
Autonomic Mechanisms in Cloud Computing Ecosystems

HAMOUN GHANBARI

QUALIFYING EXAM REPORT

SUPERVISOR: MARIN LITOIU

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

YORK UNIVERSITY

TORONTO, ONTARIO

August 30, 2011

Abstract

As current distributed computing systems evolve the eventual emergence of Ultra Large Scale (ULS) systems poses new challenges. Such massive and pervasive systems of systems (SoS) will require that new and innovative management techniques be developed in order to handle this dramatic increase in complexity and scale. Autonomic computing attempts to solve the problem of managing complexity through engineering certain properties, referred to as self-* properties (e.g. self-adapting, self-optimizing, self-tuning, self-healing, self-configuring, self-organizing and self-protecting, into the elements of a system. The focus of this research is designing and experimenting with various optimization techniques facilitating the autonomic management of complex computing systems.

Contents

1	Introduction	5
1.1	Introduction	5
1.1.1	Individual Multi-Tenant Resources, Building Blocks of ULSRSS	7
1.2	Ultra large scale Coordinated Resource Sharing	11
1.2.1	Outline	13
2	Elements of Autonomic Control and Adaptive Systems	14
2.1	Monitoring	16
2.2	Analyzing	17
2.2.1	Parametric Models	19
2.2.2	Performance Models	22
2.2.3	Discovering Hidden Parameters of Performance Model	29
2.3	Planning and Feed-forward Predictive Control	31
2.3.1	Searching in continuous space	32
2.3.2	Searching in discrete space	33
2.4	Execution through Regulator feedback loops	34
2.5	Other ways of implementing managers	36
2.5.1	Policy based autonomic management	36

2.6	Summary	38
3	Cloud Computing: Initial Steps Towards ULSRSS	40
3.1	Building Blocks of Current Clouds	41
3.1.1	Network	42
3.1.2	Storage	43
3.1.3	Server farm	43
3.2	Use cases of Clouds	44
3.2.1	E-Commerce Websites	44
3.2.2	Data Intensive Processing with Clusters	45
3.3	Infrastructure as a Service (IaaS)	46
3.3.1	Management of Virtual Infrastructures	48
3.4	Storage as a Service	51
3.5	Computing as a Service	51
3.6	Network as a Service	53
3.6.1	Bandwidth pricing	55
3.7	Platform as a Service (PaaS)	56
3.7.1	PaaS Provider Economic Model	57
3.7.2	Multiplexing and consolidation: managing multiple services	58
3.8	Software as a Service (SaaS)	60
3.9	Summary	63
4	Conclusion	65
4.1	Open Problems	66
4.1.1	Performance Model Identification of Service Instances in a Ultra Large Resource Sharing System	66

4.1.2	Optimization of Resource Distribution to Applications in Cloud . . .	67
4.1.3	Comparing Alternatives Approaches to Autonomic Resource Man- agement	68
4.1.4	Hybrid Autonomic Management Techniques	69

List of Figures

2.1	Architecture of autonomic management suggested by IBM.	15
2.2	Effect of memory on throughput not considering swapping and paging. . . .	25
2.3	Effect of multiprogramming on throughput.	26
2.4	Multiclass Layered Queuing Model of a Web application.	29
2.5	Structure of an optimization based planner.	32
2.6	The general architecture of resource allocation controller.	35
2.7	Structure of an LQG regulator.	35
2.8	Architecture of a rule-based autoscaler.	38
3.1	The interaction of layers in our optimization mechanism.	58
3.2	Smooth service level utility function; vertical line indicates the service level objective of a platform (as defined in SLA).	61

Chapter 1

Introduction

1.1 Introduction

Ultra Large Scale (ULS) systems, a term coined by researchers at Carnegie Mellon's Software Engineering Institute, refers to a system composed of a large set of systems with a variety of stakeholders communicating and operating to satisfy separate (possibly conflicting) goals. Examples of ULS systems are Internet, International Telephone System, and United States electric Grid.

As a special case in computer science domain, an Ultra Large Scale Resource Sharing System (ULSRSS) aims at ultra large scale delivery of on-demand computational power (specifically network storage, computational devices, and services on top of these) to user community. Although such a system currently does not exist, it is largely believed that as current large scale resource sharing systems (RSS) such as Grid[1, 2] and Cloud[3, 4, 5, 6] computing evolve and grow, ULSRSS will emerge.

Grid, risen from high-performance computing (HPC) community near 2000, targets delivering on-demand computational power using a unified network of loosely coupled com-

puters in the form of a "super virtual computer." The aim of designers was to let users plug their own programs into the infrastructure, and use resources for a desired duration. Although computational grid can be a very general concept, current implementations usually target long running resource intensive applications (also called jobs) that are submitted by a few users. Examples of these applications are distributed simulation [7, 8], scientific visualization [9], continual queries [10, 11], video conferencing [12], and transcoding [13].

Cloud computing is another manifestation of large scale RSS that, according to many, has caught on in mainstream enterprises. In Cloud computing, large condensed data centers, possibly operated by multiple stakeholders, are offered at different levels of abstraction (e.g. infrastructure (IaaS), platform (PaaS) and software (SaaS)) on-demand as commodities to a large community of users. In cloud there is usually a clear distinction between provider and customer and the extent of sharing is governed by economical rules and pricing scheme (usually pay-as-you-go).

Although Cloud and Grid are successful examples of RSS, the ultimate departure to ULSRSS can only be realized with the emergence of new and innovative management techniques. These management techniques should be able to handle dramatic increases in complexity and scale; the complexity that will rise from the interplay of a large number of sub-systems [14, 15], stakeholders, and users in a massive and pervasive systems of systems (SoS).

As it turns out, current management techniques able to deal with the complexity of these ULSRSS systems are too costly since they usually rely on human actors. Autonomic computing attempts to solve the problem of managing complexity through engineering certain properties, referred to as self-* properties (e.g. self-adapting, self-optimizing, self-tuning, self-healing, self-configuring, self-organizing and self-protecting) into the elements of a system. *Adaptive systems* (the outcome of employing autonomic computing concepts) are

able to move toward certain objectives despite sudden irregularities that might happen during their lifetime. Autonomic computing so far has been successfully applied to many RSS related areas, including (i) fault tolerance, (ii) smart provisioning in the presence of workload spikes, (iii) efficient resource allocation, consolidation, and user multiplexing, (iv) smart power consumption, and (v) business objective driven optimization. For example virtual-market oriented techniques, using numerous simple actors interacting with money, can be used to maintain balance among congested nodes in a grid infrastructure [1, 2].

A current challenge is to identify a set of contributions that autonomic computing is able to make to ULSRSS systems and to design such systems in a scalable fashion through the developed knowledge of autonomic computing. The focus of this research is designing and experimenting with various optimization techniques facilitating the autonomic management of complex resource sharing computing systems.

1.1.1 Individual Multi-Tenant Resources, Building Blocks of ULSRSS

Resource sharing is not a new concept; even the earliest computers were built with some sharing mechanism built-in. Before the age of personal computer (PC) s, hundreds of users accessed mainframes through dummy terminals. Even, most PCs come with a multi-tenant operating system. Study of the extent to which our current IT systems and resources implement sharing capabilities would help in answering the question of how ULSRSS rises from individual resource sharing systems. In this subsection, we investigate multi-tenancy features in commonly used IT resources.

Multiplexing computational resource

At the hardware level computational resources composed of CPU, RAM can be multiplexed through virtualization techniques. There are several techniques for virtualization and par-

avirtualization of hardware resources for operating systems. Each of the operating systems installed in a provided virtual machine takes a configurable share of hardware. A scheduler within the virtual machine manager (also called hypervisor) entitles operating systems to the configured hardware resource shares. The hypervisor usually uses hardware traps; on interrupt events, the control of CPU will be handed to interrupt handler code that is patched by the hypervisor. This code is responsible for switching execution among operating systems.

OS level multiplexing occurs among a set of processes. An operating system is responsible for scheduling these processes to use different system resources. Several scheduling policies can be employed by an operating system. Shortest job first (SJF) and fair sharing are examples. The trap mechanism for an OS is system calls and interrupts. Whenever an OS gets a system call, it can assess the possibility of switching among the processes. Another aspect of OS level multiplexing is memory management; the OS has to somehow select and bring the appropriate portion of the virtual memory requested by active processes to the available physical memory, given by the hardware or virtualization layer. This ongoing selection and substitution is governed by different paging and swapping policies. An efficient paging policy and memory multiplexing will result in more locality of reference from actual memory and less overhead.

Multiplexing computing clusters

Sharing resources might be expanded from a physical computer to a set of computers that are perceived as a single virtual computer. This type of computing clusters is managed by a 'container' Container instances are installed a top computers of the cluster. There are different strategies to implement containers; one can run a container as one or several single or multi-threaded OS processes. In web or database containers where multiplexing

occurs at container level users are distinguished by container but not by operating system. These containers usually partition their logical resources and maintain a user to resource mapping. When handling requests resource entitlement can be enforced by controlling the properties of requests queues (e.g. concurrent active requests) and other decisions. As an example, in a database system users are authenticated separately, and can query a set of tables or execute stored procedures that they have the right to access. DBMS system might be able to restrict the number of concurrent connections from specific users to manage utilization of its resources. In a web container, users are not usually distinguished; however, resources can be partitioned into separate applications and applications can have different configurations for access. Clustering on top of these containers can be performed using a load-balancing layer in front of container machines or in the same machines as the cluster.

Multiplexing a Network

A network is inherently a multi-tenant resource. IP networks are composed of thousands of routers connecting local networks. Once a client packet leaves its originating device, it will share the network medium with millions of other users. For example, a smart-phone connected to a 3G network shares the mobile cell spectrum with others. Once a packet reaches the first switch, it can be handed to one of the routers working as a network gateway along with millions of other data frames. The router decides on the path that this packet should be forwarded. Routers along the path route millions of packets every second, usually without even knowing the originating and destination device (since routing tables are aggregated according to the IP hierarchical addressing based on the location of a router in the network graph). This partially explains why despite several tries (e.g. IntServ, MPLS, etc.) bandwidth guarantees for specific users at the Internet scale is impossible.

Storage multi-tenancy

Storage can be multiplexed among several users in different ways. In the lowest level, a storage device, potentially composed of several disks, can be accessed concurrently by multiple computing nodes. This is usually achieved with block I/O protocols such as a Fiber Channel or an Internet SCSI (iSCSI) and is called Storage Area Network (SAN). For example, compute nodes treat iSCSI logical units (identified by a Logical Unit Number or LUN) the same way as they would a raw SCSI or IDE hard drive (i.e. they can format and manage file-systems). In SAN, managing the concurrent access to storage units is left to compute nodes. In this case, the concurrency control for shared storage devices can be handled by operating systems.

In 'cluster file systems' (e.g. Red Hat Global File System (GFS) [16]) translation of file-level operations (requested by user processes) to block-level ones (to be performed on SAN) in each node is done in coordination with other compute nodes (e.g. using distributed locking or centralized coordinator). This provides consistent and serializable view of the file system, and avoids corruption and unintended data loss despite concurrent access by nodes.

In case of virtualized environment, where VMs of different users might use different file systems, access to storage devices are coordinated and isolated through hypervisors on physical machines (see Parallax [17] and VMware's VMFS [18]). On top of this virtualized isolated block device, cooperating VMs can still employ cluster file systems for sharing. Other trends in storage sharing are (i) Network Attached Storage (NAS) where both storage and a file system provided by the same device and managed centrally and (ii) distributed file systems (e.g. Hadoop Distributed File System or HDFS [19]) where file level access is provided over a set of non-shared block level storage nodes each operated by a compute node.

In the higher levels of storage such as databases, information is organized in schemas. Each schema is composed of a metadata, data files, and indexes. Here because of existence of metadata, the information access mediator (here database process) has a more granular control over data and can thus employ dataset or record level locking. In multi user distributed database systems where data sets are replicated over wide area networks where transactions needs to be supported, several optimizations for lowering latency (e.g. loosening consistency requirement[20]) have been employed.

1.2 Ultra large scale Coordinated Resource Sharing

It is one thing to have a set of multi-tenant devices and systems and something else to make a multi-tenant ecosystem out of those. Currently most of these resources on the planet are handed out to individuals or organizations having them working in isolation. This results in less overall utilization and loss of investment. Future computing, demands techniques that support large scale sharing of devices aiming at full utilization and efficiency. This way in micro-scale, the initial investment for such devices, the cost of operation (e.g. in terms of space, power, cooling), and cost of third party subscribed resources and services (like Internet) will be partially returned. In macro scale, it results in a better utilization of current IT resources in general, lowering overall shipped and idle computing devices and a greener IT for the planet.

In a large scale resource sharing environment, users have coordinated access to massive amount computers, files, programs and software, data, sensors, network and other resources. The granularity, complexity, size, and scope of shared resources can be different. Internet (network of networks) that provides a unified shared medium for communication and information exchange can be itself viewed as a resource in the hypothetical universal

LSRSS.

An application or a service instance in a LSRSS system can span multiple resources belonging to multiple organizations. It can also have numerous users or stake-holders that pay for these resources. In some cases, such as many Grid computing examples, users, stake-holders, and resource providers are the same entities. As an example consider a collection of geographical databases and earthquake simulation systems that is hosted on a set of computers at different universities to be used by collaborative researchers. Most of the time unlike this example, however, users of application or service instance, stake-holders and resource providers are different entities that have different responsibilities, and economic benefits. An example is a social networking website that is deployed on several datacenters by a company for making profit and used by millions of users worldwide. In this view, regardless the roles each entity plays, every service instance or application can be considered as a collaboration point for several parties, for certain period of time (possibly unlimited) that makes use of (hypothetically) unified network of computing resources on the planet, the ULSRSS.

An important set of tools, services, and techniques are focused on integrated environments for coordinated sharing in which participants of resource sharing are assisted in making choices of getting optimal sets of resources that satisfies their objective; these tools might also help providers in maximizing utilization of their resources. In environments where there is a clear distinction between providers and users, these techniques should consider mutual benefit of both resources providers and consumers in the ULSRSS system. providers optimize for more utilization and more customers, and customers optimize for minimal resource that can satisfy the jobs. Later in chapter 3, in the most well-known candidate of ULSRSS, cloud computing, some of the techniques to optimize for providers will be discussed. However, considering the whole consumer provider environment as an

ecosystem the aim of future research should be to come up with better strategies that result in mutual satisfaction of all participated stake-holders and customers.

1.2.1 Outline

This work focuses on explaining autonomic computing in depth, its contribution in building ULS systems essentially those large scale multi-tenant networks of IT (computing, storage, etc) resources. In addition, we present our initial results in this area. As an example, we look at the application of autonomic computing in delivery of utility computing through Cloud. In this work, feedback based loops are utilized to reach the desired objectives, both for users and providers. Chapter 2 presents an overview of autonomic computing paradigm, a way to build self-managing complex computing systems. Chapter 3 demonstrates Cloud computing as the current state of art in ULSRSS systems. Different cloud offerings including infrastructure (IaaS), platform (PaaS) and software (SaaS) as a service and the enabling technology of each offering is discussed, essentially in terms of resource allocation and multi-tenancy mechanism. Price models and economic objectives of these offerings from user and provider perspective are also discussed in each type of offering. Finally, chapter 4 concludes the work and discusses interesting problems for future research in this domain.

Chapter 2

Elements of Autonomic Control and Adaptive Systems

Autonomic computing, a term introduced by IBM [21], indicates systems that are self-managing, self-tuning, self-healing, self-protecting, self-adapting, self-configuring, and self-organizing (briefly called self-* systems) [22]. Some examples of IT related self-* areas of research targeted so far are: adaptive parameter-level configuration management [23, 24, 25], adaptive client-server communication [26, 27, 28], adaptive resource allocation [29, 30], self-configuring network services [31], workload adaptive services [32], self-managing storage [33], statistical inference based decision-making management [34], and change and configuration management [35].

The portion of autonomic computing discipline that is relevant to our work is building self-managing complex resource sharing systems that manage themselves in accordance with high-level objectives specified by humans [36]. Thus the autonomic aspect focuses on the fact that management is performed the systems themselves rather than being actively performed by human actors. However, this management should be according to objectives

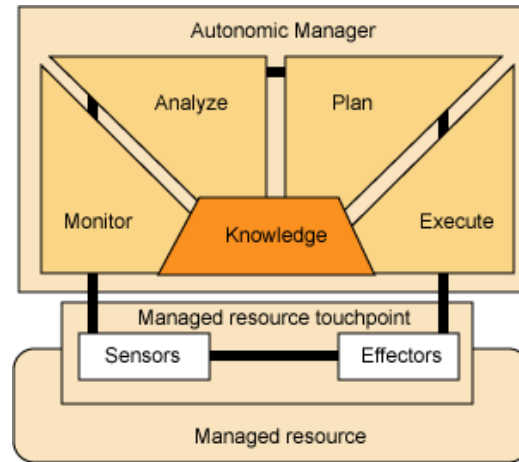


Figure 2.1: Architecture of autonomic management suggested by IBM.

expressed by administrators. An autonomic manager usually understands the desired system objectives, matches those objectives with current or forecasted system behavior, and incorporates these into its decision governing the system.

In terms of architecture, there is variety of ways for implementing autonomic management systems. In the one focused in this research, adaptation strategies and mechanisms are separated from the applications or systems [37, 38] (i.e. externalized adaptation). The architecture used is the well-known Monitor-Analyze-Plan-Execute (MAPE) loop suggested by IBM [21] where several components such as an analyzer, automated learner, forecaster, and a planner are used to decide about proper action(s) to be taken by actuators based on the current system measures. Figure 2.1, adopted from [39], presents a schematic structure of such loop. In next sections, we describe the elements of autonomic computing loop in context of managing computer systems.

2.1 Monitoring

Monitoring subsystem is responsible for measuring inputs, and outputs of the managed system, quantifying them, sometimes aggregating them, and keeping them as a history. Another feature of a complete monitoring system would be to send notifications to inform the administrator or autonomic manager about an important condition (e.g. high CPU load, full memory, or a failed health check) using the network. These conditions can be user-defined and based on performance values breaching certain thresholds. The monitoring data will be processed in subsequent autonomic management subsystems and eventually makes its way into decisions taken by the autonomic manager.

In computer systems, from performance point-of-view, there are several metrics to be monitored. For example, in a typical transactional system one can use hardware level metrics of each host, operating system metrics (e.g. file, network, and memory management subsystems) of each virtual machine, process level metrics, and service specific metrics for different types of server processes (e.g. load balancer, web server, application server, and database server). In a batch oriented system, in addition to hardware and OS level metrics, metrics such as the average number of jobs in the job queue or average job's queuing time also exist. Table 2.1 summarizes these metrics.

There are several existing tools for monitoring computer systems. *Collectd* [40] is a monitoring daemon for collecting system performance statistics. It accepts plug-ins for collecting new types of metrics. *Nagios* [41] is another powerful industry level tool for monitoring entire IT infrastructure. Java Management Extensions (JMX) [42] technology included in the Java SE platform provides a generic standard for publishing and using the data of monitoring devices especially Java base web and application containers.

Alert Target	Metrics
Hardware	CPU utilization, disk access, network interface access, memory usage
General OS Process	cpu-time, pagefaults, real-memory (resident set) size
Load balancer	request queue length, session rate, number of current sessions, transmitted bytes, num of denied requests, num of errors
Web server	transmitted bytes and requests, number of connections in specific states (e.g. closing, sending, waiting, starting, ...)
Application server	total threads count, active threads count, used memory, session count
Database server	number of active threads, number of transactions in (write, commit, roll-back, ...) state
Message Queue	average number of jobs in the queue, average job's queuing time

Table 2.1: List of the metrics used in defining autoscaling alerts.

2.2 Analyzing

The *analyzer* subsystem of autonomic management loop targets identification of system under management. This identification enables the autonomic manager to project system's behavior and state under different actions in the future. One way to gain this identification is through obtaining a model of the system. This model can take several forms such as symbolic representation, neural nets, fuzzy rules, or statistical model. In an autonomic system, however the major concern is the ability to update and synchronize the model based on observed behavior of the system. The major focus of this chapter is to describe the use of statistical techniques to obtain this model. The major benefit of statistical techniques is the ability to incorporate the data in modeling; other techniques not described, however, might be equally or more effective in different situations; for example when it comes to encoding human knowledge (i.e. experts) other techniques such as neuro-fuzzy techniques have been successful in providing both data driven learning and experts knowledge encoding [43].

In statistical modeling, one tries to quantify behavior of the system in terms of a set of relationships between numerical (or nominal) variables. The general structure of the model is chosen first (e.g. broad categories of deterministic or stochastic, and continuous-time, discrete time, or steady-state) and then model is identified by estimating its parameters based on available data. The estimation is usually carried out using techniques based on Least Square Estimation (LSE) or Maximum Likelihood Estimation (MLE). In MLE, one should be able to compute likelihood of certain model given a certain observation (i.e. Assuming a model, how likely is it that we have certain observation?). Based on this, it tries to choose model parameters that maximize likelihood of data given the model, (i.e. $P(data|model)$).

Having the model identified, there are several possibilities of how to use it: (i) given a model, and an observable portion of samples, one can find the hidden portion that is most likely to be observed (estimation). (ii) when model is sequential and incorporates time, given a set of sequentially observed samples one can predict the samples most likely to be observed in future (forecasting). (iii) if model is built using some underlying theory, and latent variables used in theory are discovered, other results of theory can be derived. In an autonomic management system, all these types of uses might be incorporated. For example, one might use a dynamically updated model of the system with some estimated hidden variables and forecasted inputs to project system behavior in future under different configurations and find the optimal behavior that results in satisfaction of the given objectives. If the system is characterized in terms performance modeling theories, better reasoning can be done in terms of concepts of the theory. For example, if performance of an application is modeled by a multi-class queuing system that maps service demands for classes of users and their workloads to their response time, the resulting model can be used to predict the saturation point of the system (e.g. the knee in the performance curve) with respect to vari-

ous workloads. In the rest of this section, different models and identification techniques are explored.

2.2.1 Parametric Models

In the parametric form, the model is assumed known and system behavior is captured using standard system identification techniques. These approaches are useful when an exact theory underlying the system is not available and the relationship between system inputs and outputs is simple.

Regression models [44, 45] relate a dependent variable (i.e. outputs Y) to independent variables (i.e. inputs X) as follows:

$$Y \approx f(X, \beta) \quad (2.1)$$

where β is a scalar or a vector representing model parameter(s) to be identified using identification and estimation techniques from a given data set. Using the identified model, for each given sample of observed variables X the dependent variable Y can be estimated.

Autoregressive integrated moving average (ARIMA) models combined with identification approaches like Box-Jenkins [46] are used for forecasting of sequential data or time series such as environmental inputs to a system like workload in e-commerce and web applications. A general forecasting model ω that computes estimated value $\hat{\omega}(k)$ based on set of n previously observed values $\omega(k-1, n) = \{\omega(k-1), \dots, \omega(k-n-1)\}$, has the form

$$\hat{\omega}(k) = \varphi(\underline{\omega}(k-1, n), a(k)) \quad (2.2)$$

where $a(k)$ denotes related parameters of the estimation method that might be updated at each time step to minimize the forecasting error $\|\hat{\omega}(k) - \omega(k)\|$. As an example, arrival-rate of users to a system $\hat{\lambda}(k)$ can be estimated using ARIMA model with exponentially weighted moving-average filter:

$$\hat{\lambda}(k) = \beta \cdot \lambda(k) + (1 - \beta) \cdot \hat{\lambda}(k - 1) \quad (2.3)$$

State-Space models are another type of models capable of dealing with sequential data. In state-space models, variables can be divided as input, state, and output (as opposed to input and output). Output variables depend on inputs as well as states. The hidden portion of data to be estimated is associated with state variables (rather than output variables). Observable portion of data is divided into dependent and independent variables (input and output).

The formulation of these models makes them suitable to analyze data that is associated with input and output of a system (in signal processing terms). The following discrete-time state-space equation is a generic model for operating dynamic of a system:

$$x(k + 1) = f(x(k), u(k), \omega(k)) \quad (2.4)$$

$$y(k) = g(x(k), u(k), \omega(k))$$

where $x(k)$ is the system state at time step k , $u(k)$ denotes control input chosen at step k , and $\omega(k)$ is environment parameter at time k . The first equation is called state equation while the second one is called measurement equation.

A linear vector based formulation of same system with p inputs, q outputs and n state

variables in general state-space is:

$$\dot{\mathbf{x}}(t) = A(t)\mathbf{x}(t) + \mathbf{B}(t)\mathbf{u}(t) \quad (2.5)$$

$$\mathbf{y}(\mathbf{k}) = \mathbf{C}(\mathbf{t})\mathbf{x}(\mathbf{t}) + \mathbf{D}(\mathbf{t})\mathbf{u}(\mathbf{t})$$

where \mathbf{x} is 'state vector', \mathbf{y} is 'output vector', \mathbf{u} is 'input (or control) vector', and A, B, C, D are model parameters.

The Kalman filter [47] is a well-known parameter estimator for state-space models of this sort. The filter maintains the estimate of \mathbf{x} and updates it using the linear feedback equation based on new measurements \mathbf{z} :

$$\mathbf{x}(\mathbf{k}) = \mathbf{x}(\mathbf{k}) + \mathbf{K}(\mathbf{z}(\mathbf{k}) - \mathbf{y}(\mathbf{k})) \quad (2.6)$$

where t denotes a discrete time index, $\mathbf{z}(\mathbf{k}) - \mathbf{y}(\mathbf{k})$ denotes the prediction error, and K is the "Kalman Gain" matrix obtained statically from model parameters and updated dynamically based on estimation error (i.e. to construct an adaptive estimator).

As an example a time-varying queuing model can be formulated as a discrete-time state-space equation to estimate current queue length $q(k)$ (system state) in a system based on the average monitored response time $r(k)$ (system output), workload $\lambda(k)$ (system input), and service rate $\mu(k)$ (system input):

$$\hat{q}(k+1) = q(k) + \left(\hat{\lambda}(k) - \mu(k) \right) \cdot T \quad (2.7)$$

$$\hat{r}(k+1) = (1 + \hat{q}(k)) / \mu(k)$$

where T denotes the sampling duration, $\hat{\lambda}(k)$ denotes the predicted arrival rate of

incoming requests at time k , $\hat{q}(k)$ is the estimated queue size at time k , and $\mu(k)$ represents the service rate. Using these two equations, in addition to predicted workload arrival rate $\hat{\lambda}(k)$, estimated service rate, and observed queue length at time k one can estimate queue length at time $k + 1$.

2.2.2 Performance Models

Second category of models used to represent the behavior of computing systems is performance models. These models attempt to describe the expected performance of a system in relation to various inputs, using queuing theory. A performance model is an abstraction that takes the physical layer specification of an application and its workload and map this to various quality attributes (e.g., response time, throughput). Several different models have been suggested for deriving performance metrics of applications (i.e. response time and throughput) based on resource share and workloads. For example queuing theory based models [48, 49, 50] and Layered Queuing Models (LQM) [51, 52] developed upon mean value analysis (MVA) of queuing networks have been vastly used to capture the behavior of multi-tier distributed applications (such as web services) [53, 54, 55, 56].

In the following, we describe some excerpts of these models.

Modeling Workload

The traffic of a transactional application can be modeled as either *closed* or *open*. In an open transactional model, customers are assumed to make requests on average every τ seconds, independent of when they receive responses for previous requests. This resembles a full push based model. Open workloads are identified by request inter-arrival time (τ) which is measured in seconds or arrival rate (λ) which is measured in requests per seconds. Often, these models assume that the arrival rate (λ) is a homogeneous Poisson process, and that

inter-arrival times (τ) are exponentially distributed with parameter λ (mean of $1/\lambda$). Using mean value analysis, one could derive a solution for general open queuing network models [57]. In case of open multi-class models where there are C customer classes, workload intensity is denoted by $\lambda \equiv (\lambda_1 \dots \lambda_C)$, where λ_c , is the class c arrival rate.

In a closed transactional model, clients are assumed to wait for responses to their requests. Upon receipt of this response, a client spends time analyzing (i.e., determining the next action to take) before issuing a further request. Closed models are defined by the number of users (N) and think time (Z). A closed multi-class model consists of C classes, each of which has a fixed population. We denote the workload intensity by $N \equiv (N_1 \dots N_C)$, where N_c , is the class c population size. Obtaining solutions for closed models is somewhat more complicated than for open models and usually involves iterative methods with some convergence criterion.

Solution to Flat Models

Separable flat queuing network models (as opposed to layered, tiered, etc) are usually a simplistic yet fine approximation of how system is performing at hardware layer. Devices can be mapped to queuing centers and requests can be mapped to customers. General mean value analysis (MVA) based solution for flat queuing network models can be obtained easily based on a set of laws governing these networks. For example, algorithm 1, adopted from [57], represents such solution for open model workloads on a flat queuing network.

Modeling Thread Pools

Real computing systems can be a lot more complex than to be represented using flat models. Flat separable networks models simplify computing systems by treating all layers of software and hardware stack as queuing (or delay) centers sequentially visited by a number

Algorithm 1: General solution for flat seperable queuing network models with open workload.

- input** : Arrival rates λ_c , and demands $D_{c,k}$ for all customer class c and each server k
output : Response times R_c , queue lengths $Q_{c,k}$, response times $R_{c,k}$ for each c and k
- 1 The throughput ¹ for each traffic class is equal to arrival rate of that class ²: $X_c(\lambda) = \lambda_c$ (c refers to a specific class).
 - 2 Utilization³ of each server for each class could be obtained from the arrival rate (λ_c) and service time ($D_{c,k}$) of that class c on the server k as follows: $U_{c,k}(\lambda) = \lambda_c D_{c,k}$.
 - 3 Residence time (at server and queue) for each customer class is obtained from

$$R_{c,k}(N) = \begin{cases} D_{c,k} & \text{for delay centers} \\ \frac{D_{c,k}}{1 - \sum_{j=1}^C U_{j,k}(\lambda)} & \text{for queueing centers} \end{cases}$$
 - 4 Average number of customers of class c in the queue of server k is then obtained from $Q_{c,k}(\lambda) = \lambda_c R_{c,k}(\lambda)$.
 - 5 The system total response time for class c customers is obtained from $R_c(\lambda) = \sum_{k=1}^K R_{c,k}(\lambda)$.
-

of customers. As an example a set of resource (e.g. processors, disks, and queues) whose access is governed by a thread pool (e.g. using web container) cannot be modeled as a queuing center or delay centers or combination of those. In such system, departure (processing) rate of requests μ depends on how far the thread pool is saturated. Only if thread pool is not saturated the departure rate will be equal to throughput of underlying hardware system and easily obtained by solving its flat model with open workload component λ (see 2.2.2). Moreover, the saturation level of thread pool depends on departure rate. To overcome this circularity, models associated with two layers should be iteratively solved until some convergence criterion is met [58].

Modeling Memory, Swapping, and Paging

Memory can be viewed as a temporary buffer to store code and data segments of active threads. Memory and its management affect the performance of processes running inside OS by controlling the extent to which processing resources (CPUs, disks, etc.) can be uti-

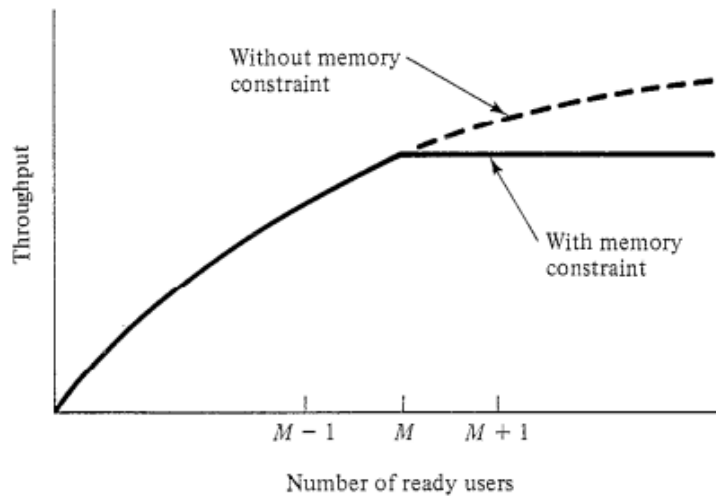


Figure 2.2: Effect of memory on throughput not considering swapping and paging.

The throughput-limiting effect of a memory constraint can be even more complicated in real computer systems because of complicated memory management mechanisms such as *swapping*. All current operating systems have implemented ways to oversubscribe the memory (i.e. remove the hard constraint on the number of active processes sharing memory) using a swapping mechanism. Swapping is about bringing the extra processes into memory and taking them back to a secondary larger storage as they become active. With swapping, one can highly increase the multiprogramming level with cost of frequent swapping between primary memory and secondary storage. The amount of swapping at each given time depends on the total number of active processes N , and the portion of these processes that can fit into the actual memory simultaneously M . If N is less than M then no swapping will occur, otherwise there will be swapping with some probability which depends on M and N (with growth of N compared to M the probability will be increased). As

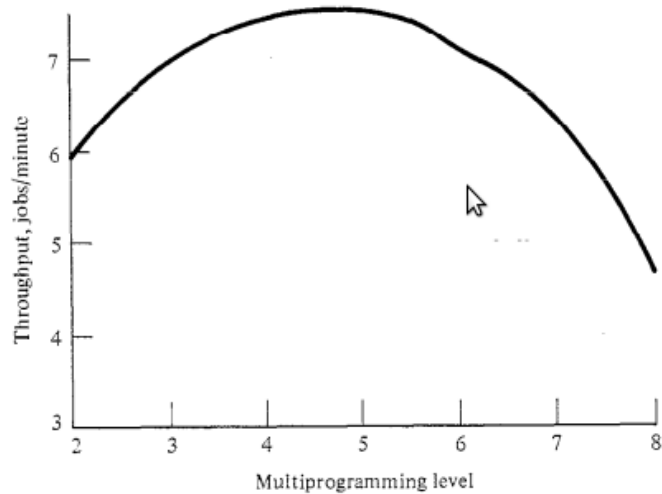


Figure 2.3: Effect of multiprogramming on throughput.

Another mechanism commonly used in current operating systems that makes it difficult to identify or predict their behavior is *paging*. In paging, physical memory to be divided into some number of fixed-size page frames, and portions of program quantified with these fixed size pages (as opposed to the whole code or data segments) are swapped into the memory. These portions are selected according to the concept of 'locality of reference': from a large address space owned by a program, only a small portion will be referenced during any short time interval. Paging lets OS to allocate a much larger virtual address space for the program than the actual physical primary memory available. This leads to 1) accommodation of programs whose virtual address spaces are larger than the amount of physical memory and 2) increasing the number of concurrently active programs. However, management of the virtual memory (i.e. moving pages between primary memory and disk

in response to page faults) consumes CPU and I/O resources.

The OS itself tries to optimize the performance of given programs for the specific amount of memory given by (i) controlling the amount of programs that are allowed to compete for memory resources (multiprogramming level), (ii) distributing proper number of page frames to each of these programs, (iii) choosing the pages that should occupy the page frames allocated to a program and (iv) deciding on the page that should be removed from primary memory for bringing a currently non-resident referenced page. These items form the page replacement policy of operating system, which in conjunction with memory reference characteristics of programs ⁴, and the amount of given memory will result in specific system performance.

As we move up the layers of software stack tracing the effects of different configurations on memory management becomes significantly more difficult. For example in container level, by modifying internal multiprogramming level of a container process (see subsection 2.2.2) its memory usage pattern (or program lifetime function) will change, resulting in different system's paging and swapping for its OS process.

Modeling Tiered Software with Synchronous Calls

Distributed applications usually contain one or more layers of software servers. Examples include three tier database driven applications; applications developed using Java's remote method invocation (RMI), Remote Procedure Call (RPC), or Enterprise Java Beans (EJB). In such systems Performance features (e.g. response time) are affected by the software design (e.g. message passing versus synchronous calls), the multi-threading level, number of instances of software processes, and the allocation of processes to processors. The stan-

⁴Reference characteristics of a program can be denoted by its *lifetime function*, which is the average number of milliseconds of CPU service that elapse between page faults for various numbers of allocated page frames.

standard model for representing these systems is Layered Queuing Model (LQM) that has been developed together with its analytical techniques such as Method of Layers (MOL) [59]. This analytical technique models the system tiers each with its own software and hardware layers and captures the contention delays at each layer or tier. Each layer is represented by a Queuing Network Model (QNMs) which can be solved by the mean value analysis (MVA). By iterating among the layers of QNMs, a fix point solution is found for the whole LQM [60].

These models help capturing different types of resource demands at different tiers. For example, in a web based tiered system front-end servers are more stressed on their IO and CPU capabilities, I/O for direct content serving to the users, and CPU because of the connection rate and number of concurrent connections. Application servers are more CPU-stressed because of support for business logic. Finally, the requests translated by application layer to database commands are executed in backend database servers that put a lot of demand on storage tier. Model inputs for LQMs include: (i) the structure of the model including the services and their interactions for representative scenarios, and a topology of the underlying middleware and hardware, and (ii) the same set of quantitative performance metrics as flat queuing models (e.g. workload component, and service times or demands (S_c) for each class of service c on each resource k). Figure 2.4 shows a typical Layered Queuing Model (LQM) of a web-based application. We assume that there are C classes of requests; the “User” block in the figure represents N_c users in class c at their browsers. Z_c is used to denote the mean think time of class c users. N_c and Z_c are the workload parameters for class c .

The *WebServer* block represents the server software with M threads, running on processor *WSProc* (this is indicated by the *host* relationship). The box labeled *webOp* represents the operation done for the users, and requires a mean CPU demand of $S_{w,c}$ for requests in

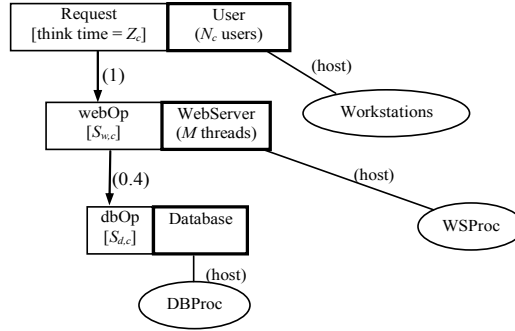


Figure 2.4: Multiclass Layered Queuing Model of a Web application.

class c and on average one database operations. The database is running on its own device *DBProc*. *dbOp* represents the mean CPU demands of $S_{d,c}$ at *DBProc* for a class c request.

Outputs of LQM of Figure 2.4 include mean response times to class c users $\langle R_1 \dots R_C \rangle$, mean response times to class c users at database server $\langle R_1^d \dots R_C^d \rangle$, throughput of each class c users $\langle X_1 \dots X_C \rangle$, utilization of web server processor U_w , and database processor U_d .

2.2.3 Discovering Hidden Parameters of Performance Model

The objective of an analyzer is to describe the system (or environment) in terms of a given parametric model. This identification process includes finding or estimating parameters of the model using the data obtained from the system (or environment). The parameters to be estimated are usually what performance models are built upon, the workload components (e.g. given by think time Z_c and number of users N_c) and resource demands (e.g. the CPU demands $S_{w,c}$ and $S_{d,c}$ in example). For layered models (i.e. with thread pools) size of thread pools and for tiered models number of calls per request are added to these hidden parameters. The data obtained from the system and used in estimation process is composed of the directly measurable performance data including response times and throughputs for classes of users and utilizations of resources (e.g. $\langle R_1 \dots R_C, U_w, U_b, X_1 \dots X_C \rangle$ in example of Figure 2.4).

There are different approaches to estimate hidden parameters of a given performance model using the measured data. As far as we have seen in the literature, there are two major techniques to discover these parameters. The first approach is to select some equations of the performance model and reshape them into a commonly known parametric form in which un-accessible variables are mapped to model parameters and the training dataset provides value for all inputs of outputs of the model (i.e. no missing data). In this case common supervised learning methods such as regression techniques can be used to estimate unknown parameters. For example [61] uses the queuing theory formula that relates utilization of each resource center k , U_k , to system's multi-class workload component $\langle \lambda_1, \dots, \lambda_C \rangle$ and resource centers' service demands $D_{c,k}$,⁵ $U_k = \sum_{c=1, \dots, C} \lambda_c D_{c,k}$, to form a multivariate linear regression problem with C independent variables: $Y = X\beta + \varepsilon$. Here, Y is a $T \times 1$ vector of resource utilization U_k samples over time T , X is $T \times C$ matrix of arrival rates $\langle \lambda_1, \dots, \lambda_C \rangle$ over time, and β is $C \times 1$ vector of resource demands for all classes of service $\langle D_{1,k}, \dots, D_{C,k} \rangle$. Other examples of CPU demand estimation using regression analysis are [62, 63, 64]. In many cases, especially when complex performance models are used, the possibility of finding a parametric form that conforms to a subset of formulas of performance model is low. To overcome this problem, some newer versions of the technique use regression splines instead of linear and polynomial regression functions [65]. In [66] and [67], multi-class queuing models were used to infer the per-class service times at different servers of a two-tier web cluster using throughput, utilization, and per-class response time measurements. They try to minimize the sum of predicted response time mean square errors using a non-linear optimization solvers and quadratic minimization programs. In addition, Maximum Likelihood Estimation or MLE (as opposed to LSE) together with queuing model have been used for the same purpose [68].

⁵Note that the service demand of a class of service on a processing center is the total time the processing center spends for each request (or client) of that class of service.

In the second approach the performance model is used as-is. Only the missing parameters are computed and tracked using a specific unsupervised statistical learning technique called Expectation Maximization (EM). For linear systems, the technique is well-known as Kalman filter. In this technique model contains three sets of variables inputs, outputs, and states. Observable portion of data is mapped to input or output variables while states are always hidden (i.e. no training data with known states). In EM one bootstraps parameter estimation by making an initial guess for parameter values and improve estimates iteratively by:

1. computing expected value of hidden variables (i.e. decoding) given model estimates and observed variables (i.e. maximizing probability observing the observed portion by navigating over possible hidden state sequences).
2. re-computing MLEs (i.e. likelihood maximization) of model parameters based on estimated hidden variables.

This process is repeated until some convergence criterion is met. This approach has been previously adopted in [69, 70, 71] using Kalman Filter and LQM with both open and closed workload types.

2.3 Planning and Feed-forward Predictive Control

In general the goal of any optimal control approach is to choose a sequence of feasible *control actions* that maximizes a defined *performance criterion* (or objective function)⁶. The analyzer components (described in the past section) provides an accurate model that can be used in calculating the cost and performance criterion by providing projection of

⁶One can instead say optimal control minimizes an expected total cost function

system's behavior and state in the future. A planner uses this comprehensive and accurate model to rapidly explore multiple decisions and find near-optimal solution during each step [53, 72] (see Figure 2.5). In context of control theory, this schema is called open-loop or feed-forward predictive control. In the next subsections we describe two common approach currently used in planning and optimization for systems performance.

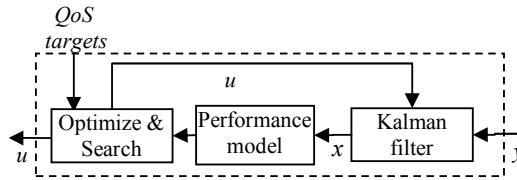


Figure 2.5: Structure of an optimization based planner.

2.3.1 Searching in continuous space

If one can describes the objectives of controller in terms of outputs of an identified system model (e.g. performance, QoS or profit), the proper control inputs that satisfy the objectives can be easily derived by searching the space formed by choosing actions and applying them to the model. In case the search can be converted into classical mathematical optimization problem, several approaches such as primal decomposition, interior point methods, simulation annealing, etc can be used to derive the proper control inputs. Usually optimizer takes initial control inputs, the number of iterations during optimization, the utility model, and a set of constraints as input and outputs optimal control values and maximum utility gained from those. This process is usually done iteratively. To skip falling into local optimums global optimization techniques can be utilized.

Although simple, the applicability of this form of search is reduced when the system model is discrete or time-varying. In the following subsection, planning for such models is explained.

2.3.2 Searching in discrete space

In many cases, an autonomic system designer deals with a discrete model that varies over time. The most commonly known tool that makes use of such model is a Limited lookahead controller (LLCs) [73, 74]. It explores a search space formed by different choices of control actions (while control inputs must be chosen from a finite set) over the predicted model [75] to find the optimal solution; thus, management problem is posed as a sequential optimization under uncertainty. Compared to steady-state optimization approach studied in subsection 2.3.1, LLCs use time-varying models that take into account the transient behavior of the system such as steady-state formulation of equation 2.4. However, control inputs $u(k)$ are chosen from a finite set of control options available $U(x)$ ⁷ at the corresponding system state x . Here systems' state space X is defined by operating constraints described by the inequality $H(x) \leq 0$.

The objective of control, which is encoded as transient cost or utility using norm-based function, implicitly defines desired state and preferable paths toward this state:

$$J(x(k), u(k), \Delta u(k)) = \alpha_1 \cdot \|x(k) - x^*\| + \alpha_2 \cdot \|u(k)\| + \alpha_3 \cdot \|\Delta u(k)\| \quad (2.8)$$

here $x(k)$ is the current state, $u(k)$ denotes the control inputs, $\Delta u(k)$ denotes the corresponding change in these inputs, and α_1 , α_2 , and α_3 are user-defined weights denoting the relative importance of the variables in the cost function.

According to feed-forward controller scheme, the controller explores a set of future states within the lookahead horizon, At every predicted state, environmental inputs to the controller must be predicted and different choices of control inputs should be navigated. Controller then selects a path that minimizes the cumulative cost while satisfying both state

⁷The finite set of all permissible control inputs can be denoted as U .

and input constraints within lookahead horizon $u^*(j)|j \in [k+1, k+N]$:

$$\begin{aligned}
 & \text{compute: } \min_U \sum_{j=k}^{k+N} J(x(j), u(j)) \\
 & \text{subject to: } H(f(x(j), u(j), \hat{w}(j))) \leq 0 \\
 & u(j) \in U(x(j))
 \end{aligned} \tag{2.9}$$

The first control input leading to this path is chosen as the next control action.

2.4 Execution through Regulator feedback loops

The simplest form of a goal for a management system is to regulate a system output around some value (let us call it a set point). A feedback loop for this purpose can be constructed by feeding back the control error (difference between a set-point and a measured output) as an input to the system (often with some intermediate processing).

For example, in [76, 77] controllers have been used to maintain the utilization of a VM's CPU at certain percentage (usually 60% to 80%) of the total allocated CPU share of the VM (i.e. headroom principal). In very simple cases (e.g. non-layered non-tiered systems), *response time* or *throughput* based performance guarantees required by applications, can also be described in terms of utilization control, and thus, be controlled by feedback loops [77, 78, 79, 80].

For a system whose dynamics are known (e.g. through system identification techniques), a proper feedback based regulator can be constructed at design time using the design techniques such as pole placement, root locus, or etc. Dynamics of a system can be represented in several mathematical forms such as transfer functions, state-space models, difference equations, etc; depending on the form used the controller might be designed

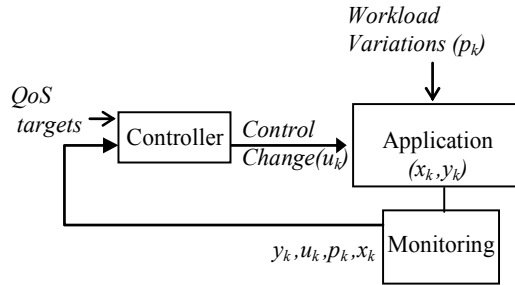


Figure 2.6: The general architecture of resource allocation controller.

using the same mathematical method (e.g. a transfer function). In several examples, classical linear continuous proportional-integral (PI) and proportional-integral-derivative (PID) controllers are applied to processor power management [81, 82], QoS adaptation in web servers [77], and load balancing [83, 84, 85].

The controller component can also include an estimator that dynamically adjusts the controller gain using the state estimates of the model. For example, Linear-Quadratic-Gaussian (LQG) control (see Figure 2.7) dynamically adjusts LQ-optimal gain based on a *state estimate* that is used to form a control signal. The estimator is usually a simple or extended Kalman filter [47] able to maintain a good estimate of model unknowns by calibrating itself to the measurements during runtime. As an example, [76] tries find the proper CPU share a_k for a VM that satisfies the headroom principal. In this case the controller tries to maximize the likelihood that $u_k = 60\%a_k + w_k$ (where u_k is CPU utilization and w_k is the random perturbation) by adjusting a_k based on the estimation error $e_k = u_k - 60\%a_k$.

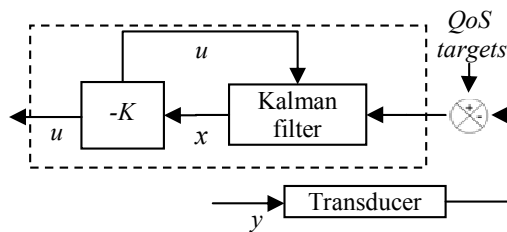


Figure 2.7: Structure of an LQG regulator.

2.5 Other ways of implementing managers

Although in this chapter we focused on a specific way to implement autonomic managers, there exists several other ways as well. The description and comparison of these approaches in terms of applicability to problems of different domains, ease of adoption, and ease of learning by humans although interesting, is out of the scope of this report. Here, we only briefly mention a commonly known approach known as policy based autonomic management.

2.5.1 Policy based autonomic management

Policies are a way of providing “formal behavioral guides” to systems about potential actions improving the system’s behavior [86]. Policy based management (PBM) is usually used in systems with several components while each component behaves according to the policies passed to it. It is used where there is a need for independence in components, either because consulting with central controller on every action is impossible or costly (assume a router that asks for destination of a packet on every arrival) or because components or are prepackaged with certain controllable behavior through predefined policies (i.e. Policy Enforcement Points or PEPs).

PBM covers a broad spectrum from security models (i.e. control access, confidentiality, an integrity) to network management (routing, traffic flow, etc), and computer systems configuration (server configuration). There are also several policy description languages (i.e. Imperial College’s Ponder, Bell Labs PDL⁸, IETF’s PCIM⁹, IBM’s PMAC¹⁰ and ACPL¹¹).

Policy can be specified different levels of abstraction. Goal policies, describe the con-

⁸Policy Description Language

⁹Policy Core Information Model

¹⁰Policy Management for Autonomic Computing

¹¹Autonomic Computing Policy Language

ditions to be attained without specifying how to attain them. In business driven policy management, goal policies might target the long term high level business objectives (e.g. minimizing cost and satisfying performance constraints [87]). Goal policies can be refined into operational or action policies for components [88, 89]. Action policies target real time reactive short term control of systems. A low level action policy is usually in the form of event-action rules [90, 91, 92]. For example, in an event of 'saturation of component', the action might be 'adding more capacity to X'. Event can be a violation of a constraint on certain properties of the configuration or value(s) of monitored metric(s) (e.g., CPU utilization of an individual VM instance is less than 20 for 10 minutes).¹² Note that an implicit target exists for the types of management rules (e.g. individual server). The action part of the rule can be however more complex, including a tactic determination procedure which chooses a set of actions to be executed. Further, an "action" might, involve the realization of a global management decision (e.g. adding or removing servers), a local management decision (e.g. increasing a process's memory), or emitting an event (e.g., an *alert*). See this scheme applied to autoscaling [93, 94]).

In conclusion, policy based techniques are a way to implement the same concept as feedback and feed-forward control, but using a more natural construct for humans to understand which results in ease of use and adoption. This, however, comes with the cost of losing some rigor. Optimality, a criteria usually targeted by control based approaches, cannot necessarily be achieved using policy sets of certain type. Thus, other techniques for policy set optimizations (at design time or runtime) have already been suggested [95, 96, 88]

¹²A constraint violation may be caused by multiple triggering problems. Thus a condition can be a conjunction or disjunction of other conditions

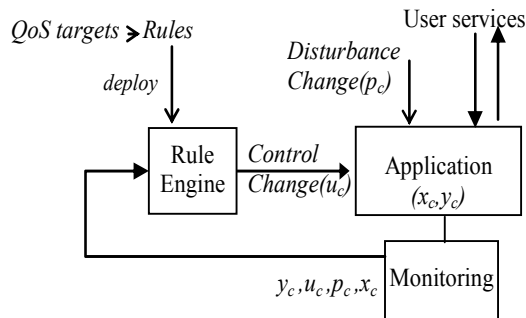


Figure 2.8: Architecture of a rule-based autoscaler.

2.6 Summary

Autonomic computing, a term introduced by IBM, indicates systems that are self-managing, self-tuning, self-healing, self-protecting, self-adapting, self-configuring, and self-organizing (briefly called self-* systems).

Autonomic computing discipline is relevant to building self-managing complex resource sharing systems that manage themselves in accordance with high-level objectives specified by humans.

A very common architecture to implement autonomic managers is the well-known Monitor-Analyze-Plan-Execute (MAPE) loop suggested by IBM. In this style, adaptation strategies and mechanisms are separated from the applications or systems. The architecture includes components such as a system monitor, analyzer, automated learner, forecaster, and planner that are used to decide about proper action(s) to be taken by execution subsystem based on the current system measures.

A monitoring subsystem of MAPE loop is responsible for measuring inputs, and outputs of the managed system, quantifying them, sometimes aggregating them, and keeping them as a history.

An analyzer subsystem of autonomic management loop targets identification of system under management. This identification enables the autonomic manager to project system's

behavior and state under different actions in the future. One way to gain this identification is through obtaining a model of the system. This model can take several forms such as symbolic representation, neural nets, fuzzy rules, or statistical models. A special case of statistical models used in computer systems design are performance models which let better understanding of system properties and internal (the saturation point of the system with respect to various workloads).

A planner subsystem of autonomic management loop uses the model provided by analyzer subsystem to rapidly explore multiple decisions and find near-optimal solution. The goal of planner is to choose a sequence of feasible control actions that maximizes a defined performance criterion (or objective function) or simply regulates the system towards an objective. Based on the system model given by analyzer the type of search performed by planner to find near optimal solution can be different. The classes of models we targeted here were discrete time-varying and continuous static.

An execution subsystem is responsible for enforcing management decisions on the system. This management decision might be regulation of a system attribute around some value. Feedback loops can be used to enforce such decision despite disturbances that might occur in system environment. In general, a feedback loop for this purpose can be constructed by feeding back the control error (difference between a set-point and a measured output) as an input to the system (often with some intermediate processing).

Although in this report we focused on a specific implementation of autonomic management, other choices in the domain exist. As the most notable example, policy based autonomic management offers somehow a less costly solution by only providing "formal behavioral guides" to systems about potential actions improving the system's behavior rather than searching for optimal actions.

Chapter 3

Cloud Computing: Initial Steps Towards ULSRSS

Probably currently most outstanding example of ULS systems and System of Systems in IT domain after Internet is Cloud computing. According to [97] cloud is “a large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.” Like any other large-scale distributed system, Cloud contains a large number of computing nodes. This massively distributed system provides access to a resource and/or a set of service mostly related to computing power, storage, and network, leveraging existing technologies such as virtualization, multi-tenancy, service-orientation and grid computing. Users are multiplexed using multi-tenancy techniques such as virtualization, OS processes, container level thread pools, or application level techniques. Upon joining the Cloud, users have access to a large amount of resources and services that can scale based on their demand (i.e. scalability). Resource oriented Clouds are referred to as Infrastructure-as-a-Service

(IaaS) while software oriented ones are either Software-as-a-Service (SaaS) or Platform-as-a-Service (PaaS). User can use Cloud resources and services in a dynamically changing fashion and the environment might transparently manage users' resources (i.e. Elasticity). Users' resources are logically virtualized; that is users have their own view of resources and are not concerned about multiplexing detail. With pay-per-usage model, users only pay for the resources they need without planning on buying, maintenance, or upgrading of physical resources and infrastructure in the Cloud.

From socio-technical point-of-view, Cloud can be a tool to sharing common information. Numerous amounts of data that is generated everyday can be stored, streamed, processed, and analyzed in the Cloud using the storage, processing, and network services. Example of this data is weather data provided by largely deployed sensors, road traffic information obtained using distributed sets of cameras, user activities in a social network generated through their smart phones, etc. As Cloud gets more popular, users would use it to simultaneously collaborate and work on these common data sets. Since the Cloud nodes are scattered through the globe, mobility and the ability to access this information from various locations with low cost is a natural consequence.

In the following subsection, we enumerate different types of cloud offerings, the technological challenges in building them, and their typical usage pricing models.

3.1 Building Blocks of Current Clouds

Currently Clouds are formed by interconnected set of datacenters. A data center environment is composed of communication system, servers, and storage subsystem. These computing resources are hosted in controlled environments and under centralized management. Additional set of services provided by datacenters are (i) Infrastructure services such as

routing, switching (ii) Application services such as load balancing and caching (iii) Storage services such as SAN architecture, fiber channel switching, and back up. In the following subsections, we enumerate each of these items.

3.1.1 Network

In terms of network architecture, in a generic enterprise data center is composed of three layers: (i) *edge routers* provide connectivity to Internet, (ii) *core* routers connect internal network to the Internet connectivity layer, (iii) *aggregation* (or *distribution*) layer is composed of multilayer aggregation switches, performing aggregation, and connectivity for internal Layer 2 switches to the *core*, (iv) access layer provides Layer 2 connectivity and features to the server farm by connecting them to aggregation layer.

Performance of network (i.e. in terms of latency) depends on a number of factors. At hardware layer, the most dominant factor the presence of congestion when switch buffers are saturated. Software protocol related considerations could also affect performance by changing the ways hardware is utilized. One protocol might allow better utilization of redundant links between layers of network leading into more parallelism and better performance. For example, bypassing Ethernet's Spanning Tree Protocol (STP) limitations in terms of aggregate bandwidth (i.e. near root congestion in distribution and core layers of datacenter) is being made possible by adopting layer 3 protocols, layer 2 multi-path protocols (for example L2MP/MIM¹ and L2MP/TRILL²), or active-active switch connectivity.

By controlling a portion of network links and bandwidth associated with each activity or customer (i.e. by logically partitioning the network into a set of virtual networks or VLANs and associate each to an activity) one can have fine grained control over performance.

¹Layer 2 Multi-Path/Mac-in-Mac

²Layer 2 Multi-Path/Transparent Interconnection of Lots of Links

3.1.2 Storage

Storage Area network (SAN) is made up of storage devices such as disk and tape sub-systems and fiber channel switches that provide connectivity between servers and storage devices (through fast block-level access). SAN environments commonly use Fiber Channel (FC) suite of standards to connect servers to the storage devices and to transmit small computer system interface (SCSI) commands. However fiber channels can be substituted with cheaper alternatives such as Ethernet switches supporting FCoE (Fiber Channel over Ethernet), or routers that support iSCSI (SCSI over IP) or FCIP (fiber channel over IP). Note that SAN is different than Network attached storage technology (NAS) since the latter provides file-level access instead of block level and it uses higher level Network File System (NFS) or Server Message Block (SMB) instead of SCSI commands.

3.1.3 Server farm

Server farm is composed of servers that are placed at leaves of the network and are connected through access layer switches that can be configured at Layer 2 or 3 (bridging and routing). Each server usually has multiple multi-core CPUs, some amount of RAM, multiple Ethernet cards, to connect to access switches, a fiber channel(FC) Host Based Adaptor (HBA) to connect to FC switches, and multiple hard drives.

Servers in data center are placed into racks and racks are placed with distance to provide enough space for cooling facility. Conventionally servers were placed in racks horizontally while each rack could contain up to 20 Servers. With the new generation of Blade servers, each rack is divided to multiple enclosures. Enclosure provides power, cooling, and interconnectivity to a set of server blades or switch blades. Each enclosure contains up to 16 vertically placed Blades minimizing required physical space, number of cables between the

blade servers and the access switches and energy consumption.

3.2 Use cases of Clouds

No matter what the type cloud offering and the server multiplexing technique used in the cloud, software is what users run on it. Numerous instances of software might be run in a cloud to support tasks performed or services offered by cloud users.

The lowest layer of deployable software typically deployed on IaaS Clouds is operating system (i.e. OS). In IaaS, each user runs a separate operating system instance on top of encapsulated units of virtualized hardware provided through a hypervisor. User processes are hosted in operating systems to perform different tasks or provide services on behalf of the cloud user. Performance in this type of deployment is usually measured by utilization of several virtualized hardware resources (e.g. CPU, disk, memory, and NIC)

3.2.1 E-Commerce Websites

A very well-known use of cloud can be to run web based services. These services can be run on top of suitable PaaS clouds with proper programming language, or by building the software stack composed of proper web container on top of OS deployed on a IaaS cloud. A web application is usually composed of two or three tiers: (i) front-end servers (e.g. Apache HTTP server) serve static pages locally and establish SSL connections without introducing lot of latency. (ii) load balancers (e.g. HAProxy) provide high reliability by regularly checking the health of application servers (next tier) and distributing traffic evenly among healthy ones. (iii) application servers (e.g. Tomcat server) handle request for dynamic content usually by communicating with a backend database. (iv) A redundant database such as MySQL (e.g. with Master/Slave architecture) that covers fail-over sce-

narios and possibly uses a regular backup system. In a PaaS cloud all of these items are combined and encapsulated as platforms; A PaaS provider maintains a pool of containers for each tier (e.g. web containers, application container, database container) which execute the programs written in the language of that container (e.g. Java servlet, Enterprise Java Beans (EJB), and SQL). Users are then multiplexed using container specific multi-tenancy techniques. In a IaaS Cloud, these containers are deployed by user themselves as group of processes within obtained virtual machines. The strategy that containers take for using OS capabilities (i.e. single or multi-process implementation) is different. For example, in realm of web containers, Mongrel [98]³ is deployed as multiple single-threaded OS processes; WEBrick [99]³ is implemented as a single process multi-threaded server that launches a thread per request; Thin [100]³ and Tomcat [101]⁴ are implemented as single process and use a fixed size pool of threads (this lowers context switching and lock contention); and Apache [102]⁵ implements a hybrid multi-process multi-threaded server with detailed control on the number of threads deployed by each child process (using `ThreadsPerChild` parameter), the maximum total number of threads that may be launched (using `MaxClients` parameter), hard limit on the number of active child processes (`ServerLimit` parameter) and number of server threads (`ThreadLimit` parameter).

3.2.2 Data Intensive Processing with Clusters

High Performance Computing (HPC) is one of the highly popular areas is computer science that is tightly coupled with data center environment. After creation of world wide web, and explosion of information, there was a demand in market to analyze the produced information and use it towards business needs. New generation of large scale data process-

³A web container for ruby on rails framework.

⁴A Java servlet container.

⁵A web server for static content with extensions to connect to various containers.

ing frameworks, such as Hadoop⁶, that are able to utilize thousands of computing nodes in processing massive amount of data have emerged.

Datacenters are enabling technology for performing large scale data processing. They can lend lots of processing power to computationally intensive jobs. High available bandwidth in a datacenter makes it easy to transfer large amount of data between nodes. The data is usually persisted in a distributed fashion amongst nodes local disks. Jobs being given to clusters are decomposed into a set of tasks on chunks of the persisted data. Since task distribution and data distribution do not map perfectly nodes start requesting data from one another. In this situation, high replication increases the chance of block's existence on one of the nodes of the same rack, and reduces the demand on intra-rack switches with off-rack reads. Conversely, with low replication factors writes will have better performance since they traverse less network switches. In an extreme case, the write is performed only locally on the same node with no network usage.

3.3 Infrastructure as a Service (IaaS)

IaaS multiplexes the hardware layer and offers computing services such as storage, CPU and memory to users. The multiplexing technology employed in IaaS layer is hardware virtualization.

Original purpose of virtualization was better utilization of resources by increasing multi-tenancy, lowering energy consumption, and reduction of cost through consolidation of virtual entities on hardware. The new wave of virtualization techniques which targeted commodity hardware started by development of Stanford's VMware[104] in 90s. This movement was followed by development of well-known open source virtualization frameworks,

⁶The Apache Hadoop software library is open source implementation of Google's Map-Reduce computation framework [103] that allows for distributed processing of large data sets across clusters of computers.

API	Description
Instance management	Run and terminate instances
IP address management	Allocate, associate, and disassociate addresses
block level storage management	Create, attach, detach and delete volumes. Create and delete snapshots.
VPN and VPC management	Create and delete Virtual Private Cloud (VPC). Create and delete subnet. Create, attach, detach, and delete VPN gateway. Create or delete VPN connection.
OS image management	Create, bundle, migrate, upload image.
File level storage management	Make, remove, modify access control list of a bucket. Put file into or get, copy, move, and delete files from a bucket. List objects or buckets.
Content Distribution Network (CDN) management	List, create, delete a distribution point.

Table 3.1: List of management APIs offered by Amazon Elastic Computing Cloud.

Xen [105] in 2000 and KVM [106] in 2007. In IaaS the CPU, memory and storage are packaged as virtual machines of different sizes sharing the hardware. A public and simple remote interface is provided by IaaS to users for managing resources, and, users provide, deploy and run virtual machine images (VMIs) without the cloud administrator help. The cost model for users is pay-as-you-go, and prices depend on leasing model (described later) and leasing duration.

The responsibility of cloud manager at IaaS layer is to maintain the agreed resource entitlements for all VM instances running in the cloud. Some current samples of IaaS providers include Amazon EC2 (Elastic computing Cloud) [107], ElasticHosts [108], GoGrid [109], FlexiScale [110], and RackSpace [111].

Services offered by IaaS layer can be broken down into three broad category of computing, storage and network as-a-service (see Table 3.1) which are described individually in the following subsections 3.4, 3.5 and 3.6.

3.3.1 Management of Virtual Infrastructures

The major concern in providing IaaS is management of pool of distributed resources. Current publicly available *virtual infrastructure managers*, such as Eucalyptus [112], OpenNebula [113], OpenStack [114], address this concern differently. Usually the management logic at IaaS layers boils down to a set of decisions. Most important decision is *Resource selection*. Resource selection at IaaS refers to finding a feasible placement plan for VMs. Usually for each request physical machines are ranked based on some criteria associated with the requested VM and physical machines themselves ; then VMs are instantiated to the suitable PMs. In current implementation of IaaS layer (e.g. Eucalyptus[1]) resource selection decisions cascade down the cloud hierarchy, namely datacenters, racks, and nodes; the cloud front-end will choose the datacenter or rack which handles the request, and the rack manager chooses the proper physical machine.

After resource selection is performed, the physical machine selected to launch the VM (i.e. hypervisor) retrieves the instance image from the image repository through a file transfer protocol⁷; allocates the memory (i.e. requested by user or mentioned in the instance type), boots the instance and starts scheduling the instance for CPU slots. During boot process a user defined context data given at the instance request time is made available to the instance. This data is used by instance to configure itself as an element in a topology (e.g. a cluster).

A VM during its lifetime might be *migrated* to a more suitable resource [115]. This migration can be due to performance concerns (i.e. congestion of disk, network, or CPU at the physical machine hosting instance), or power consumption reduction (i.e. evacuating

⁷Instead of transferring the image, the image can be remotely accessed from SAN storage, and progressively obtained and cached as needed. This will significantly reduce the instantiation time by eliminating the copy time. However, the performance of SAN can affect subsequent performance of such instance during its lifetime.

underutilized PMs and shutting their components down).

Once user requests *termination of certain instance*, the manager will signal the associated physical machine's hypervisor to terminate the instance. The hypervisor will shut down the instance and cleans up the resources such as swap space, memory, network connections and/or copy of the virtual image⁸.

Resource selection criterion

Upon receiving user request for a virtual machine (which contains parameters such as virtual image, memory size, CPU speed, and required bandwidth), an allocation algorithm has to determine which physical host (hypervisor) is best for hosting the instance. The minimum requirement for the hypervisor is to: (i) have enough disk space for the image, and swap space, (ii) have more memory than the amount user requested, and (iii) practically has low enough CPU utilization that can guarantee the user will get his requested VCPU speed (in term of instruction per second).

On top of basic criterion for placement, there are several strategies that target an optimal placement. Optimal solutions always depend on the context of allocation. Some approaches assume allocated capacity is chosen from a discrete set of quanta with CPU, memory, network bandwidth, and storage encapsulated, while others assume free independent amount of resource can be requested along each of capacity dimensions. The way to achieve optimality also changes with objectives that should be met; increasing performance, reducing power consumption, consolidation are examples of different aspects objectives can be defined upon.

One strategy in placement, is balancing placement of VMs among datacenters, racks of

⁸Another optimization here is to figure out if to keep the copy of the image for future use. This leads to the fact that sending requests, for instances of the same image to the same hyper will eliminate image copy time or the continuous image serialization latency.

each datacenter, and physical machines of each rack. This results in lightly loaded physical machines and better 'per VM performance' in case of workload spikes. As an example, in a heuristic implementation, the criteria for handling incoming requests can be the PM's current utilization level and the average over data center or rack. If the PM's utilization is less than that of the calculated average plus some threshold, it will participate in handling the incoming requests.

Another aspect is providing high-availability and avoiding failures. In these scheme instances are placed in independent sections of a datacenter with different electricity distribution network called *availability zones*. Availability zones are used to add redundancy and fault tolerance to a given datacenter. For example Amazon Cloud currently has five datacenter regions, namely Northern, Virginia, Northern California, Ireland, Singapore and Tokyo. Each of these datacenters is further broken data into zones (e.g. us-east-1a, etc).

Another strategy is reducing energy consumption through minimizing the number of active hardware components in hosting environment [116, 117]. One way to achieve this is consolidating existing VMs into a fewer number of servers in a datacenter. Reducing active component does not necessarily imply hibernation of the physical servers. This could be done by turning off a CPU core, or just switching to lower CPU frequency. Power and migration cost aware application placement where optimization is performed to exploit cost performance tradeoff is targeted by [118]. Efficient allocation through resource overbooking for profiled applications in shared hosting platforms is discussed in [119]. Allocation considering higher level SLAs violations is discussed in [120]. Polynomial Time Approximation Scheme (PTAS) solution for Knapsack optimization problem [121], which is closely related, is discussed in [122]; although it does not consider the cost of VM migration.

3.4 Storage as a Service

Storage in IaaS clouds is offered to users at block level or file level. Block level access to a portion of physical machine's disk space offered to VMs through hypervisor. Decoupled network based volumes are also provided to VMs through multi-tenant block I/O protocols such as iSCSI SAN. VMs are capable of reaching multiple volumes (LUNs) from multiple storage arrays. The storage traffic generated can be bridged over conventional Ethernet (to lower the complexity and cost using legacy Ethernet switches and NICs) or be handled by fiber.

Another type of storage is, file level persistent storage. This service can be used to store common datasets, virtual machines images, etc. It can also be shared among multiple cloud users or web users. Pricing for persistent storage is usually per unit of storage and time interval up to a ceiling amount in combination with a price to be paid per number of IO operations.

As a part of storage, Content Delivery Network (CDN) can be offered as a way to distribute the data to physically distinct datacenters around the world. This helps users retrieve the files quickly from the location near them.

3.5 Computing as a Service

Computing is offered to users through delivery of virtual machines (instances) each containing a piece of CPU, RAM, local hard disk, network, and sometimes GPUs. In multi-core architectures, each VM is given shares of some processors or cores. Each share is identified by number of instructions per second and quantified with, million instructions per second (MIPS) unit. Note that in a multiprocessor VM the total service rate of n processors can be significantly less than n times a single CPU case. To realize resource entitlements to

individual VMs the hypervisor's CPU scheduler implements weighted fair sharing of the CPU capacity. It divides resource access time into fixed-length intervals and allocating each VM a certain share of the time in each interval. Notice that unlike memory, CPUs could be oversubscribed.

The interface for using computing cloud provided by the provider includes launching, stopping or terminating instances (see Table 3.1's "instance management" section). The price of an instance depends on its type and the amount of resource it has.

The pricing scheme also depends on capacity allocation approach and leasing model. Capacity can be allocated from a discrete set of encapsulated quanta of CPU, memory, network bandwidth, and storage. It can also be allocated based on independent amount of resources supplied by users. The leasing model can be generally divided into three categories:

in-advance reservation where resources must be available at specified time,

best effort reservation where request are queued and serviced accordingly,

immediate reservation where resources are processed right away or rejected, or

auction based reservation where customers bid for some number of a particular configuration and as soon as dynamically adjusted resource price lowers the bid amount the resources are allocated.

In case of immediate leasing with discrete allocation set, IaaS provider offers a set of possible *VMI configurations* each associated with a set of $\langle \text{time interval}, \text{price} \rangle$ tuples. Launching a VMI for this configuration is billed accordingly at the specified price rate. As an example in Amazon EC2, each hour of small configuration (i.e. $\langle 1 \text{ hour}, \text{small} \rangle$) costs \$0.085. It is assumed that a client may request any number of VMIs for any type at

any given moment. An adjustment may be done to adjust the cost of instance leases in response to various factors such as (i) the number of instances of a particular type purchased at one time (ii) the total number of VMIs purchased (iii) the number VMIs of a particular type purchased.

In in-advance reserved instances, the client has some precedence over immediate leasing clients. Usually an up-front fixed price charge is required to initiate a reservation and discounted rate is charged for the instances throughout the duration of the reservation. For example, Amazon requires an up-front payment for its reservation services and then charges a discounted hourly rate for instances during the reservation period. Associated with any offered reserved configuration there is a fixed price (i.e., down-payment) to be paid over the time interval and a set of $\langle \text{price}, \text{time interval} \rangle$ tuples. The client may select any valid pairing of this set and is billed accordingly.

Finally, in auction based reservation a bid for some number of a particular configuration of instance configuration is offered. For example, n instances are requested for the billing time interval t at a ceiling price per instance per time interval of p . If the provider can fully fill the request then it is met. Existence of partially filled bids is a matter of policy. Moreover, the technique used for dynamic pricing and the frequency of changing the price can be different.

3.6 Network as a Service

Network is also offered as a service to cloud users through virtual machines or platforms. Instances can use the networking fabric (i.e. NIC) to communicate to other instances or to internet based services. Layer 2 switches, and layer 3 routers are configured upon creation and termination of instances. At layer 2, Virtual private clouds (VPCs) or *virtual applica-*

tion networks(VAN)s can be deployed as a mean of providing more security . The offered API usually includes creating a subnet on top of VLANs, and creating, attaching, detaching a VPN gateway, and creating VPN connection ⁹. Some Cloud providers (e.g. Amazon) provide elastic reusable IPs that can be leased by instances. This can help in failover scenarios to keep the service functional by seamlessly replacing an instance by another and taking its IP address.

In a IaaS cloud, hypervisors create virtual NICs for VMs guest OSs each with its own MAC (and IP) address. Hypervisor can handle incoming traffic from VMs differently by acting as a layer 2 bridge, a layer 3 router, or a TCP based NAT. If Ethernet level bridging is used, no IP addresses will be associated with the bridge; it just distributes packets like switch and all VMs appear on the network as individual hosts. At layer 2, the IaaS manager, maintains a pool of MAC addresses. These MAC addresses are leased and associated with each VM using each NIC.

Virtual layer 2 networking for added security and better utilization becomes possible through Virtual LANs (i.e. VLAN) with help of VLAN enabled switching hardware. VLANs provide isolation of L2 network segments as routers do for L3. This combined with a VLAN enabled hypervisor such as Xen ^{citePBarhaEtAlSOSP2003} can enable VMs placed on different physical hosts to form a virtual network and have private communication. VLANs provide a tool towards programmable network configuration.

Using Layer 2 networking and hardware based VLAN is however not a feasible option for large scale Cloud providers. This is because (i) due to lack of hierarchical addressing (L2 addresses are flat and simplified) scheme in layer 2 switching information cannot be aggregated, (ii) efficient network utilization through multi-path is hard because STPs are formed to avoid loops, and (iii) shortest-path selection is not possible per packet is not

⁹Note that VPN is at IP layer (layer 3) while VLAN is at media access layer (layer 2)

possible. This makes Layer 2 networking hard to scale and puts a limit on size of virtualized data centers with lots of customers; a practical number of servers working in Layer 2 in datacenters are 10,000-. If the network is IP based top-to-bottom (as opposed to partially layer 2 MAC based), features such as multi-path network usage, network load balancing, and scalability come as natural consequences. As a result, so far major cloud providers like EC2 (EC2 has 60,000+ servers at the time of writing this report) only offer layer 3 (L3) oriented networks. This however comes with the cost of losing layer 2 capabilities such as multicast that might be mandatory for some cloud users with legacy applications.

To counter these issues, new wave of software based network virtualization has emerged recently to allow customers of datacenter to use low level capabilities offered by layer 2 network (i.e. designing subnets, VLANs, multi-cast, broadcast, etc) without losing scalability. The main concept behind software based network virtualization is an abstraction layer that emulates L2 for customers on top of provided L3 (also called "L2 over L3"). Current protocols and tools target this area include Generic Routing Encapsulation (GRE), Virtual Distributed Ethernet (VDE) [123], L2 tunneling over UDP, OpenFlow [124], Open vSwitch [125] and ESX server virtual switches [126]. Current open source IaaS implementations (e.g. Eucalyptos [127] and OpenNebula [113]) and cloud providers (e.g. Amazon EC2 Virtual Private Cloud [128]) have started using this type of virtualization.

3.6.1 Bandwidth pricing

There are at least two ways to charge for bandwidth into and out of the infrastructure. The first is a pay for use model in which the client is billed some dollar value per amount of bandwidth used. The second is a reservation based approach similar to with VM instances in which an upfront cost is incurred to ensure lower costs per unit bandwidth used.

An ordered set of ingress bandwidth rates, specifies different price per unit bandwidth

up to some *ceiling* amount. The client is billed according to the initial rate up to ceiling amount ingress bandwidth then according to next rate up to the next ceiling according to next rate, etc. Similarly, a second ordered set representing the egress bandwidth rates is also defined and utilized in an identical manner for egress bandwidth belonging to the client. The client is billed according to these elements for both data transmission into and out of the infrastructure.

3.7 Platform as a Service (PaaS)

A *PaaS provider* is an enterprise that is responsible for leasing application environments or *programmable services* to customers for various durations of time.

Costs incurred to PaaS provider comes from software and hardware resources. Software costs are usually quantified in terms of set of licenses purchased to support platform instances. hardware cost for PaaS provider comes from two places: (i) hardware consumed for running servers (e.g., LDAP servers, databases, etc.) that handles global system services (e.g. bookkeeping and billing customers, etc). (ii) Hardware resources consumed to run servers on behalf of platform instances. These hardware resources might be non-virtualized physical hardware owned by PaaS provider, privately managed IaaS using owned hardware, or even purchased quantized hardware from an external IaaS provider.

Customers of PaaS could be (among all others) SaaS providers who deliver software as a service by deploying their own programs on purchased platforms from PaaS provider. Usually platforms delivered to customers, exhibit a form of *elasticity* through autonomic resource provisioning. Meaning that, PaaS provider is responsible for modifying, adding and removing certain component elements on-demand to and from the platforms as necessary. How and when resources (e.g., application server instances at the PaaS layer) are

added to and/or removed from a platform is the subject to *elasticity policy*.

3.7.1 PaaS Provider Economic Model

The amount of resource for any platform service may be increased or decreased at runtime in response to requests by the client or occurrence of environmental changes. However, system services are global and not dedicated to specific application environment, and thus individual customers do not negotiate their quality in SLA or do not request a change in underlying structure of the service. The pricing of platform services would take two forms:

Charging for service usage: When each platform service is managed by PaaS provider, each configuration is defined by a set of per service-usage prices over different time-intervals and QoS targets. In this case, PaaS provider decides the necessary hardware to satisfy offered QoS. The relation between QoS offered by platform services and necessary hardware in multiplexed environment is investigated in subsection . An example of this pricing scheme is Google App Engine[129].

Charging for underlying resource: PaaS provider can even delegate the specification of underlying hardware or IaaS instances in support of platform services to the customer (i.e. the client simply makes a request for the addition/removing of n platform instances of type i). In this case, quality of platform services will not be guaranteed by provider and the price is defined based on underlying IaaS instances purchased. In extreme case customer himself pays the IaaS provider and PaaS provider only charges for management and system services; although PaaS provider might still let users to specify complex management rules to control underlying hardware/IaaS for each service, for example to define thresholds where in response to crossings, actions are taken by the PaaS provider (i.e., adding, removing platform instances). In this

case, SLA at the PaaS Layer would need to contain price for the topology per time interval, and the price for additional platform instances (added according to lease rate). An example of platform provided through this pricing scheme is Amazon Elastic MapReduce where pricing is in addition to normal Amazon EC2 and Amazon S3 pricing.

3.7.2 Multiplexing and consolidation: managing multiple services

Optimization at PaaS layer is possible for the provider when the prices for platform services are defined in terms of platform usage (as opposed to hardware usage) with specific performance guarantees, and the amount of multiplexing is decided by the provider based on the resource cost and QoS targets. A global multi-platform optimization problem involves the allocation of hardware resources to platform services (see Figure 3.1), such that cost to the provider is minimized resource sharing is maximized while attempting to meet all client application requirements as specified in the Service Level Agreements (SLA)s¹⁰ [87, 130, 131, 132]. The decisions made by the provider about resource allocation to platform services will directly influence both the performance of platforms and the provider’s cost of operations.

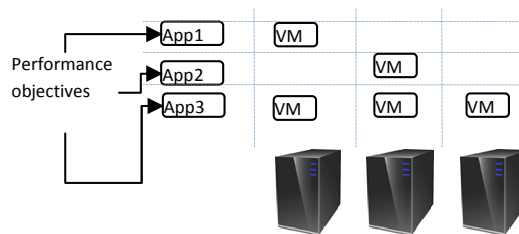


Figure 3.1: The interaction of layers in our optimization mechanism.

¹⁰It can be quite complex and comprehensive (e.g., considering aspects of both functional and non-functional requirements); however, in this work, only performance objectives that can be extracted from an SLA are considered. No attempt is made to fully model or develop an SLA or an SLA management framework.

Different approaches formulate the introduced optimization problem in different forms. One approach attempts to minimize cost subject to performance constraints [133, 134]. In this approach, the SLA represents constraints rather than flexible goals. Constraints could be based on response time or throughput. For example [133] finds the minimum cost deployment subject to processing capacity and user throughput constraints. It seeks deployments which minimize the overall cost of the hosts used, subject to meeting average delay and throughput constraints for each application as posed by its SLA. It is also possible for one to try optimizing a combined QoS measure subject to cost constraints.

A second approach attempts to simultaneously minimize cost while maximizing QoS attributes, through multi-objective optimization or MOO [135]. For example, Pareto-optimal solutions can find a good trade-off between conflicting performance and cost-saving goals rather than finding a single global optimum [136]. Geometrically, these well-balanced solutions concentrate around the “knee” point of the mutual objectives curve.

A third approach is to optimize a utility function combining application-level SLAs and resource costs with tunable parameters for the administrator to specify trade-offs between the two [135]. In this approach, a system-level global utility is defined in terms of local utilities that are in turn based on the achieved service level of the applications. Figure 3.2 represents a sample service level utility function (or local utility function) for a platform, where the vertical line indicates the SLA target of a platform in terms of its delivered response time. Note that utility decreases as the value of service level approaches the SLA limit. These local utilities are combined with a set of coefficients that allows for the high level control of performance goal fulfillment and the resource cost savings. A global utility function U_0 is expressed as the difference of the sum of platform-provided resource-level

utility functions and an operating cost function as follows:

$$U_0 = \left(\sum_{j \in App} u_j(A_j) \right) - \omega.cost(A) \quad (3.1)$$

where ω denotes an adjustable weight (working as tunable parameter for the administrator), u_j maps the platform j 's resource allocations (i.e. A_j) to the local utility function for platform j . U_0 can be associated with the profit of the cloud derived by subtracting the revenue and the cost respectively. PaaS provider objective is to maximize U_0 subject to a set of capacity constraints which come from the physical layer of the private cloud as follows:

$$\begin{aligned} &\text{maximize: } U_0 \\ &\text{subject to: } U_1, \dots, U_n \end{aligned} \quad (3.2)$$

It is assumed that each allocation signal a_{ij} is constrained to lie in the interval $[0, c_i]$ meaning that a platform can get the whole capacity of a PM. Notice that the problem has a best effort nature and we treat a service level objective (i.e. target on a specific QoS metric) as a soft constraint by incorporating it into the objective function. One can use a mathematical optimizer to solve the introduced optimization problem defined in equation 3.2. Given the utility model u_j for each platform the optimization algorithm should output the optimal allocation vector, and maximum utility gained from that allocation.

3.8 Software as a Service (SaaS)

SaaS type of Cloud provides instances of software and applications that are typically installed in businesses' computer networks or personal computers. Examples of this software include customer relationship management (CRM), accounting, invoicing, human resource

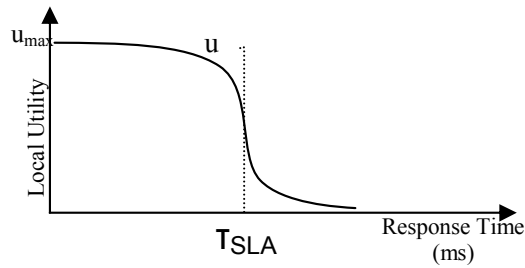


Figure 3.2: Smooth service level utility function; vertical line indicates the service level objective of a platform (as defined in SLA).

management (HRM), and content management (CM)[137].

while hosting of business application on behalf of users dates back to 1960s with IBM mainframes providing banks with isolated financial service and 90s with Application Service Providers (ASP) providing centralized management of particular business applications, current SaaS deployment has a number of distinct features: (i) software offered currently as a service are usually web-based rather than client-server thus users can use common web browsers instead of installing custom clients. (ii) Instead of running separate instance of the application for each business, SaaS solutions normally utilize a multi-tenant architecture, where applications are designed with built-in multi-tenancy (i.e. multiple businesses, users, and data partitions). (iii) inspired by web 2.0 and on-line social networks some SaaS based applications allow for collaboration and information among users of multiple organizations.

The second item means that businesses share more portion of software and hardware stack (i.e. a single version of the application with a single configuration of application server, database container, operating system, network, and hardware). This usually results in a (i) unified and hardened stack of software (this includes management software) and configuration which is maintained by the SaaS provider, (ii) short learning curve for community of users (here businesses)¹¹ and (iii) low cost for user provisioning for SaaS

¹¹Note that the details of multi-tenancy is still transparent to SaaS users.

provider. The down side of this scheme is that applications are not customizable for a single customer through change of source code, database schema or GUI. Thus, customers can only apply different configuration parameters already designed in the software. To support scalability in this environment, the applications are scaled horizontally (installed on multiple machines). The most well-known example of a SaaS provider using this architecture is Salesforce.com[138].

The pricing of joining a SaaS provider and using its general services (using software systems) is usually per subscription (i.e. monthly fee or an annual fee). Applications however are usually priced based on some usage parameters, such as the number of users ("seats") using the application, number of transactions, etc. Note that applications might belong to someone other than SaaS provider. This is clearly different from a perpetual software license with an up-front fee. In a long run users have to compare the tradeoff between the initial setup cost and licensing fee of a conventional software with per-usage cost of a SaaS.

While in the simplest case, a SaaS offering will involve only one class of service, there will typically be multiple classes of service to consider, each with different types of guarantees and assurances offered for various prices on potentially multiple timescales. A SaaS provider will offer a set of configurations (also referred to as classes of service). Each configuration/class of service is associated with a set of $\langle \text{price}, \text{time interval} \rangle$ tuples, representing the interval and the associated billed interval. The client may select any subscription from the possible options.

Information that an SLA at the SaaS layer would need to contain the follows: a configuration of the subscription, the price, the time interval over which the price should be paid, the start time to the millisecond of the subscription.

It is assumed that the SaaS provider's business objectives are (i) to maximize revenue

generation while (ii) to minimizing cost. Maximization of revenue generation varies directly with number of clients serviced. On the other hand, minimization of cost varies directly with the amount of hardware that is purchased over time. The choice of using a IaaS or PaaS provider or buying owned hardware directly affects SaaS provider cost.

3.9 Summary

Cloud computing revolves around the idea of Internet scale delivery of on demand computing power to external customers. It is probably the most outstanding example of large scale resource sharing systems.

Cloud contains a large number of computing nodes. Currently Clouds are formed by interconnected set of datacenters. A data center environment is composed of communication system, servers, and storage subsystem. In Cloud, users are multiplexed using multi-tenancy techniques such as virtualization, OS processes, container level thread pools, or application level techniques.

Resource oriented Clouds are referred to as Infrastructure-as-a-Service (IaaS) while software oriented ones are either Software-as-a-Service (SaaS) or Platform-as-a-Service (PaaS). IaaS multiplexes the hardware layer and offers computing services such as storage, CPU and memory to users using hardware virtualization techniques. Two common use cases of current IaaS Clouds are setting up e-commerce websites and performing data intensive processing using high performance clusters. A PaaS Cloud offers application environments or programmable services to customers for various durations of time. A SaaS provider offers instances of services (e.g. CRM) by deploying them on its own infrastructure. Services are accessed by users through network. One of the differences between a SaaS service and a normal multi-user web service is the fact that application deployed on

SaaS is oblivious to the fact that it is instantiated several times. In other words multi-tenancy is transparent to the application through certain APIs.

Chapter 4

Conclusion

This report presented an overview of resource sharing systems (RSS), architecture of typical large scale RSS. It also discussed the problem of complexity in managing such systems and introduced autonomic computing as a promising solution to handle this complexity. The aim was at building self-managing complex resource sharing systems that manage themselves in accordance with high-level objectives specified by humans [36]. The autonomic computing architecture discussed was the one proposed by IBM based on Monitor-Analyze-Plan-Execute (MAPE loop) model. The approach discussed to implement such autonomic control loop here was using different element of mathematics such as statistics, optimization, and control theory.

Cloud computing as a promising potential ULSRSS was presented. Cloud provisions scalable on-demand computational power using a unified network of datacenters. Datacenter as the main building block of the cloud was elaborated architecture wise. Virtualization as the main tool to provide high-scale multiplexing in computational environment and the enabler of IaaS Cloud was discussed. Different types of cloud offering, was presented and objectives and pricing model of each type of offering was discussed.

In the following subsections, we enumerate open problems in the introduced domain that are most probably subject of further investigations by author.

4.1 Open Problems

There are many open problems in the area of ultra large scale computing resource delivery. However, problems mentioned here, are in the space formed by intersection of cloud computing and autonomic computing.

4.1.1 Performance Model Identification of Service Instances in a Ultra Large Resource Sharing System

Estimation techniques have been largely applied to track hidden performance parameters (e.g. service demands) of web based software systems. In a ULSRSS, there are variable number of dynamically deployed applications or service instances that are utilizing the shared system resources. Modeling behavior of these service instances in terms of performance under different resource entitlements will help in discovering resource requirement and proper resource allocation given a set of performance objectives for each service instance. If each service instance is treated individually, the cost of performance model estimation (i.e. in terms of computation) becomes excessive. Thus, it seems natural to try to group the service instances with similar resource usage into a smaller number of classes aiming at finding a low complexity model yet with enough accuracy. Combination of clustering algorithm and tracking filter can be used for effective grouping of service instances, Since the resource demands associated with each service instance may change with time (e.g. due to workload changes) adaptive clustering techniques that regroup the service instances depending on their time-varying resource demands should be deployed.

4.1.2 Optimization of Resource Distribution to Applications in Cloud

The main objective of a cloud provider is to maximize profit (i.e., revenue – cost). Optimization techniques allow the provider to determine resource allocations to various clients in order to best maximize its revenue while minimizing its costs. Due to economic benefits, optimization has been the subject of much investigation [139, 133, 134, 135, 140, 141, 142, 143].

One type of optimization possible for a PaaS provider is when the price model and service level agreement (SLA) are defined in terms of performance measures, and the amount of supplied resource is decided by the provider based on the resource cost and SLA targets. For example, in a virtualization based PaaS Cloud, it is common to tune the performance of an application and the provider's cost of operation through scale-up/down (i.e., adding/removing resource to individual virtual machines (VM)), scale-out/in (i.e., adding/removing VMs to an application environment), and migration (i.e., moving VMs over the physical infrastructure). An interesting problem involves the allocation of resources in this type of Cloud such that cost to the provider is minimized (through a maximization of resource sharing) while attempting to meet all client application requirements as specified in their respective Service Level Agreements (SLA)s¹ [87, 130, 131, 132].

In a IaaS cloud, optimization is decomposed into dynamic infrastructure pricing mechanism offered by provider [144, 145] and elastic resource allocation policies employed by individual consumers [146, 147, 148] to satisfy their QoS requirements. Strategies taken by provider regarding dynamic infrastructure pricing mechanisms based on optimizations on the market model and users' behaviors can be an interesting subject to look.

¹An SLA is a contract which defines the relationship between a service provider and its clients that fully specifies all obligations for both parties, the price to be paid for the service(s) offered and associated penalties should obligations be unmet. It can be quite complex and comprehensive (e.g., considering aspects of both functional and non-functional requirements); however, here, we limit the discussion to performance objectives that can be extracted from an SLA.

4.1.3 Comparing Alternatives Approaches to Autonomic Resource Management

At the heart of any autonomic resource management system, there is an implicit or explicit feedback loop that maps the current system measures to the proper action(s) to be taken by an actuator. Two alternative approaches for implementing this autonomic management loop are as follows: (1) through the use of a control theoretic approach; and (2) through specification of a set of heuristic rules (sometimes called policy based management² in narrower sense). While the first approach is based on mathematical modeling and is designed more with robustness than comprehensibility in mind the second approach is more intuitive, lightweight and easier for a human administrator to interpret,

In control theoretic approaches, a decision for selection of resources to allocate for an application, can be made dynamically based on current application performance and its projection on modified resources. The projection is usually obtained by a performance model, which quantitatively relates application level quality of service (QoS) metrics (e.g., response time of a customer application) with a given resource.

In the heuristic, rule-based approach, Event-Condition-Action rules are utilized instead of robust control formulas (see [90, 93, 94, 91, 92]). For example, thresholds are defined on things like CPU utilization of an individual virtual machine (VM) instance hosting an application server for an application. Should this threshold be breached (e.g., “If CPU utilization is less than 20 for 10 minutes”) a request may be sent (action) for an elastic increase to occur (i.e., of VM instances in this tier of the application).

An interesting subject is to form a comparison between these different methods. different criteria such as design complexity, ease of comprehension, and maintenance can be used

²A policy can be understood to represent “...any type of formal behavioral guide” that is input to the system [86].

for this comparison. The resulting research will help in determining how these approaches can be used in the governance of resources to better meet a high-level goal over time. It also helps in tailoring low-cost durable solutions for specific problems.

4.1.4 Hybrid Autonomic Management Techniques

Different approaches might be used to obtain proper real-time actions with accordance to goals. In policy based management, actions are derived from a set of *policies* encapsulating management logic. Under certain situation (regarding environmental) and assumptions (regarding system behavior) possibly for a closed time interval, goals can be decomposed into a set of policies. These policies then guide the system towards achieving the goals by governing the actions taken in that specific time interval. At run-time, should these assumptions underpinning these policies prove incorrect, the system may behave poorly and different goal decompositions [96] should be tried.

Attempting to fully automate derivation of this decomposition from a set of given objectives and environmental conditions would be too optimistic. On the other hand, relying on human actors to get proper decompositions contradict the initial motivation of autonomic computing. Thus, exploring automated techniques for process of policy selection can be interesting; The rigorous approaches to autonomic computing discussed in this report can be combined with policy based management (rather than be used in isolation) to form robust autonomic management systems. For example if a *system model* exists that can predict some system properties in future resulting from taking a certain policy set, policy set selection can take advantage of that, and more automated solution might emerge.

Bibliography

- [1] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of High Performance Computing Applications*, vol. 11, no. 2, p. 115, 1997.
- [2] I. Foster, C. Kesselman, J. Nick, and S. Tuecke, "Grid services for distributed system integration," *Computer*, pp. 37–46, 2002.
- [3] B. Hayes, "Cloud computing," *Communications of the ACM*, vol. 51, no. 7, pp. 9–11, 2008.
- [4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, February 2009, <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html>.
- [5] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan, "The reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, pp. 535–545, july 2009.
- [6] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, 2009.
- [7] W. Tsai, C. Fan, Y. Chen, and R. Paul, "DDSOS: a dynamic distributed service-oriented simulation framework," *Simulation Symposium, 2006. 39th Annual*, 2006.
- [8] W. Tsai, Z. Cao, X. Wei, R. Paul, Q. Huang, and X. Sun, "Modeling and simulation in service-oriented software development," *Simulation*, vol. 83, no. 1, pp. 7–32, 2007.
- [9] M. Wolf, Z. Cai, W. Huang, and K. Schwan, "SmartPointers: Personalized Scientific Data Portals In Your Hand," in *Supercomputing, ACM/IEEE 2002 Conference*. IEEE, pp. 20–20.

- [10] S. Babu and J. Widom, “Continuous queries over data streams,” *ACM Sigmod Record*, vol. 30, no. 3, pp. 109–120, 2001.
- [11] V. Kumar, B. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan, “Resource-Aware Distributed Stream Management Using Dynamic Overlays,” in *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*. IEEE, pp. 783–792.
- [12] A. Huang and P. Steenkiste, “Network-sensitive service discovery,” *Journal of Grid Computing*, vol. 1, no. 3, pp. 309–326, 2003.
- [13] “radiantGrid platform,” <http://www.radiantgrid.com/feature-grid/> [online August 2011].
- [14] A. Turing, “The chemical basis of morphogenesis,” *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, vol. 237, no. 641, p. 37, 1952.
- [15] A. Gierer and H. Meinhardt, “A theory of biological pattern formation,” *Biological Cybernetics*, vol. 12, no. 1, pp. 30–39, 1972.
- [16] “Red Hat Global File System,” <http://www.redhat.com/gfs/> [online August 2011].
- [17] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand, “Parallax: managing storage for a million machines,” in *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*. Berkeley, CA, USA: USENIX Association, 2005, pp. 4–4. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1251123.1251127>
- [18] “VMware’s VMFS,” <http://www.vmware.com/products/vmfs/overview.html> [online August 2011].
- [19] “Hadoop Distributed File System,” <http://hadoop.apache.org/hdfs/> [online August 2011].
- [20] D. Abadi, “Problems with cap, and yahoos little known nosql system,” 2010. [Online]. Available: <http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html>
- [21] “An architectural blueprint for autonomic computing.” 2005.
- [22] O. Babaoglu, M. Jelasity, and A. Montresor, “Grassroots approach to self-management in large-scale distributed systems,” *Unconventional Programming Paradigms*, pp. 286–296, 2005.

- [23] B. Ensink and V. Adve, "Coordinating Adaptations in Distributed Systems," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*. IEEE Computer Society, 2004, pp. 446–455.
- [24] F. Chang and V. Karamcheti, "Automatic configuration and run-time adaptation of distributed applications," in *hpdc*. Published by the IEEE Computer Society, 2000, p. 11.
- [25] J. Cangussu, K. Cooper, and C. Li, "A control theory based framework for dynamic adaptable systems," in *Proceedings of the 2004 ACM symposium on Applied computing*. ACM, 2004, pp. 1546–1553.
- [26] J. Loyall, R. Schantz, J. Zinky, and D. Bakken, "Specifying and measuring quality of service in distributed object systems," in *Object-Oriented Real-time Distributed Computing, 1998.(ISORC 98) Proceedings. 1998 First International Symposium on*. IEEE, 1998, pp. 43–52.
- [27] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker, "Agile application-aware adaptation for mobility," *ACM SIGOPS Operating Systems Review*, vol. 31, no. 5, pp. 276–287, 1997.
- [28] R. Balan, M. Satyanarayanan, S. Park, and T. Okoshi, "Tactics-based remote execution for mobile computing," in *Proceedings of the 1st international conference on Mobile systems, applications and services*. ACM, 2003, pp. 273–286.
- [29] R. P. Doyle, J. S. Chase, O. M. Asad, W. Jin, and A. M. Vahdat, "Model-based resource provisioning in a web service utility," in *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*. Seattle, WA: USENIX Association, 2003, pp. 5–5.
- [30] C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, "A scalable solution to the multi-resource QoS problem," in *Real-Time Systems Symposium, 1999. Proceedings. The 20th IEEE*. IEEE, 1999, pp. 315–326.
- [31] A. Huang and P. Steenkiste, "Building self-configuring services using service-specific knowledge," in *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*. IEEE, pp. 45–54.
- [32] M. N. Bennani and D. A. Menascé, "Assessing the robustness of self-managing computer systems under highly variable workloads," *Autonomic Computing, International Conference on*, vol. 0, pp. 62–69, 2004.
- [33] M. Mesnier, E. Thereska, G. Ganger, D. Ellard, and M. Seltzer, "File classification in self-* storage systems," 2004.

- [34] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase, “Correlating instrumentation data to system states: A building block for automated diagnosis and control,” in *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation-Volume 6*. USENIX Association, 2004, pp. 16–16.
- [35] Y. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. Wang, C. Yuan, and Z. Zhang, “STRIDER: A black-box, state-based approach to change and configuration management and support,” in *Proceedings of the 17th USENIX conference on System administration*. USENIX Association, 2003, pp. 159–172.
- [36] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, p. 4150, 2003.
- [37] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure,” *Computer*, vol. 37, no. 10, pp. 46–54, 2004.
- [38] G. Tesauro and J. O. Kephart, “Utility functions in autonomic systems,” in *Proceedings of the First International Conference on Autonomic Computing*. IEEE Computer Society, 2004, pp. 70–77.
- [39] “Autonomic computing in Canadian academia,” <http://www.ibm.com/developerworks/library/ac-canada/index.html> [online August 2011].
- [40] “collectd The system statistics collection daemon,” <http://collectd.org/> [online August 2011].
- [41] “Nagios IT Infrastructure Monitoring,” <http://www.nagios.org/> [online August 2011].
- [42] “Java Management Extensions (JMX),” <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html> [online August 2011].
- [43] N. Kasabov, *Foundations of neural networks, fuzzy systems, and knowledge engineering*, 1996.
- [44] M. Kutner, C. Nachtsheim, and J. Neter, *Applied linear regression models*. McGraw-Hill New York, NY, 2004.
- [45] D. Ratkowsky, *Handbook of nonlinear regression models*. M. Dekker, 1990.
- [46] A. Pankratz, *Forecasting with Univariate Box-Jenkins Models*. Wiley Online Library, 2008, vol. 3.

- [47] G. Welch and G. Bishop, “An introduction to the Kalman filter,” *University of North Carolina at Chapel Hill, Chapel Hill, NC*, 1995. [Online]. Available: <http://www.cs.unc.edu/~welch/kalman/kalmanIntro.html>
- [48] D. C. Petriu, “Approximate mean value analysis of client-server systems with multi-class requests,” in *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems*. ACM New York, NY, USA, 1994, pp. 77–86.
- [49] D. C. Petriu and C. M. Woodside, “Approximate mean value analysis based on markov chain aggregation by composition,” *Linear Algebra and its Applications*, vol. 386, pp. 335–358, 2004.
- [50] E. Badidi, L. Esmahi, and M. Serhani, “A queuing model for service selection of multi-classes qos-aware web services,” in *Third IEEE European Conference on Web Services (ECOWS)*, nov 2005, p. 9.
- [51] J. A. Rolia and K. C. Sevcik, “The method of layers,” *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689–700, 1995.
- [52] S. Ramesh and H. G. Perros, “A multi-layer client-server queueing network model with synchronous and asynchronous messages,” in *Proceedings of the 1st international workshop on Software and performance*. ACM New York, NY, USA, 1998, pp. 107–119.
- [53] M. Litoiu, M. Woodside, and T. Zheng, “Hierarchical model-based autonomic control of software systems,” *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, May 2005.
- [54] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy, “Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates,” *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 2, 2006.
- [55] H. Ghanbari, M. Litoiu, M. Woodside, T. Zheng, J. Wong, and G. Iszlai, “Tuning tracking of performance model parameters using dynamic job classes,” in *Proceedings of the second ACM/SPEC International Conference on Performance Engineering (ICPE 2011)*, to appear. ACM, 2011.
- [56] T. K. Liu, S. Kumaran, and Z. Luo, “Layered queuing models for enterprise javabean applications,” in *Proc. 5th International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 4–7.
- [57] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1984.

- [58] D. Menasce, V. Almeida, L. Dowdy, and L. Dowdy, *Performance by design: computer capacity planning by example*. Prentice Hall, 2004.
- [59] J. A. Rolia and K. C. Sevcik, “The method of layers,” *IEEE Transactions on Software Engineering*, vol. 21, no. 8, pp. 689–700, 1995.
- [60] D. C. Petriu and C. M. Woodside, “Approximate MVA from markov model of software client/server systems,” in *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*. Dallas, Texas: IEEE Computer Society, 1991, pp. 322–329.
- [61] G. Pacifici, W. Segmuller, M. Spreitzer, and A. Tantawi, “CPU demand for web serving: Measurement analysis and dynamic estimation,” *Performance Evaluation*, vol. 65, no. 6-7, pp. 531–553, 2008.
- [62] J. Rolia and V. Vetland, “Parameter estimation for performance models of distributed application systems,” in *Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 1995, p. 54.
- [63] Q. Zhang, L. Cherkasova, and E. Smirni, “A Regression-Based analytic model for dynamic resource provisioning of Multi-Tier applications,” in *Proceedings of the 4th IEEE International Conference on Autonomic Computing*. Jacksonville, Florida: IEEE Computer Society, 2007, p. 27.
- [64] J. Rolia and V. Vetland, “Correlating resource demand information with ARM data for application services,” in *Proceedings of the 1st international workshop on Software and performance*. ACM New York, NY, USA, 1998, pp. 219–230.
- [65] M. Courtois and M. Woodside, “Using regression splines for software performance analysis,” in *Proceedings of the 2nd international workshop on Software and performance*. Ottawa, Ontario, Canada: ACM New York, NY, USA, 2000, pp. 105–114.
- [66] L. Zhang, C. Xia, M. Squillante, and W. N. M. III, “Workload service requirements analysis: A queueing network optimization approach,” in *Proceedings of 10th IEEE International Symposium on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems*. Washington, DC: IEEE Computer Society, 2002.
- [67] Z. Liu, L. Wynter, C. H. Xia, and F. Zhang, “Parameter inference of queueing models for it systems using end-to-end measurements,” *Performance Evaluation, Elsevier*, vol. 63, no. 1, pp. 36–60, 2006.
- [68] S. Kraft, S. Pacheco-Sanchez, G. Casale, and S. Dawson, “Estimating service resource consumption from response time measurements,” in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies*

and Tools, ser. VALUETOOLS '09. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009, pp. 48:1–48:10.

- [69] M. Woodside, T. Zheng, and M. Litoiu, “The use of optimal filters to track parameters of performance models,” in *Quantitative Evaluation of Systems, 2005. Second International Conference on the*, 2005, pp. 74–83.
- [70] J. Xu, A. Oufimtsev, M. Woodside, and L. Murphy, “Performance modeling and prediction of enterprise JavaBeans with layered queuing network templates,” in *Proceedings of the 2005 conference on Specification and verification of component-based systems (SAVCBS '05)*. Lisbon, Portugal: ACM, 2005.
- [71] 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, p. 345.
- [72] S. Aiber, D. Gilat, A. Landau, N. Razinkov, A. Sela, and S. Wasserkrug, “Autonomic self-optimization according to business objectives,” in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, pp. 206–213.
- [73] S. Abdelwahed, N. Kandasamy, and S. Neema, “A control-based framework for self-managing distributed computing systems,” in *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. ACM, 2004, pp. 3–7.
- [74] N. Kandasamy, S. Abdelwahed, and J. Hayes, “Self-optimization in computer systems via on-line control: application to power management,” in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, pp. 54–61.
- [75] V. Bhat, M. Parashar *et al.*, “Enabling self-managing applications using model-based online control strategies,” in *2006 IEEE International Conference on Autonomic Computing*. IEEE, 2006, pp. 15–24.
- [76] 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 6th international conference on Autonomic computing (ICAC '09)*. Barcelona, Spain: ACM, Jun. 2009, pp. 117–126.
- [77] T. F. Abdelzaher, K. G. Shin, and N. Bhatti, “Performance guarantees for web server end-systems: A control-theoretical approach,” *IEEE Transactions on Parallel and Distributed Systems*, pp. 80–96, 2002.

- [78] T. Abdelzaher and C. Lu, "Modeling and performance control of internet servers," in *Proceedings of the 39th IEEE Conference on Decision and Control*, vol. 3. IEEE, 2000, pp. 2234–2239.
- [79] 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 *Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium.*, 2003, pp. 208–217.
- [80] T. F. Abdelzaher, J. Stankovic, C. Lu, R. Zhang, and Y. Lu, "Feedback performance control in software services," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 74–90, Jun. 2003.
- [81] Z. Lu, J. Hein, M. Humphrey, M. Stan, J. Lach, and K. Skadron, "Control-theoretic dynamic frequency and voltage scaling for multimedia workloads," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems.* ACM, 2002, pp. 156–163.
- [82] V. Sharma, A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu, "Power-aware QoS management in Web servers," in *Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE.* IEEE, pp. 63–72.
- [83] J. Hellerstein and I. ebrary, *Feedback control of computing systems.* Wiley Online Library, 2004.
- [84] C. Lu, G. Alvarez, and J. Wilkes, "Aqueduct: online data migration with performance guarantees," in *Proceedings of the Conference on File and Storage Technologies.* USENIX Association, 2002, pp. 219–230.
- [85] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using control theory to achieve service level objectives in performance management," *Real-Time Systems*, vol. 23, no. 1, pp. 127–141, 2002.
- [86] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *POLICY '04: Proceedings of the 5th IEEE International Workshop on Policies for Distributed Systems and Networks.* IEEE Computer Society, June 2004, pp. 3–12.
- [87] J. Sauvé, F. Marques, A. Moura, M. C. Sampaio, J. Jornada, and E. Radziuk, "SLA design from a business perspective," *Proceedings of the 16th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM)*, pp. 72–83, October 2005.

- [88] M. Beigi, S. Calo, and D. Verma, “Policy transformation techniques in policy-based systems management,” in *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*., june 2004, pp. 13 – 22.
- [89] L. Su, D. Chadwick, A. Basden, and J. Cunningham, “Automated decomposition of access control policies,” in *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005)*., june 2005, pp. 3 – 13.
- [90] M. Desertot, C. Escoffier, and D. Donsez, “Autonomic management of J2EE edge servers,” in *Proceedings of the 3rd international workshop on Middleware for grid computing*. Grenoble, France: ACM, 2005, p. 16.
- [91] T. Chieu, A. Mohindra, A. Karve, and A. Segal, “Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment,” in *Proceedings of the 2009 IEEE International Conference on e-Business Engineering*. IEEE Computer Society, 2009, pp. 281–286.
- [92] Y. Zhang, G. Huang, X. Liu, and H. Mei, “Integrating Resource Consumption and Allocation for Infrastructure Resources on-Demand,” in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*. IEEE, 2010, pp. 75–82.
- [93] “Rightscale autoscaling using voting tags,” http://support.rightscale.com/12-Guides/Lifecycle_Management/03_-_Understanding_Key_Concepts/RightScale_Alert_System/Alerts_based_on_Voting_Tags/Understanding_the_Voting_Process [online January 2011].
- [94] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, “Elastic management of cluster-based services in the cloud,” in *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ser. ACDC ’09. New York, NY, USA: ACM, 2009, pp. 19–24.
- [95] B. Simmons and H. Lutfiya, “Strategy-trees: A feedback based approach to policy management,” in *Proceedings of the 3rd IEEE international workshop on Modelling Autonomic Communications Environments*, ser. MACE ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 26–37.
- [96] J. Rubio-Loyola, J. Serrat, M. Charalambides, P. Flegkas, and G. Pavlou, “A methodological approach toward the refinement problem in policy-based management systems,” *Communications Magazine, IEEE*, vol. 44, no. 10, pp. 60 –68, 2006.
- [97] I. Foster, Y. Zhao, I. Raicu, and S. Lu, “Cloud computing and grid computing 360-degree compared,” in *Grid Computing Environments Workshop, 2008. GCE ’08*, nov. 2008, pp. 1 –10.

- [98] “Mongrel, a HTTP server for Ruby,” <http://rubyforge.org/projects/mongrel/> [online August 2011].
- [99] “Webrick, a HTTP server for Ruby,” <http://www.ruby-doc.org/stdlib/libdoc/webrick/rdoc/index.html> [online August 2011].
- [100] “Thin, a HTTP server for Ruby,” <http://code.macournoyer.com/thin/> [online August 2011].
- [101] “Apache Tomcat, an application server for Java Servlet and Java Server Pages,” <http://tomcat.apache.org/> [online August 2011].
- [102] “Apache HTTP server,” <http://httpd.apache.org/> [online August 2011].
- [103] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [104] “VMware virtualization technology,” <http://www.vmware.com> [online August 2011].
- [105] N. Fallenbeck, H. J. Picht, M. Smith, and B. Freisleben, “Xen and the art of cluster scheduling,” in *Proc. of 1st International Workshop on Virtualization Technology in Distributed Computing*. USA: IEEE Computer Society, Nov. 2006.
- [106] “KVM, a kernel based virtual machine,” <http://www.linux-kvm.org> [online August 2011].
- [107] Amazon, *Elastic Compute Cloud*. [Online]. Available: <http://aws.amazon.com/ec2/>
- [108] “ElasticHosts Cloud hosting,” <http://www.elastichosts.com/> [online August 2011].
- [109] “GoGrid Cloud Hosting,” <http://www.gogrid.com/> [online August 2011].
- [110] “FlexiScale Cloud Hosting,” <http://www.flexiant.com/products/flexiscale/> [online August 2011].
- [111] “Rackspace Cloud Hosting,” <http://www.rackspace.com/> [online August 2011].
- [112] “Eucalyptus open source Cloud platform,” <http://open.eucalyptus.com/> [online August 2011].
- [113] “OpenNebula cloud management solution,” <http://opennebula.org/> [online August 2011].
- [114] “OpenStack, an open source software for building private and public clouds,” <http://www.openstack.org/> [online August 2011].

- [115] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif, “Black-box and gray-box strategies for virtual machine migration,” in *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2007, p. 229242.
- [116] D. Gmach, J. Rolia, L. Cherkasova, G. Belrose, T. Turicchi, and A. Kemper, “An integrated approach to resource pool management: Policies, efficiency and quality metrics,” in *IEEE International Conference on Dependable Systems and Networks*. IEEE, 2008, pp. 326–335.
- [117] S. Mehta and A. Neogi, “Recon: A tool to recommend dynamic server consolidation in multi-cluster data centers,” in *Network Operations and Management Symposium (NOMS '08)*. IEEE, pp. 363–370.
- [118] A. Verma, P. Ahuja, and A. Neogi, “pmapper: power and migration cost aware application placement in virtualized systems,” *Middleware 2008*, pp. 243–264, 2008.
- [119] B. Urgaonkar, P. Shenoy, and T. Roscoe, “Resource overbooking and application profiling in shared hosting platforms,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 239–254, 2002.
- [120] N. Bobroff, A. Kochut, and K. Beaty, “Dynamic placement of virtual machines for managing sla violations,” in *10th IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 21 2007–yearly 25 2007, pp. 119 –128.
- [121] D. Shmoys and É. Tardos, “An approximation algorithm for the generalized assignment problem,” *Mathematical Programming*, vol. 62, no. 1, pp. 461–474, 1993.
- [122] R. Gupta, S. Bose, S. Sundarrajan, M. Chebiyam, and A. Chakrabarti, “A two stage heuristic algorithm for solving the server consolidation problem with item-item and bin-item incompatibility constraints,” in *Services Computing, 2008. SCC'08. IEEE International Conference on*, vol. 2. IEEE, pp. 39–46.
- [123] “Virtual Distributed Ethernet (VDE),” <http://vde.sourceforge.net/> [online August 2011].
- [124] “OpenFlow, Network Virtualization for Infrastructure as a Service Clouds,” 2010, <http://cloudscaling.com/blog/wp-content/uploads/2009/12/iaas-building-guide-v1.pdf> [online August 2011].
- [125] “Open vSwitch, an open source production quality, multilayer software virtual switch,” <http://openvswitch.org/> [online August 2011].
- [126] “VMWare ESX server virtual networking concepts,” www.vmware.com/files/pdf/virtual_networking_concepts.pdf [online August 2011].

- [127] “Eucalyptus Network Configuration (1.6),” http://open.eucalyptus.com/wiki/EucalyptusNetworking_v1.6 [online August 2011].
- [128] “Amazon Virtual Private Cloud (Amazon VPC),” <http://aws.amazon.com/vpc/> [online August 2011].
- [129] “Google App Engine PaaS Cloud,” <http://code.google.com/appengine/> [online August 2011].
- [130] I. n. Goiri, F. Julia, J. Ejarque, M. d. Palol, R. M. Badia, J. Guitart, and J. Torres, “Introducing virtual execution environments for application lifecycle management and sla-driven resource distribution within service providers,” in *Proceedings of the 2009 Eighth IEEE International Symposium on Network Computing and Applications*, ser. NCA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 211–218.
- [131] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, and R. Yahyapour, “Establishing and monitoring slas in complex service based systems,” in *Proceedings of the 2009 IEEE International Conference on Web Services*, ser. ICWS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 783–790.
- [132] A. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, C. Zsigri, R. Sirvent, J. Guitart, R. Badia, K. Djemame, W. Ziegler *et al.*, “OPTIMIS: a Holistic Approach to Cloud Service Provisioning,” *Future Generation Computer Systems*, *accepted*, 2010.
- [133] J. Z. Li, J. Chinneck, M. Woodside, and M. Litoiu, “Fast scalable optimization to configure service systems having cost and quality of service constraints,” in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 159–168.
- [134] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai, “Performance model driven QoS guarantees and optimization in clouds,” in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*. IEEE Computer Society, 2009, pp. 15–22.
- [135] H. Li, G. Casale, and T. Ellahi, “SLA-driven planning and optimization of enterprise applications,” in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM, 2010, pp. 117–128.
- [136] A. Soror, U. Minhas, A. Aboulnaga, K. Salem, P. Kokosielis, and S. Kamath, “Automatic virtual machine configuration for database workloads,” *ACM TRANSACTIONS ON DATABASE SYSTEMS*, vol. 35, no. 1, p. Article 7, 2010.
- [137] “Software as a Service examples by Cloud Taxonomy,” <http://cloudtaxonomy.opencrowd.com/taxonomy/software-as-a-service/> [online August 2011].

- [138] “Salesforce enterprise cloud computing,” <http://www.salesforce.com/> [online August 2011].
- [139] X. Liu, X. Zhu, P. Padala, Z. Wang, and S. Singhal, “Optimal multivariate control for differentiated services on a shared hosting platform,” in *Proceedings of the 46th IEEE Conference on Decision and Control*. IEEE, 2007, pp. 3792–3799.
- [140] H. N. Van, F. D. Tran, and J. M. Menaud, “SLA-Aware virtual resource management for cloud infrastructures,” in *IEEE Ninth International Conference on Computer and Information Technology*. IEEE, 2009, pp. 357–362.
- [141] G. Tesauro, W. E. Walsh, and J. O. Kephart, “Utility-Function-Driven resource allocation in autonomic systems,” in *Proceedings of the Second International Conference on Automatic Computing*. IEEE Computer Society, 2005, pp. 342–343.
- [142] X. Wang, Z. Du, Y. Chen, S. Li, D. Lan, G. Wang, and Y. Chen, “An autonomic provisioning framework for outsourcing data center based on virtual appliances,” *Cluster Computing*, vol. 11, no. 3, pp. 229–245, 2008.
- [143] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Wang, “Appliance-Based autonomic provisioning framework for virtualized outsourcing data center,” in *Proceedings of the Fourth International Conference on Autonomic Computing*. IEEE Computer Society, 2007, p. 29.
- [144] H. H. Huang, “A control-theoretic approach to automated local policy enforcement in computational grids,” *Future Generation Computer Systems*, vol. 26, no. 6, pp. 787 – 796, 2010.
- [145] C. S. Yeo, S. Venugopal, X. Chu, and R. Buyya, “Autonomic metered pricing for a utility computing service,” *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1368 – 1380, 2010.
- [146] 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.
- [147] L. Rodero-Merino, L. M. Vaquero, V. Gil, F. Galn, J. Fontn, R. S. Montero, and I. M. Llorente, “From infrastructure delivery to service management in clouds,” *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1226 – 1240, 2010.
- [148] M. A. Murphy and S. Goasguen, “Virtual organization clusters: Self-provisioned clouds on the grid,” *Future Generation Computer Systems*, vol. 26, no. 8, pp. 1271 – 1281, 2010.