [Online]. Available: http://arxiv.org/abs/1104.0057

[42] J. Mirković, "D-WARD: DDoS Network Attack Recognition and Defense," 2002.

[43] M. Sachdeva, G. Singh, and K. Kumar, "Deployment of Distributed Defense against DDoS Attacks in ISP Domain," *International Journal of Computer Applications*, vol. 15, no. 2, pp. 25–31, February 2011, published by Foundation of Computer Science. [Online]. Available: http://dx.doi.org/10.5120/1918-2561

[44] A. Garg and A. L. Narasimha Reddy, "Mitigation of DoS attacks through QoS regulation," in *Quality of Service, 2002. 10$^{th}$ IEEE International Workshop on.*

Washington, DC, USA: IEEE Computer Society, 2002, pp. 45–53. [Online]. Available: http://dx.doi.org/10.1109/IWQoS.2002.1006573

[45] A. Juels and J. G. Brainard, "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks." in *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 1999. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss1999.html#JuelsB99

[46] T. H. Nguyen, C. T. Doan, V. Q. Nguyen, T. H. T. Nguyen, and M. P. Doan, "Distributed defense of distributed DoS using pushback and communicate mechanism," in *Proc. Int Advanced Technologies for Communications (ATC) Conf*, 2011, pp. 178–182. [Online]. Available: http://dx.doi.org/10.1109/ATC.2011.6027461

[47] S. M. Khattab, C. Sangpachatanaruk, R. Melhem, D. l Mosse, and T. Znati, "Proactive server roaming for mitigating denial-of-service attacks," in *Proc. ITRE2003 Information Technology: Research and Education Int. Conf*, 2003, pp. 286–290. [Online]. Available: http://dx.doi.org/10.1109/ITRE.2003.1270623

[48] M. Long, C.-H. J. Wu, J. Y. Hung, and J. D. Irwin, "Mitigating performance degradation of network-based control systems under denial of service attacks," in *Proceedings 30$^{th}$ Annual Conference of IEEE Industrial Electronics Society IECON 2004*, vol. 3. Washington, DC, USA: IEEE Computer Society, 2004, pp. 2339–2342. [Online]. Available: http://dx.doi.org/10.1109/IECON.2004.1432165

[49] A. K. Pandey and C. Pandu Rangan, "Mitigating denial of service attack using proof of work and Token Bucket Algorithm," in *Proc. IEEE Students' Technology Symp. (TechSym)*, 2011, pp. 43–47. [Online]. Available: http://dx.doi.org/10.1109/TECHSYM.2011.5783861

[50] X. Wu and Y. K. Y. David, "Mitigating denial-of-service attacks in MANET by incentive-based packet filtering: A game-theoretic approach," in *Proc. Third Int. Conf. Security and Privacy in Communications Networks and the Workshops SecureComm 2007*, 2007, pp. 310–319. [Online]. Available: http://dx.doi.org/10.1109/SECCOM.2007.4550349

[51] P. Jain, J. Jain, and Z. Gupta, "Mitigation of Denial of Service (DoS) Attack," *International Journal of Computational Engineering and Management (IJCEM)*, vol. 11, pp. 38–44, Jaunuary 2011.

[52] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A survey of intrusion detection techniques in cloud," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 42–57, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jnca.2012.05.003

[53] IBM, "An architectural blueprint for autonomic computing," IBM, Tech. Rep., 2005. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[54] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003. [Online]. Available: http://dx.doi.org/10.1109/MC.2003.1160055

[55] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A simple network management protocol (snmp)," 1990. [Online]. Available: https://tools.ietf.org/html/rfc1157

[56] M. C. Huebscher and J. A. McCann, "A survey of autonomic computingdegrees, models, and applications," *ACM Computing Surveys*, vol. 40, no. 3, pp. 7:1–7:28, Aug 2008. [Online]. Available: http://dx.doi.org/10.1145/1380584.1380585

[57] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1516533.1516538

[58] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan, "Qmon: Qos- and utility-aware monitoring in enterprise systems," in *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, ser. ICAC '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 124–133. [Online]. Available: http://dx.doi.org/10.1109/ICAC.2006.1662390

[59] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing*. Springer, 2013.

[60] A. E. Walsh, *J2EE 1.4 Essentials*, 1st ed. Hungry Minds, Incorporated, 2003.

[61] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[62] M. Litoiu, "A performance engineering method for web applications," in *Web Systems Evolution (WSE), 2010 12$^{th}$ IEEE International Symposium on*,

Sept 2010, pp. 101–109. [Online]. Available: http://doi.acm.org/10.1109/WSE.2010.5623583

[63] T. Saaty, *Fundamentals of Decision Making and Priority Theory With the Analytic Hierarchy Process*, ser. AHP series. RWS Publications, 2000.

[64] D. A. Menasce and V. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[65] H. A. Simon, *The Sciences of the Artificial*. Cambridge, MA, USA: MIT Press, 1996.

[66] R. Geist, R. Allen, and R. Nowaczyk, "Towards a model of user perception of computer systems response time," *SIGCHI Bull.*, vol. 17, no. SI, pp. 249–253, May 1986. [Online]. Available: http://doi.acm.org/10.1145/30851.275638

[67] B. Shneiderman and C. Plaisant, *Designing the User Interface: Strategies for Effective Human-Computer Interaction (4th Edition)*. Pearson Addison Wesley, 2004.

[68] J. A. Hoxmeier and C. Dicesare, "System response time and user satisfaction: An experimental study of browser-based applications," in *Proceedings of the Association of Information Systems Americas Conference*, 2000, pp. 10–13.

[69] A. Bouch, A. Kuchinsky, and N. Bhatti, "Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI

'00. New York, NY, USA: ACM, 2000, pp. 297–304. [Online]. Available: http://doi.acm.org/10.1145/332040.332447

[70] C. U. Smith and L. G. Williams, *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2002.

[71] M. Litoiu, "A performance analysis method for autonomic computing systems," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 2, no. 1, p. 3, 2007. [Online]. Available: http://doi.acm.org/10.1145/1216895.1216898

[72] J. Adams, S. Koushik, G. Galambos, and G. Vasudeva, *Patterns for e-Business: A Strategy for Reuse*. IBM Press, 2001.

[73] C. U. Smith, *Performance Engineering of Software Systems*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990.

[74] M. Woodside, T. Zheng, and M. Litoiu, "Service system resource management based on a tracked layered performance model," in *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, June 2006, pp. 175–184.

[75] H. Ghanbari, C. Barna, M. Litoiu, M. Woodside, T. Zheng, J. Wong, and G. Iszlai, "Tracking adaptive performance models using dynamic clustering of user classes," in *2nd ACM International Conference on Performance Engineering (ICPE 2011)*. New York, NY, USA: ACM, 2011.

[76] "Java Management eXtension (JMX)." http://java.sun.com/products/JavaManagement/.

[77] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[78] R. Kazman, M. Klein, and P. Clements, "Atam: Method for architecture evaluation," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2000-TR-004, 2000. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=5177

[79] R. Kazman, J. Asundi, and M. Klein, "Quantifying the costs and benefits of architectural decisions," in *Proceedings of the 23rd International Conference on Software Engineering*, ser. ICSE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 297–306. [Online]. Available: http://dl.acm.org/citation.cfm?id=381473.381504

[80] J. Asundi, R. Kazman, and M. Klein, "Using economic considerations to choose among architecture design alternatives," DTIC Document, Tech. Rep., 2001.

[81] J. Kramer and J. Magee, "Self-managed systems: an architectural challenge," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 259–268.

[82] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Runtime software adaptation: Framework, approaches, and styles," in *Companion of the 30th International Conference on Software Engineering*, ser. ICSE Companion '08. New York, NY, USA: ACM, 2008, pp. 899–910. [Online]. Available: http://doi.acm.org/10.1145/1370175.1370181

[83] J. C. Georgas and R. N. Taylor, "Policy-based self-adaptive architectures: A feasibility study in the robotics domain," in *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, ser. SEAMS '08. New York, NY, USA: ACM, 2008, pp. 105–112. [Online]. Available: http://doi.acm.org/10.1145/1370018.1370038

[84] D. Sykes, W. Heaven, J. Magee, and J. Kramer, "From goals to components: A combined approach to self-management," in *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, ser. SEAMS '08. New York, NY, USA: ACM, 2008, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/1370018.1370020

[85] P. Oreizy, N. Medvidovic, and R. N. Taylor, "Architecture-based runtime software evolution," in *Proceedings of the 20th International Conference on Software Engineering*, ser. ICSE '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 177–186. [Online]. Available: http://dl.acm.org/citation.cfm?id=302163.302181

[86] T. Vogel and H. Giese, "Adaptation and abstract runtime models," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '10. New York, NY, USA: ACM, 2010, pp. 39–48. [Online]. Available: http://doi.acm.org/10.1145/1808984.1808989

[87] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A multi-model framework to implement self-managing control systems for qos management," in *Proceed-*

*ings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 218–227.

[88] Z. Wang, X. Zhu, and S. Singhal, "Utilization and slo-based control for dynamic sizing of resource partitions," in *Ambient Networks*, ser. Lecture Notes in Computer Science, J. Schnwlder and J. Serrat, Eds. Springer Berlin Heidelberg, 2005, vol. 3775, pp. 133–144. [Online]. Available: http://dx.doi.org/10.1007/11568285_12

[89] X. Zhu, Z. Wang, and S. Singhal, "Utility-driven workload management using nested control design," in *American Control Conference, 2006*. IEEE, 2006.

[90] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty, *Nonlinear Programming: Theory and Algorithms*, 2nd ed. Wiley-Interscience, 1993.

[91] G. Balbo and G. Serazzi, "Asymptotic analysis of multiclass closed queueing networks: multiple bottlenecks," *Performance Evaluation*, vol. 30, no. 3, pp. 115–152, 1997. [Online]. Available: http://dx.doi.org/10.1016/S0166-5316(97)00005-9

[92] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[93] Y. Lu, T. Abdelzaher, C. Lu, L. Sha, and X. Liu, "Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers," in *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2003, p. 208.

[94] D. A. Menascé and M. N. Bennani, "On the use of performance models to design self-managing computer systems," in *Proceedings of the Computer Measurement Group Conference*, 2003, pp. 7–12.

[95] A. Bogárdi-Mészöly, T. Levendovszky, and A. Szeghegyi, "Improved performance models of web-based software systems," in *INES'09: Proceedings of the IEEE* $13^{th}$ *international conference on Intelligent Engineering Systems*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 23–28.

[96] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining performance regression testing repositories for automated performance analysis," in *10*$^{th}$ *International Conference on Quality Software (QSIC)*, July 2010, pp. 32–41. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2010.35

[97] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proceedings of the International Conference on Software Maintenance (ICSM).*, 2008, pp. 307–316. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2008.4658079

[98] H. Malik, B. Adams, A. E. Hassan, P. Flora, and G. Hamann, "Using load tests to automatically compare the subsystems of a large enterprise system," in *Proceedings of the 34*$^{th}$ *Annual Computer Software and Applications Conference*, ser. COMPSAC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 117–126. [Online]. Available: http://dx.doi.org/10.1109/COMPSAC.2010.18

[99] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef, "Performance management for cluster based web services," in *Integrated Network*

*Management VIII*, ser. The International Federation for Information Processing (IFIP), G. Goldszmidt and J. Schnwlder, Eds. Springer US, 2003, vol. 118, pp. 247–261. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-35674-7_29

[100] D. A. Menascé, H. Ruan, and H. Gomaa, "Qos management in service-oriented architectures," *Perform. Eval.*, vol. 64, no. 7-8, pp. 646–663, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2006.10.001

[101] A. Kraiss, F. Schoen, G. Weikum, and U. Deppisch, "Towards response time guarantees for e-service middleware," *Bulletin of the Technical Committee on*, p. 58, 2001.

[102] D. Perez-Palacin, J. Merseguer, and S. Bernardi, "Performance-aware open-world software in a 3-layer architecture," in *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. New York, NY, USA: ACM, 2010, pp. 49–56. [Online]. Available: http://dx.doi.org/10.1145/1712605.1712614

[103] W. G. Morein, A. Stavrou, D. L. Cook, A. D. Keromytis, V. Misra, and D. Rubenstein, "Using graphic turing tests to counter automated DDoS attacks against web servers," in *Proceedings of the 10$^{th}$ ACM conference on Computer and communications security*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 8–19. [Online]. Available: http://dx.doi.org/10.1145/948109.948114

[104] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A survey of intrusion detection techniques in cloud," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 42 – 57, 2013.

[105] C. Barna, M. Litoiu, and H. Ghanbari, "Autonomic load-testing framework," in *Autonomic Computing, 2011. International Conference on*, ser. ICAC '11. New York, NY, USA: ACM, June 2011, pp. 91–100. [Online]. Available: http://dx.doi.org/10.1145/1998582.1998598

[106] M. Litoiu and C. Barna, "A performance evaluation framework for web applications," *Journal of Software: Evolution and Process*, 2012. [Online]. Available: http://dx.doi.org/10.1002/smr.1563

[107] S. Oshima, T. Nakashima, and T. Sueyoshi, "Early dos/ddos detection method using short-term statistics," in *Complex, Intelligent and Software Intensive Systems (CISIS)*, 2010, pp. 168–173. [Online]. Available: http://dx.doi.org/10.1109/CISIS.2010.53

[108] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu, "Model-based adaptive dos attack mitigation," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on*, ser. SEAMS 2012. New York, NY, USA: ACM, 2012, pp. 119–128. [Online]. Available: http://dx.doi.org/10.1109/SEAMS.2012.6224398

[109] G. Galante and L. de Bona, "A survey on cloud computing elasticity," in *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, 2012, pp. 263–270. [Online]. Available: http://dx.doi.org/10.1109/UCC.2012.30

[110] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Exploring alternative approaches to implement an elasticity policy," in *Proceedings of the 4th IEEE*

*International Conference on Cloud Computing*. Washington DC, USA: IEEE, 2011.

[111] M. Maurer, I. Brandic, and R. Sakellariou, "Enacting slas in clouds using rules," in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, ser. Euro-Par'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 455–466.

[112] P. Marshall, K. Keahey, and T. Freeman, "Elastic site: Using clouds to elastically extend site resources," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 43–52.

[113] E. Caron, L. Rodero-Merino, F. Desprez, and A. Muresan, "Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds," INRIA, Rapport de recherche RR-7857, 2012.

[114] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1:1–1:39, 2008. [Online]. Available: http://dx.doi.org/10.1145/1342171.1342172

[115] H. Ghanbari, B. Simmons, M. Litoiu, C. Barna, and G. Iszlai, "Optimal autoscaling in a iaas cloud," in *Proceedings of the 9th International Conference on Autonomic Computing*, ser. ICAC '12. New York, NY, USA: ACM, 2012, pp. 173–178. [Online]. Available: http://dx.doi.org/10.1145/2371536.2371567

[116] M. Smit and E. Stroulia, "Autonomic configuration adaptation based on simulation-generated state-transition models," in *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, Aug 2011, pp. 175–179.

[117] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[118] M. Shtern, R. Sandel, M. Litoiu, C. Bachalo, and V. Theodorou, "Towards mitigation of low and slow application ddos attacks," in *2014 IEEE International Conference on Cloud Engineering (IC2E)*, March 2014, pp. 604–609. [Online]. Available: http://dx.doi.org/10.1109/IC2E.2014.38

[119] J. Idziorek and M. Tannian, "Exploiting cloud utility models for profit and ruin," in *IEEE International Conference on Cloud Computing*, 2011, pp. 33–40.

[120] M. Shtern, M. Smit, B. Simmons, and M. Litoiu, "A runtime cloud efficiency software quality metric," in *New Ideas and Emerging Results (NIER) track, Proc. of the 2014 Intl. Conference on Software Engineering (ICSE)*, 2014.

[121] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu, "Mitigating DoS attacks using performance model-driven adaptive algorithms," *Transactions on Autonomous and Adaptive Systems*, vol. 9, no. 1, pp. 3:1–3:26, Mar. 2014. [Online]. Available: http://doi.acm.org/10.1145/2567926

[122] J. Schad, J. Dittrich, and J.-A. Quiane-Ruiz, "Runtime measurements in the cloud: Observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1, 2010.

[123] C. Barna, M. Shtern, M. Smit, V. Tzerpos, and M. Litoiu, "Model-based adaptive dos attack mitigation," in *ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, ser. SEAMS 2012. New York, NY, USA: ACM, 2012, pp. 119–128.

[124] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring of heterogeneous clouds using stream processing," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2103–2114, 2013.

[125] J. Z. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, "Performance model driven QoS guarantees and optimization in clouds," in *in Proceedings of Workshop on Software Engineering Challenges in Cloud Computing @ ICSE 2009*, 2009.

[126] D. Bacigalupo, J. van Hemert, A. Usmani, D. Dillenberger, G. Wills, and S. Jarvis, "Resource management of enterprise cloud systems using layered queuing and historical performance models," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010, pp. 1–8.

[127] B. Suleiman and S. Venugopal, "Modeling performance of elasticity rules for cloud-based applications," in *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International*, Sept 2013, pp. 201–206.

[128] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, "Adaptive resource provisioning for read intensive multi-tier applications in the cloud," *Future Generation Computer Systems*, vol. 27, no. 6, pp. 871–879, 2011.

[129] K. Xiong and H. Perros, "Service performance and analysis in cloud computing," in *Services-I, 2009 World Conference on*. IEEE, 2009, pp. 693–700.

[130] R. Han, M. M. Ghanem, L. Guo, Y. Guo, and M. Osmond, "Enabling cost-aware and adaptive elasticity of multi-tier cloud applications," *Future Generation Computer Systems*, vol. 32, pp. 82–98, 2014.

[131] K. J. Aström and R. M. Murray, *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2010.

[132] Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, and M. Smit, "A design space for self-adaptive systems," in *Software Engineering for Self-adaptive Systems II*. Springer, 2013, pp. 33–50.

[133] D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva, "Control theory for model-based performance-driven software adaptation," in *Proceedings of the 11$^{th}$ International ACM SIGSOFT Conference on Quality of Software Architectures*. ACM, 2015, pp. 11–20.

[134] T. Zheng, J. Yang, M. Woodside, M. Litoiu, and G. Iszlai, "Tracking time-varying parameters in software systems with extended kalman filters," in *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2005, pp. 334–345.

[135] T. Söderström and P. Stoica, *System identification*. Prentice-Hall, Inc., 1988.

[136] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," *Cluster computing*, vol. 12, no. 1, pp. 1–15, 2009.

[137] R. Kalman, "On the general theory of control systems," *IRE Transactions on Automatic Control*, vol. 4, no. 3, pp. 110–110, 1959.

[138] R. de Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker *et al.*, "Software engineering for self-adaptive systems," in *Dagstuhl Seminar*, vol. 10431. Springer, 2009.

[139] M. Litoiu, "Autonomic computing-scientific and engineering challenges," in *Proceedings of the ACM ICSE 4$^{th}$ International Workshop on Adoption-Centric Software Engineering*. Edinburgh, Scotland: ACM, 2004, pp. 1–7.

[140] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *Proceedings of the 6$^{th}$ International Conference on Autonomic Computing*, ser. ICAC '09. New York, NY, USA: ACM, 2009, pp. 117–126. [Online]. Available: http://dx.doi.org/10.1145/1555228.1555261

[141] T. F. Abdelzaher, J. A. Stankovic, C. Lu, R. Zhang, and Y. Lu, "Feedback performance control in software services," *Control Systems, IEEE*, vol. 23, no. 3, pp. 74–90, June 2003. [Online]. Available: http://dx.doi.org/10.1109/MCS.2003.1200252

[142] A. Filieri, H. Hoffmann, and M. Maggio, "Automated design of self-adaptive software with control-theoretical formal guarantees," in *Proceedings of the*

*36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 299–310. [Online]. Available: http://dx.doi.org/10.1145/2568225.2568272

[143] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, "Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server," in *Network Operations and Management Symposium*, ser. NOMS 2002, 2002, pp. 219–234. [Online]. Available: http://dx.doi.org/10.1109/NOMS.2002.1015566

[144] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Feedback-based optimization of a private cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 104–111, January 2012. [Online]. Available: http://dx.doi.org/10.1016/j.future.2011.05.019

[145] D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva, "Control theory for model-based performance-driven software adaptation," in *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, ser. QoSA '15. New York, NY, USA: ACM, 2015, pp. 11–20. [Online]. Available: http://dx.doi.org/10.1145/2737182.2737187

[146] T. Patikirikorala, A. Colman, J. Han, and L. Wang, "A multi-model framework to implement self-managing control systems for qos management," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2011, pp. 218–227.

[147] C. M. Woodside, T. Zheng, and M. Litoiu, "Performance model estimation and tracking using optimal filters," *IEEE Transactions on Software*

*Engineering*, vol. 34, no. 3, pp. 391–406, 2008. [Online]. Available: http://dx.doi.org/10.1109/TSE.2008.30

[148] A. Filieri, L. Grunske, and A. Leva, "Lightweight adaptive filtering for efficient learning and updating of probabilistic models," *ICSE. IEEE*, 2015.

[149] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on.* IEEE, 2009, pp. 111–121.

[150] H. Ghanbari, M. Litoiu, P. Pawluk, and C. Barna, "Replica placement in cloud through simple stochastic model predictive control," in *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on.* IEEE, 2014, pp. 80–87.

[151] R. E. Kalman *et al.*, "Contributions to the theory of optimal control," *Bol. Soc. Mat. Mexicana*, vol. 5, no. 2, pp. 102–119, 1960.

[152] B. Solomon, D. Ionescu, M. Litoiu, and G. Iszlai, "Autonomic computing control of composed web services," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '10. New York, NY, USA: ACM, 2010, pp. 94–103. [Online]. Available: http://dx.doi.org/10.1145/1808984.1808995

[153] B. D. Anderson and J. B. Moore, *Optimal control: linear quadratic methods*. Courier Corporation, 2007.

[154] M. Smit, B. Simmons, and M. Litoiu, "Distributed, application-level monitoring of heterogeneous clouds using stream processing," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 2103–2114, 2013.

[155] C. Barna, H. Ghanbari, M. Litoiu, and M. Shtern, "Hogna: A platform for self-adaptive applications in cloud environments," in *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2015 IEEE/ACM 10th International Symposium on*, May 2015, pp. 83–87. [Online]. Available: http://dx.doi.org/10.1109/SEAMS.2015.26

[156] "Octave optim package v1.4.1," http://octave.sourceforge.net/optim/overview.html, 2015.

[157] L. Reichlin, "Octave control package v2.3.8," http://octave.sourceforge.net/control/overview.html, 2015.

[158] J. P. Hespanha, P. Naghshtabrizi, and Y. Xu, "A survey of recent results in networked control systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 138–162, Jan 2007. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2006.887288

[159] Q. Zhang, Q. Zhu, M. Zhani, and R. Boutaba, "Dynamic service placement in geographically distributed clouds," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, June 2012, pp. 526–535. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2012.74