Revenue Maximization Problems in Commercial Data Centers

Thesis by

Michele Mazzucco

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

University of Newcastle upon Tyne

Newcastle upon Tyne, UK

2009

(Defended May 7, 2009)

Acknowledgements

I am deeply grateful to the following people for their help and support during the period I have spent at the School of Computing Science.

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Isi Mitrani, for his support, guidance and insight as well as for the many valuable suggestions and comments he made. Isi is not just a very good supervisor, he is a marvellous person too. Isi, working with you has been a great pleasure and honour.

Secondly, I would like to express my gratitude to the people that have been around me in the last three years. During discussions in the office, coffee breaks, heavy training sessions in the gym or in the swimming pool, football matches, meals and trips to the beach in Tynemouth, my colleagues and friends have always been keen to help me out with my problems. A special mention goes to Joris Slegers for clarifying some of my doubts about queueing systems, Marios Andreou for discussing with me some of the low-level details of the TCP protocol, Christiaan Lamprecht for the conversations about race conditions and other issues arising in multi-threaded systems, Jennie Palmer for running the simulations shown in Chapter 3, and Manuel Mazzara for his friendship and wise suggestions.

Outside the University, I'd like to thank Panos, Costas (both), Argyris, Marios and Carlo. Guys, you rock! I am also thankful to my old friends, and in particular to Davide, Andrea (both), Mirco and Gabriele, because every time I go back home they make me feel as if I had never left.

My research has been funded partly by the research project QoSP (Quality of Service Provisioning), sponsored by British Telecom and, for a short but vital period, by Isi's personal funds. Apart from the financial support, I am very grateful to Mike Fisher and Paul McKee of British Telecom for their precious feedback and comments.

Last but not least, I would like to thank my parents for supporting and encouraging me. Without their constant support I would not have made it.

Abstract

As IT systems are becoming more important everyday, one of the main concerns is that users may face major problems and eventually incur major costs if computing systems do not meet the expected performance requirements: customers expect reliability and performance guarantees, while underperforming systems loose revenues. Even with the adoption of data centers as the hub of IT organizations and provider of business efficiencies the problems are not over because it is extremely difficult for service providers to meet the promised performance guarantees in the face of unpredictable demand. One possible approach is the adoption of Service Level Agreements (SLAs), contracts that specify a level of performance that must be met and compensations in case of failure.

In this thesis I will address some of the performance problems arising when IT companies sell the service of running 'jobs' subject to Quality of Service (QoS) constraints. In particular, the aim is to improve the efficiency of service provisioning systems by allowing them to adapt to changing demand conditions.

First, I will define the problem in terms of an utility function to maximize. Two different models are analyzed, one for single jobs and the other useful to deal with session-based traffic. Then, I will introduce an autonomic model for service provision. The architecture consists of a set of hosted applications that share a certain number of servers. The system collects demand and performance statistics and estimates traffic parameters. These estimates are used by management policies which implement dynamic resource allocation and admission algorithms. Results from a number of experiments show that the performance of these heuristics is close to optimal.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

IT systems are becoming more important everyday as they help to support most aspects of our daily life. The wide variety of services and resources available over the Internet presents new opportunities and new challenges for companies and organizations. People and organizations rely on IT systems to address most of their needs, such as health and entertainment, and to access news and services. Thus, in the last few years one of the main challenges for information technology has been the integration of applications within and across organizational boundaries. More recently the adoption of Web Services, self-describing, open components that support rapid, low-cost composition of distributed applications [82], has captured the attention of both companies and academia as a promising solution to low cost and immediate integration with other applications and partners: the use of Web Services, in fact, eases systems' interoperability because they allow programs to interact with each other over the Internet via open protocols and standards like SOAP [114] and HTTP [56]. As a result now many traditional providers such as Google and Amazon are boosting their traffic through Web Service APIs [7, 43].

One of the main concerns, however, is that users may face major breakdowns and eventually incur major costs if IT systems do not meet the expected performance requirements. In fact, as Web Services proliferate more and more widely, whether offered within an organization or as part of a paid service across organization boundaries, the issues related to service quality become very relevant and they will eventually be a significant factor in distinguishing the success or the failure of service providers. For example, when customers contact customer service centers they usually interact with automated support systems and expect an immediate answer. Unfortunately, if the issue requires human intervention, it is likely that customers are put on hold before being able to speak to

a human being. Similarly, when the stock market is unstable, a large number of online traders tend to overload the trading sites, making the systems non-responsive. The inability to buy and sell in a timely manner may cause major financial losses. These situations cause a lot of frustration and are a major cause for companies to lose customers and revenues. For example, it has been reported that Amazon tried delaying the page generation by 100 milliseconds and found out that even very small delays would result in substantial and costly drops in revenue (1% sales drop for 100ms delay) [68]. Also, Google found that an extra 0.5 seconds in search page generation would kill user satisfaction, with a consequent 20% traffic drop. Finally, a broker could lose 4 USD million in revenues per millisecond if their electronic trading platform is 5 milliseconds behind the competition.

As emerges from the above examples businesses are under constant pressure, and IT organizations must respond to these pressures in their own operations in order to provide solutions to help the whole business more efficiently. The usual approach is the adoption of the data center as the hub of IT organizations and provider of business efficiencies. The problem, however, is still rather complicated: it is extremely difficult for service providers to meet the promised performance guarantees in the face of unpredictable user demand. One approach that can offer some comfort in certain situations is the adoption of Service Level Agreements (SLAs), contracts that specify a level of performance that must be met and compensation in case of failure.

Quality of Service (QoS) can be addressed from several points of view, such as the engineering point of view (*i.e.*, how to provide a service subject to performance constraints), or the semantic one (*i.e.*, how to dynamically discover or select services with tight performance requirements [117], or how to negotiate QoS requirements at run time [35]). This thesis is focused in the former one. Specifically, it addresses some of the performance issues arising when companies sell the service of running 'jobs' subject to QoS constraints. From the provider's perspective, the problem can be defined as *'How to maximize the earned revenue?'* or, *'How to minimize the probability of failing to honour the commitments for agreed service quality?'*.

1.2 Thesis Organization and Contributions

This thesis has two strands. First, it considers the problem of how to structure and control a service provisioning system subject to Quality of Service constraints. Second, it presents an architecture of a hosting system where a number of servers are used to provide different types of Web Services

to paying customers. The system must handle demands of different types and must adapt dynamically to changing conditions. There are different Service Level Agreements specifying charges for running jobs and penalties for failing to meet promised performance metrics. This dissertation introduces different models whose objective is to maximize an utility function based on the average revenue earned per unit time. The system collects demand and performance statistics, estimates parameters and makes dynamic decisions concerning server allocation and admission control. The proposed algorithms are based on approximate mathematical models of system behaviour. Their effectiveness and robustness are evaluated experimentally, under different traffic conditions.

The structure of the thesis is the following:

Chapter 2 provides the background necessary to understand the remainder of this thesis and presents an overview of related work. The chapter focuses on work on QoS control, with a particular emphasis on admission control and resource allocation algorithms, that is fundamental to the models that will be presented in the next chapters.

Chapter 3 proposes a quantitative model of user demand, service provision and admission policy capable of dealing with single and independent jobs subject to response time or waiting time constraints. A $M/M/n/k$ queueing model augmented with some economic parameters is used as a basic building block. The scheme is validated via numerical simulations, shown at the end of the chapter.

Chapter 4 describes the design and implementation of SPIRE (Service Provisioning System for Revenue Enhancement), a service provisioning system that I have developed in order to (i) assess the effectiveness of the policies presented in this dissertation and (ii) studying different possible ways to structure a service provisioning system.

Chapter 5 experimentally evaluates the performance of the SPIRE system: CPU-bound jobs are queued and executed on real servers, while messages are sent and delivered by a real network. The results demonstrate the effectiveness of the adaptive algorithms introduced in Chapter 3 and the autonomic system discussed in Chapter 4.

Chapter 6 describes the issues affecting the single-job model introduced in Chapter 3 (*e.g.*, broken sessions with consequent wastage of system resources), thus it introduces different session-

based admission control schemes. Such algorithms are based on the $GI/G/n$ queueing model; however, since there is no closed form solution for it, this chapter describes an effective approximation that can be used to predict the average waiting times. Some results obtained via simulation are shown at the end of the chapter.

Chapter 7 describes the changes made to SPIRE in order to allow the system to deal with session-based traffic.

Chapter 8 empirically evaluates the effect of the admission policies for service streams introduced in Chapter 6. Again, experiments consist of CPU-bound jobs and involve their execution on real servers and the transit of messages on a real network.

Chapter 9 concludes the thesis suggesting some possible directions for future research and possible extensions to the way commercial data centers can be structured and controlled.

The work presented here was carried out as part of the research project QoSP (Quality of Service Provisioning), funded by British Telecom. It has given rise to the following publications:

- Chapter 3: Jennie Palmer, Isi Mitrani, Michele Mazzucco, Mike Fisher and Paul McKee – *"Optimizing Revenue: Service Provisioning Systems with QoS Contracts"*. In Proceedings of the Second International Conference on E-Business (ICE-B '07), Barcelona, pp 187–191, 2007.

- Chapters 4 and 5 : Michele Mazzucco, Isi Mitrani, Jennie Palmer, Mike Fisher and Paul McKee – *"Web Service Hosting and Revenue Maximization"*. In Proocedings of the Fifth European Conference on Web Services (ECOWS'07), Halle, pp 45–54, 2007.

- Chapters 4 and 7: two European patent applications on behalf of BT.

- Chapter 6: Michele Mazzucco, Isi Mitrani, Mike Fisher and Paul McKee – *"Allocation and Admission Policies for Service Streams"*, In Proocedings of the 16th annual IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008), Baltimore, pp 155–162, 2008 (Best Paper award).

- Chapter 8: Michele Mazzucco, Isi Mitrani, Jennie Palmer, Mike Fisher and Paul McKee, *"Revenue Maximization in Web Service Provision"*, to appear in Computer Science – Research and Development (special issue on Web Services), Springer.

Chapter 2

Background and Related Work

This chapter discusses fundamental related work on service provisioning systems, with a particular emphasis on service provision subject to service quality constraints. This includes papers and solutions for resource allocation and overload protection of servers and data centers (Sections 2.1 and 2.2, respectively), economic models for service provision in Section 2.3, and Cloud Computing systems in Section 2.4.

A service provisioning system – or Internet utility – is a collection of servers connected by high-speed networks. In recent years Internet utilities have become an indispensable component of customer-facing websites and enterprise applications: at their best, provisioning systems allow businesses to outsource their infrastructure and get rid of their IT department and many startups use them to reduce costs.

Internet utilities must keep response time low to satisfy users' performance requirements. However, because when requests arrive concurrently response times grow due to queueing delays [78, chap. 3], in the absence of overload protection algorithms, business critical production application environments are usually over-provisioned to limit the impact of queueing in relation to service times. For example, in 2002 Andrzejak et al. [10] analysed the workload of some commercial data centers and found out that the typical CPU utilisation of servers in Internet utilities ranges between 15% and 35% while, according to Stewart et al. [94], the utilization of any resource in excess of 50% occurs very rarely. This means that, (i) while queueing times should not be ignored, service times usually account for much of the response times, while (ii) the non-queueing congestion effects described by Karlsson et al. [61], like cache interference or contention for shared resources occuring in overload conditions, are likely to happen rarely in practice. As Cockcroft and Walker [33] describe, even in server-consolidation scenarios where the aim is to boost the resource utilization, practitioners tend to keep peak CPU utilization below 70%. Clearly, if one can achieve similar

performance with fewer resources by managing them more efficiently, that would be a desiderable objective.

If we consider a scenario in which each service utility sells the service of executing jobs, *i.e.*, tasks that can be repeated, rather than just raw resources such as CPU time, storage capacity or bandwidth, some sort of middleware – a layer that resides between the operating system and the applications – is usually employed. It (i) creates the illusion of a single system image and (ii) supports functionalities such as resource management, job queueing, scheduling and execution. In such markets, clients and providers negotiate contracts that might include some measure of service quality assurance as well as the price for the service provision. For example, clients may pay more for better service, while providers may incur a penalty if they fail to honour the commitments for agreed service quality.

2.1 Resource Management in Cluster Utilities

Job management systems are software architectures that allow users to run more jobs in less time (or with less resources) by matching the jobs' processing needs with the available resources. Existing implementations such as Condor [103], Load Sharing Facility [84], Portable Batch System [47], IBM LoadLeveler [54] or Sun Grid Engine [96] do not support utility-driven computing because they use algorithms aimed to optimise performance measures such as (maximising) processor throughput, utilisation of the cluster, or (minimising the) average waiting or response time of the submitted jobs. In other words, these solutions do not allow users to 'prioritise' requests in any way, such as through smart scheduling or by allocating more capacity to some jobs.

If one associates a user-specified *utility function* to each task then it becomes possible to model an Internet utility as a utility-driven computing system where resources can be acquired on demand: at job submission, customers can specify the value of the job through Quality of Service parameters, and these parameters can be taken into account by the service provider to decide how many servers to allocate to a certain service, whether to accept a request, or the next request to execute when a resource becomes available. In other words, through QoS requirements, customers are capable of affecting the way such choices are made. No matter what utility function is used, it should be (i) monotonically increasing with the amount of allocated resources, and (ii) fast growing for small amounts of allocated resources, adding more and more resources will yield lower increase in utility. The intuition behind the second point is that once a resource is plentiful then adding additional

capacity would have little or no benefit. For example, if an application can use at most 10 servers, its utility does not change if 12 nodes are allocated to it.

Even though theoretical studies of economic models applied to Internet utilities are only now beginning to emerge (see Section 2.3), several commercial vendors and startups have introduced to the market products embedding economics into their resource allocation systems. Systems like Sun Microsystems' N1 Grid System [15, 97] and IBM's Tivoli Intelligent ThinkDynamic Orchestrator [55] provide dynamic service delivery where users pay only for what they use and thus save from investing heavily on computer facilities. Providers and users agree on Service Level Agreements (SLAs) specifying the expected level of service performance such that service providers are liable to compensate their customers if the level of performance is not satisfactory. Sun Microsystem's product is a grid hosting environment which uses undisclosed policies to provide dynamic and automatic resource provisioning, enabling service level management. IBM's technology is capable of increasing resource utilisation and achieving dynamic infrastructure allocation. The available resources can be allocated and managed dynamically without human intervention through a software layer called Policy-Based Orchestration. The system allows the administrator to define custom policies, while Tivoli automatically deploys and dynamically optimises resources in response to business objectives and conditions.

There is an extensive literature on adaptive resource management techniques for commercial data centers. Yet, the majority of previous work does not take the economic issues related to SLAs into account. As a consequence the service providers would still need to over-provision their data centers in order to address highly variable traffic conditions. Moreover, existing studies do not consider admission policies as a mechanism to protect the data center against overload conditions. However, as it will become clear later in this thesis, admission control algorithms have a significant effect on revenues.

Several studies (see for example [10, 104]) use workload profiling to estimate the resource savings of multiplexing workloads in a shared utility. Such studies focus on the probability of exceeding the agreed performance requirements for various degrees of CPU overbooking. Others, like Doyle et al. [36], vary the degree of overbooking to adapt to load changes, but they do so by considering only average-case QoS within each interval. That paper describes a queueing model for performance prediction of single-tier web servers with static content. This approach (i) explicitly models CPU, memory, and disk bandwidth in the Web server, (ii) uses knowledge of file size and

popularity distributions, and (iii) relates average response time to available resources.

In [85], Rajkumar, Lee, Lehoczky, and Siewiorek consider a resource allocation model for QoS management, where application needs may include timeliness, reliability, security and other non functional requirements. The model is described in terms of a utility function to be maximized and is further extended by Ghosh et al. [41] and Hansen et al. [49]. However although those schemes allow for variation of job computation time and frequency of application requests, once again, congestion and response or waiting time constraints are not considered.

Ranjan and Knightly [87] introduce a suite of algorithms to optimise the performance of dynamic content applications. QuID is a server migration algorithm that allocates servers on-demand within a cluster such that response times are not affected even under sudden overload conditions. As happens in SPIRE, the architecture described in Chapter 4, the system proposed in this paper can be viewed as dynamically migrating servers to and from a shared pool according to the workload and QoS requirements of the hosted application. However, unlike SPIRE, here there is a migration time associated with migrating a server from the shared pool and into a particular cluster tier that, according to some experiments, is in the order of minutes (the same occurs also when machines are released). Apart from the resource allocation scheme the paper deals with the server selection problem. The authors formulate it by assuming a single bottleneck tier in each multi-tiered cluster and modeling it as a $M/G/1$ queue, while the intercluster latencies are the cost of redirection. Two routing algorithms are proposed and evaluated. The first, called WARD, is a server selection mechanism that enables statistical multiplexing of resources across clusters by redirecting requests away from overloaded clusters, with the aim of minimising the response times. Clusters store service time as well as load averages and periodically exchange these values in order to make the 'best' routing decisions. The second routing scheme is a probabilistic redirection algorithm that does not make a decision for each incoming request. Instead, given the cluster workloads and inter-cluster latencies, it computes a fraction of client requests to be forwarded. Finally the paper proposes a policy used to decide which solution to use, namely server migration or request redirection. If the traffic burst is expected to be sustained for a period larger than the migration time, then new servers are migrated locally via QuID. In contrast, if the traffic burst is expected to last for a time-period smaller than the migration time, then migrating new servers is prohibitive and hence requests are redirected to other clusters via WARD. The performance metric considered by the authors is the 95%-ile response time, and the discussed experiments show some performance gains over simpler policies such as round-robin routing. However, as the authors stress, if the proposed algorithms

were employed on a real data center, resource over-provisioning would still be needed because of the lack of an admission control scheme.

The problem of autonomously configuring a computing cluster to satisfy SLA requirements is addressed in several papers. Some of the following papers consider the economic issues occurring when services are offered as part of a contract, however they do not address the problems affecting overloaded server systems, while others include some simple admission control schemes without taking any economic parameter into account when the system configuration needs to be updated.

Li et al. [67] propose a strategy for autonomic computing that divides the problem into different phases, Monitor, Analyze, Plan and Execute (MAPE, according to the terminology used by IBM [63]) in order to meet SLA requirements in terms of response time and server utilization. The system can be reconfigured in two ways, (i) by triggering an event when a SLA violation occurs, or (ii) proactively, every time a resource threshold is reached (both lower and upper bounds can be specified). However, unlike the work presented in this dissertation, the policy decisions involve the reconfiguration of resource allocation to services only; they do not attempt to control admissions.

Menascé et al. [77] propose an approach based on hill climbing techniques combined with analytic queueing models to guide the search for the best combination of configuration parameters of a multilayered architecture hosting a E-Commerce applications (*i.e.*, all resources of web, application and database servers are 'optimised'). The authors consider as QoS metric the average response time, the throughput and (this is indeed the main difference compared to previous work) the probability that a request is rejected. The system includes a very simple admission control scheme that uses a fixed threshold whose value depends on the number of available threads. The system collects usage and performance statistic values during a period of time, then it updates the configuration for the next interval accordingly. A problem of this solution is that it might happen that a request is served by the web server but rejected by an overloaded application server, while a smarter approach would reject the request at the front-end. Moreover, even though the shown experiments demonstrate the usefullness of the proposed approach in order to improve the average response time under heavy load, the paper does not consider the economic issues associated to enterprise service provisioning.

Walsh et al. [116] discuss a distributed architecture consisting of multiple interacting autonomic elements with the aim of solving the resource allocation problem for a dedicated data center architecture with dynamic virtual pools, like the one discussed in this dissertation. The system is

composed of different modules, including a data aggregator to store traffic related information, a demand forecaster module, an utility calculator, and a controller to decide how many resources to allocate to each virtual pool. The emphasis of these papers is on the use of utility functions as fundamental framework to automatically optimise resource usage rather than utility functions as a foundation to optimise the utility of a data center. Similarly, Bennani and Menascé [16] integrate the techniques first introduced by Menascé et al. [77] to a multiclass queueing network model to maximize an utility function that takes into account response time and throughput values. However, even though the authors mention SLAs, no charges, penalties nor any other economic value is taken into account. The problem is defined as a two-class system, where the first type of requests are online transactions while the second workload is made by batch jobs. The intensity of the first type is given by the average arrival rate, while the performance metric of interest is the average response time, while the intensity of the batch jobs is given by the concurrency level (*i.e.*, the average number of concurrent requests in execution), while the function to maximize is the average throughput. The system is composed by a workload monitor module, that collects the traffic statistics, a workload forecaster, that uses the historical values to predict future workload intensity values, and an utility function evaluator. The number of servers to allocate to each class is computed using a combinatorial search technique over the space of all possible configurations (the evaluator computes the utility function provided by each configuration).

Zhang and Ardagna [124] focus on the design of a resource allocation algorithm for shared data centers subject to QoS constraints and multi-class SLAs specifying penalties in case the service provider fails to deliver the promised service quality. The proposed scheme aims at maximising the profit by using an utility function that takes into account, apart from revenues for executing client requests and penalties for SLAs violations, also the cost to keep servers running. This work differs from existing one because, at some point, the policy might decide (depending on the system load) to switch some servers off in order to reduce the cost factor.

Liu et al. [69] propose a theoretical model that uses both load balancing and server scheduling when trying to maximize the profit of a hosting platform subject to multi-class SLAs. As in [124] there are charges for executing requests and penalties for performance quality violations, but here the performance metrics include the tail distributions of the per-class delays in addition to per-class throughputs and mean delays. The analysis assumes that servers can serve different types of requests at the same time, using a generalised processor sharing (GPS) scheduling policy, while the request routing is probabilistic to the sub-set of servers allocated to the class.

Chandra et al. [28] present a GPS based queueing model of a single resource, such as the CPU, at a web server. The model is parametrized by online measurements and is used to determine the resource allocation needed to meet desired average response time targets. Prediction and adaptation algorithms dynamically partitions the GPS-resource to the running applications according to the estimated workload. The paper presents some experiments showing the effectiveness of the proposed approach, however the framework lacks of an scheme capable to prevent overload, while SLAs are not considered.

Villela et al. [110], Urgaonkar et al. [105] and Levy et al. [66] consider an economic model similar to the one proposed in Chapter 3. There, the emphasis is on allocating server capacity only, while admission policies are not taken into account. Yet, as it will become clearer in the following chapters, revenues can be improved very significantly by imposing suitable conditions for accepting jobs.

Urgaonkar et al. [105] model analytically the behaviour of multi-tier architectures where a request can visit each tier multiple times while the SLA is specified in terms of average response time. The architecture is modeled as a network of queues, where each queue represents a tier, while the scheduling discipline is assumed to be processor sharing (PS). Active sessions are modeled using an infinite server queueing system that feeds the network of queues, thus forming a closed-queueing system. Average response time values are estimated using the mean-value analysis (MVA) algorithm for closed queueing networks. Monitoring the average service times, the average think time of a session (the average interarrival intervals between requests belonging to the same session), the number of concurrent sessions and estimating the visit ratios for the queues, the algorithm computes the average response time. Such algorithm is based on the queueing theory result that in product form closed queueing networks, when a request moves from queue Q_i to another queue Q_j, it sees, at the time of its arrival at Q_j, a system with the same state but with one less customer. One interesting feature of this model is, unlike Menascé et al. [77], the ability to handle concurrency limits. However, the model does not include any overload scheme, and thus it might happen that only some requests belonging to a session are effectively processed.

Villela et al. [110] study how a service provider should allocate the application tier of an E-commerce application subject to QoS constraints. The SLA is the same as the one used in Chapter 3 as it uses charges for executing jobs and penalties for failing the agreed service quality commitments. The paper models each server as an $M/G/1/PS$ queue, while the objective is to minimise the cost that results from SLAs violations (no admission control mechanism is employed). So the

authors assume that servers split their processing power evenly between their simultaneously active jobs using a round-robin scheduling mechanism. Thus, the time spent servicing each job depends heavily on the number of jobs running into the same server. The processor sharing model employed in this paper works well in theory as it is based on the assumption that the quantum each job receives is 'very small'. However, in practice the quanta can not be too short, otherwise the system overhead caused by task switches becomes excessively high. Moreover, it is not fixed, but it varies at runtime and from job to job, while the scheduling policy is not a 'perfect' round robin, but it takes into account priorities as well. Finally, because the authors assume a round-robin scheduling policy where all jobs receive the same amount of time to run, it is not possible to implement any form of service differentiation mechanism such as giving more time to more 'important' jobs. The paper proposes and experiments with three allocation heuristics that forbid different customers from sharing the same server and whose computational complexity is linear in the number of customers. The first algorithm assumes that the average service time of jobs for each customer is known, the second approximation requires both the first and second moments of the service time distribution, while the third algorithm assumes known bounds for the class of exponential bounded burstiness processes, to which the Poisson process belongs. The proposed policies are compared against two simple heuristics,a 'uniform' that allocates servers evenly between customer and a 'naive' one that shares the available machines in proportion to a customer arrival rate times the charge per request, divided by the tolerated response time, and appear to perform better.

Levy et al. [66] present an architecture and prototype implementation of a performance management system that supports multiple classes and dynamically allocates server resources, balances the load according of the offered user demand in order to support SLAs, that are expressed in terms of average response time. The peculiarity of the paper is that the cluster utility function is in fact a composition of two functions: a class specific utility function, for each class of requests, and a combining function that combines the class utility values into one cluster utility value. The former represent the utility from the customers' point of view (*i.e.*, customer satisfaction), while the latter represents the utility from the provider's perspective (usually profit).

2.2 Overload Protection Schemes for Service Provisioning Systems

Overload control is a very well researched topic in both telecommunications and web servers systems. This section describes two techniques that can be useful when trying to prevent a server or

collection of servers from being overloaded. Section 2.2.1 presents previous work on admission control, a technique that explicitly discards part of the client requests as a form of overload protection, while Section 2.2.2 describes some service differentiation schemes, a technique that classifies customers and delivers the best service to some of them at the expenses of the others.

2.2.1 Admission Control

As servers approach saturation response times grow noticeably, providers incur penalties (if SLAs are in operation) and customers become unhappy. When the request rate on a server increases beyond server capacity, the server becomes unresponsive. The TCP listen queue of the server's socket overflows, exhibiting a drop-tail behaviour. As a result, clients experience service outages. Admission control is a technique used to prevent systems from becoming overloaded: rather than forcing all clients to experience unacceptable performance level, admission control schemes explicitly discard a fraction of the client requests. There is an extensive literature on single-jobs admission control: this section reviews the most important ones. Some papers use SLA parameters when deciding whether to admit a client request or not, others don't. However, I am not aware of any study using admission control algorithms, resource allocation policies and economic parameters to improve the efficiency of computer systems.

Welsh et al. [118] present one solution to overcome the scalability issues related to the enterprise applications that must support a mix of fast and long running business processes, or with great or small throughput. The Stage Event Driven Architecture (SEDA) model is an architecture style that decomposes system services into a network of stages, where each stage performs part of the request. Stages are separated by dynamic resource controllers to allow applications to adjust dynamically to changing load: at each stage a monitor measures the response time and a controller decides whether to admit a given event to that particular service. If the event is not admitted, custom actions can be executed, *i.e.*, the user's request is not systematically rejected but, for instance, part of it can be redirected to another node. The fine-grained admission control mechanism enables the monitoring of single resources and allows the system to reject only those events leading to bottlenecks.

Zhou and Yang [125] present Selective Early Request Termination, a load shredding mechanism that can be used in systems that use threads or processes to handle multiple requests concurrently (this is the case of popular web servers such as Apache or Microsoft IIS and several application servers). The algorithm is based on the assumptions that (i) it is not possible to classify requests and that (ii) a relatively small percentage of long requests require excessive computing resources

dramatically affect other short requests and reduce the overall system throughput. The idea behind the model is to use an an adaptive timeout threshold, *i.e.*, a timeout varies according to the loading conditions: the system monitors all active requests and aborts those jobs that have not completed once their threshold has expired.

Elnikety et al. [39] present Gatekeeper, a proxy-based approach for admission control and resource scheduling. The proposed framework externally observes execution costs of requests online, distinguishing different request types, and performs overload protection and preferential scheduling using dynamic estimates of the loading conditions, while the admission control mechanism keeps track of the estimated current load and admits a new request only if the new load does not exceed the server's capacity. The maximum server capacity is computed off-line using a brute force approach that assumes that the system's throughput is a concave function, *i.e.* a function that grows up to a certain level and then decreases.

Urgaonkar et al. [105] present Cataclysm, a server platform designed with the aim to handle extreme overloads in hosted Internet applications. Like the framework that will be described in this dissertation, Cataclysm tries to maximize the average revenue achieved per unit time, but during traffic surges it does so by using a combination of three techniques, (i) adding server capacity, (ii) performance degradation (see next section) and (ii) admission control (when the other two approaches are not possible or are not enough). Cataclysm uses an architecture similar to the one described in Chapter 4, while the threshold-based admission control heuristic uses a $G/G/1$ queueing model. The SLA is formulated in terms of maximum response time related to the arrival rate, *i.e.*, the response time is allowed to degrade within a certain value if the arrival rate increases above a certain threshold. The model uses charges for accepting and executing a request with the desired performance quality, however there are no penalties for violating the SLA.

Mahabhashyam and Gautam [71] discuss a dynamic protocol employing both a resource allocation and an admission control algorithm for shared data centers service multiple classes of requests having different QoS requirements. The model uses two classes: the first consists of real time traffic that has bandwidth requirements such as audio or video streaming, while the other is composed by 'elastic' traffic, *i.e.*, requests that use the processor capacity not used by class 1 tasks. This model differentiates from the one introduced in Chapter 3 because it (i) assumes that servers are shared between customers' requests, (ii) only type 1 requests can be rejected, and (iii) the SLA includes a penalty for each rejected class 1 job and a penalty proportional to the delay for class 2 requests (though, it is not clear whether there are penalties for delaying type 1 jobs). Because of

the different requirements, the authors analyze the two classes separately. For type 1 jobs, if the exponentiality assumptions hold, the process $\{X_k(t), t \geq 0\}$, where $X_k(t)$ represents the number of requests of type 1 in the system at time t, can be modeled as a continuous-time Markov chain. From a queueing perspective this is a $M/M/n_k/n_k$ system where the loss probability can be computed using the Erlang B formula. The analysis of the average delay of type 2 jobs is completely different because it must take into account type 1 requests into the system as well. The stochastic process is a Quasi-birth-process that can be solved using a technique called matrix geometric method. The paper shows only some numerical results, while in the final section the authors state that, if their approach is going to be implemented in a real system, some sort of automatic technique like the ones discussed in this dissertation should be used to monitor and estimate the arrival rate and service time values.

While there is an extensive literature on request-based admission control, session based admission control is much less well studied (sessions are sequences of individual requests made by the same client and needed to complete a task). The studies described here relate to the model proposed in Chapter 6. However once again nobody has studied the effects of combining admission control, resource allocation and economics when trying to model a commercial service provisioning system subject to QoS constraints.

In [32], Cherkasova and Phaal observe that, by default, overloaded web servers discriminate against longer sessions. Moreover, an overloaded web server can experience a severe relative loss of throughput measured as the number of completed sessions per second, compared to the relative loss of requests per second. The approach proposed by Cherkasova and Phaal tries to guarantee a fair probability of completion for any accepted session, no matter of its length, because their predictive admission control strategy accepts a new session only if the server has the capacity to successfully complete all the related requests. Instead, if the server is close to being overloaded, the entire session is rejected. Request are classified into high, medium and low priority, and priority levels are used to determine admission priority and performance levels. Estimation of the server capacity and evaluation of the load generated by the sessions is computed using a basic queueing network simulation model. The algorithm described in this paper has been integrated in HP's WebQoS product [51].

Bartolini et al. [14] present a probabilistic admission control scheme that filters incoming sessions according to an adaptive rate limit. This value is computed by analysing the relationship

between the rate of admitted sessions and the corresponding measure of response time. The described model is essentially a static threshold scheme, where the threshold value is periodically re-computed. The algorithm can work in two modes, normal and flash crowd respectively, switching from one to the other according to the measured traffic conditions. The time between two subsequent updates of the admission control policy becomes increasingly shorter as traffic changes become sudden and fast, as in presence of flash crowds. This interval is set back to longer values when the workload conditions return to normality. SPIRE, instead, uses events to divide configuration intervals, a more elegant approach that has the great advantage of automatically distinguishing between 'idle' and 'busy' periods. In Bartolini et al.'s model the transition between normal and flash crowd mode occurs every time a steep change in the arrival rate is detected, while once in surge mode, the system returns to normal mode only when the admission probability has been properly adapted to ensure that the instantly measured session admission rate is below the threshold value. The SLA discussed in this paper includes three parameters, (i) the maximum acceptable value of the 95%-ile response time for each request type, (ii) the minimum guaranteed admission rate, and (iii) the observation interval between two consecutive checks of the satisfaction of the performance levels specified into the SLA. Thus, the admission control algorithm does not take any economic value into account when taking admission decisions.

Carlström and Rom [26] propose a revenue maximisation scheme based on a GPS model where the maximum number of active sessions is bound by a threshold. The admission control scheme is rate based, that is, it limits the rate at which new sessions are accepted. Once accepted, sessions are classified according to a fixed set of stages (*i.e.*, welcome, browsing with empty cart, browsing with some articles in cart, and checkout): each stage has its own FIFO queue while, as mentioned earlier, the scheduling algorithm is assumed to emulate GPS. The authors measure the QoS in terms of a *patience function*, thus the goal is to keep delay low for sessions expected to generate high reward; in case of overload conditions, sessions in stages less likely to generate any revenue (*i.e.*, sessions in 'welcome' stage) are given lower priority. Menascé et al. [76] consider a similar problem of resource scheduling (but without admission control) in e-commerce sites with the aim of maximizing revenue. The authors suggest a priority scheme for scheduling requests based on the states (navigation and purchase) of the users, assuming that users will lose patience and thus leave the system if the response time is too long.

Finally, Guitart et al. [48] propose an overload management scheme based on SSL (Secure Socket Layer) connection differentiation and admission control. This model distinguishes from

other studies mainly because of the SSL connection differentiation: the CPU time needed to establish a new SSL connection is much greater than establishing a resumed SSL connection (it reuses an existing SSL session on the server). Considering the difference, the proposed admission control algorithm (eDragon Admission Control) prioritises resumed SSL connections to maximize the performance in session-based environments and dynamically limits the number of new SSL connections accepted, according to the available resources and the current number of connections in the system, in order to avoid server overload. The admission control assumes the time needed to establish new SSL sessions or to resume existing ones to be known (those values are measured offline), then it periodically computes the number of new SSL connections that the server can accept during the next configuration interval without being overloaded.

2.2.2 Service Differentiation

Service differentiation can be considered as a special case of admission control, discriminating one set of customers against others. The core idea is to distinguish between different classes of customers – for example, at bronze, silver, and gold levels with increasingly better response times – and/or the context in which the service is provided (*i.e.*, users in the check-out phase are given higher priority over users browsing the available items), so that the high priority class receives a preferred service. In this way limited resources can be effectively used, to a certain extent even under overload conditions. The approach adopted in Chapter 4 is that the same service could be instantiated at different service levels. Each differentiated level will have its own SLA management function instantiated that strives to meet that level of service specified by the differentiation. If the load is too high for any of the differentiated services, then the admission policy will start rejecting requests in order to maintain an adequate level of performance.

The use of economic parameters when taking decisions is an elegant way to service provision at different quality levels: for example, the resource allocation algorithms described in Section 3.3 automatically allocate more resources to more expensive jobs, while the admission control module protects the data center against possible penalties deriving from shortage of servers allocated to 'cheap' queues. Even though the approaches reviewed in this section can be generalized, the majority of them only consider the problem of how to provide QoS support to single servers (and thus the service differentiation is achieved through mechanisms such as priority schedulers or queues), while they do not consider charges and penalties.

Bhatti and Friedrich [18] propose an architecture where incoming requests are classified and

reordered based on scheduling policies, then submitted to the server for normal processing. For instance, requests from premium users are given preferential processing treatment, and are thus executed as high priority processes by the host operating system. The model allows one to apply different service-level policies to each category. For example, if the server is experiencing heavy traffic, low priority requests can be redirected to an alternative server or receive rejection notices.

Kang et al. [60] present a real-time database QoS management framework for E-business applications providing performance guarantees in terms of differentiated deadline miss ratio and data freshness. User requests are classified into classes according to their importance, and each class receives different guarantees on its miss ratio to support real-time data services. In case of overload surges, the system provides preferred services to the high priority classes.

Another example is the work by Lu et al. [70], that achieves differentiated delay guarantees by dynamic connection scheduling. Significant connection delay arises when all server processes are tied up with existing connections, in which case new connection requests wait in the TCP listen queue to be accepted by a server process. A connection scheduler component allocates server processes to waiting connections according to priority of the class. Based on a theoretical model, the scheduler controls that 'process budgets' of all classes are not overwhelmed.

Finally, Xu et al. [121] present eQoS, an end-to-end QoS provisioning framework for web servers that monitors and controls user perceived service quality in real-time. eQoS is composed by two separate modules, a QoS monitor and a resource manager. The former considers network transfer time, server-side queueing delayes as well as processing time by capturing live network packets from the network. The latter comprises a classifier (to classify client requests and thus facilitates resource allocation between classes – categorisation is made according to rules defined by the service provider), a waiting queue for each request type and a process-rate allocator. The rate allocator treats each request as a scheduling unit and assigns a (dynamic) weight to each class (this allows eQoS to process requests according to their priority). The described framework has been implemented as a module for the Apache web server (Apache handles each request using a separate process), and thus the rate allocator simply controls the number of child processes allocated to each queue. In order to predict how a certain allocation will affect the response time the authors use a $M/G_p/1$ queueing model.

If the service is provided according to a best-effort mode, or if it is possible to re-negotiate the SLAs in force, then it is possible to use an overload control mechanism based on content adaptation

that does not reject any request (a similar approach was adopted by CNN [65] during the reporting of the September 11, 2001 attack in New York). In case of web sites, for example, content adaptation is used as a technique to degrade web page content while preserving essential information and reducing the resource requirements needed to deliver them. Different algorithms can be employed, depending on the structure of the web site. Images tend to account for a high bandwidth, and so they are a suitable target of a content adapting strategy, while links can be removed in order to reduce the number of pages.

The content adaptation model introduced by Abdelzaher and Bhatti [5] uses pre-generated versions of the web site. The algorithm automatically switches between the versions depending on server 'load', where the load takes into account not only the CPU utilization, but other values such as the used bandwidth too.

Finally, Andersson et al. [9] propose a content adaptation scheme that decides which page to return based on an utility function. The problem is expressed in terms of an optimisation problem whose objective is to maximize the total utility subject to cost and budget constraints.

2.3 Economic Models for Service Provisioning Systems

The most complex information system ever conceived, the Internet, takes advantage of economic ideas. While the Internet is based upon a best effort service model, supply-side economics are reflected by overprovisioning, namely an excess bandwidth, in order to ensure some form of service quality. The approaches described in this section make use of economic ideas such as dynamic pricing structures to model the service provision problem. However, they do not take congestion into account explicitly and are therefore not readily applicable in the context under study.

Huberman et al. [53] and Sallé and Bartolini [90] present two different abstract frameworks involving pricing and contracts in IT systems. The former describes a pricing structure for the provision of services with the aim of ensuring trust without requiring repeated interactions (*i.e.*, the reputation of the service provider does not matter) between providers and consumers. The model is based on the notion of 'trust', and thus the SLA is formulated as the likelihood that the service providers delivers what they commit to. The idea is to use a pricing structure with rewards and compensations in order to force the parties to make accurate assessments of their ability to do what they promise. Using the same approach, the authors also try to solve the overbooking problem in reservation, that is the problem that occurs when a provider promises more than he is able to do.

The *Management by Contract* scheme proposed by Sallé and Bartolini [90] is completely different. The aim of this paradigm is to formalize and analyze contractual relationships between service providers and consumers in order to take better decisions. The authors abstract IT management into a model composed by three layers, monitoring, diagnosis and recovery planning. The first one probes the state of the system and triggers alarms in case of failures or service degradation. The second layer identifies the root of the problem, while the third layer determines possible recovery plans and associated costs. Sallé and Bartolini's idea is to add a fourth layer to the existing model, called 'contract-based analysis'. Such module uses an utility function and the SLAs in operation to decide which of the options proposed by the recovery planning layer should be adopted.

Bai et al. [13] study a macroeconomic model for resource allocation in large-scaled distributed systems where resources belong to different organizations. The study defines a measure of 'consumer's satisfaction' that takes into account the utility resulting from resource consumption and the price paid by the client. Such measure is used instead of the QoS parameters because, the authors state, the satisfaction and the utility of the customer are not the same, but behave differently under different pricing strategies. The model uses a broker not only to ease access to resources, but also to reconcile the selfish objectives of each side with some global objectives such as maximizing the resource utilization. Instead of using an explicit admission control mechanism, the paper examines the effects of the pricing policy on the system's utility and users' satisfaction: intuitively, if the price grows, fewer customers are willing to pay for the execution of their jobs, and vice versa. Three pricing policies are experimented via simulations: (i) linear, no matter what the resource usage is, (ii) sub-linear, used to encourage consumption (the more resources are employed, the lower the average unit price becomes), and (iii) super-linear, that should be employed if a provider wants to discourage consumption (the more resources are used, the higher the average unit price becomes). Finally, the paper analyses the effect of resource abundance upon pricing strategies, *i.e.*, the price changes dynamically according to loading conditions. The experiment shows that, in some circumstances, the solution giving the highest satisfaction to the customer is neither the one which offers the largest amount of resources, nor the cheapest one.

Amir et al. [8] propose a cost-benefit framework inspired by economic principles to allocate resources and to route requests in an Internet utility. The system is examined in terms of benefits and costs, while the goal is to maximize the data center's utility and to minimize its cost. The paper uses a processor sharing model and the main hypothesis is that the cost of a resource is an exponential function of its utilization, *e.g.*, each time 10% of the CPU is used, its cost automatically doubles.

This assumption implies that it is better to execute a request on an idle machine rather than running it on a heavily loaded server. This protocol can be extended to be used as a basic throttling algorithm too: an admission control module using it would admit a new request into the system only if the revenue earned by executing it is higher that its cost.

Yeo and Buyya [122] propose a protocol in which the service provider carries out a risk analysis to analyze the effectiveness of resource management policies in achieving the following objectives: (i) meet SLAs, (ii) maintain the reliability of the provided services, and thus earn profit. The authors emphasize that maintaining the reject rate at acceptable levels is as important as delivering the promised level of performance because rejecting too many requests makes customers unhappy and causes them to switch to other competitors. The proposed model is quite general, and in fact it is experimented against several resource management policies (some including an admission control scheme too), some using a dedicated model (*i.e.*, at most one job can run on a processor at any given time), others using a processor sharing approach.

Kalé et al. [59] present Facucets, a framework aiming at providing a market-driven selection of resources, resulting in a more efficient utilization of servers. In order to do so, the paper uses (i) 'adaptive jobs', *i.e.*, jobs that can change the number of processor allocated to them on demand, (ii) 'smart job schedulers', that dynamically change the amount of resources allocated to each parallel job, and (iii) QoS parameters to facilitate server selection. Such components are used to maximize a utility function, which is evaluated every time schedulers take allocation decisions. Metrics to optimize include system utilization, average response time, or more complex payoff functions which may specify premiums for early completion and penalties for delays beyond deadlines. Finally, a bidding system is used to take both scheduling and routing decisions: jobs compete with each others with the aim to create market efficiencies. The bidding decision can be based on both local or non-local factors. Examples of the former is the load of the server(s) during the time-period covered by the job, or how far into the future the deadline is, while non-local factors might include the average price of similar contracts in the whole system.

Chen et al. [31] propose different pricing strategies for automatically tuning a server according to the measured loading conditions. The SLA includes charges for executing requests (which can be fixed or vary, depending on the policy in operation) and penalties proportional to the delay. The paper proposes three dynamic strategies. The first heuristic is a static pricing algorithm combined with an admission control mechanism based on queue-length thresholds. Such an approach is similar to the one proposed in Chapter 3 except that the maximum number of requests to admit into the

system is computed off-line, using a simple trial-and-error algorithm: the lack of a more sophisticated algorithm limits its usefullness in a real scenario. The second algorithm, a dynamic optimal pricing with no admission control, does not reject any job. Instead, it varies the price according to the arrival rate in order to find the 'optimal' arrival rate, *i.e.*, the one that maximizes the revenue per unit time. The idea behind this algorithm is that, as the price grows, less and less customers are willing to submit their jobs. The third heuristic proposed by the authors is a static pricing policy with non-negative profit-based admission control. The algorithm is not described in great detail, but from the available information the 'optimal' static price is obtained through a brute-force algorithm while a request is dropped whenever the revenue from that customer is negative (this implies that the server must maintain a state for each customer).

Finally, POPCORN [88] provides a paradigm for distributed computation across the Internet. The most interesting part of the framework is the marked-based mechanism to sell and buy CPU time, based on a trusted intermediary in charge of matching buyers and sellers according to some economic criteria. The way CPU resources are sold depends on the policy in operation; the authors propose and experiment with three different mechanisms. In the first one, a Vickrey auction [107], the CPU-time of the seller is auctioned among different buyers and the price to pay is the second highest price offered in the auction. The second is a simple sealed-bid double auction [40]: both sides offer a low, a high price, and a rate of change, and the auction ends when the buyer's price 'meets' a seller's one. The third algorithm is a repeated Clearinghouse double auction [40], which differs from the previous auction scheme only because in each round more than one buyer-seller pair is matched. At fixed-time intervals such values are used to compute the demand and supply curves and the equilibrium price, which is used in the following round. No matter what auction scheme is employed, the simulations confirm some intuitive results: (i) an increase in the supply of computing resources results in a decrease of the average price, and vice versa, (ii) a lower ask price results in a higher probability of selling, and (iii) buy offers lower than a certain threshold and sell offers higher than a certain value are never executed. This model does not employ any allocation nor admission control schemes. Instead, the utility improvements shown by the experiments are achieved exclusively by changing the economic parameters.

2.4 Cloud Computing

Even though the POPCORN framework does not take any performance constraints into account, it is of great importance because, like other distributed supercomputing infrastructures such as GPUGRID.net [44] or SETI@home [102], it is the precursor of what is known as *Cloud Computing* [22], a paradigm for the provision of computing infrastructure that combines *Software as a Service* (SaaS), *i.e.*, applications built using other services, Web 2.0 and other recent Internet trends such as resource virtualization.

Cloud Computing is often described as the easy and cheap way to achieve horizontal scalability (*i.e.*, adding more resources and making them work as a single unit) without understanding or being responsible for the IT infrastructure, depending on software as a service. This paradigm is not the same as Grid Computing, which is simply a cluster of loosely connected computers that distribute tasks among themselves (see [106] for a detailed comparison): instead, Cloud Computing is a natural next step from Grid Computing. Many Cloud Computing deployments are today powered by grids, but have autonomic characteristics and are billed like utilities. Indeed, one of the main differences between the two paradigms is the way resources are allocated: Grid Computing is used in environments where users make few but large allocation requests, and this results in sophisticated batch job scheduling algorithms of parallel computations. Cloud Computing is poles apart as it is all about lots of small allocation requests: computing resources are allocated in real-time and in fact there is no provision for queueing allocations until someone else releases some servers. This is a completely different resource allocation paradigm, a completely different usage pattern, and all this results in completely different method of using compute resources. The core idea behind Cloud Computing is *auto-scaling*, the ability to (i) monitor the user demand in real-time and (ii) automatically react by adding or removing computing resources accordingly, without any human intervention (most cloud providers are indeed still experimenting with this technology, and thus do not offer this kind of service yet).

Both academic and industrial organizations are investigating and developing technologies and infrastructures to exploit the cloud. Academic efforts include Virtual Workspaces [62] and OpenNebula [80]. Such solutions provide isolation guarantees and resource reservation for running applications through the use of virtual machines, but none of them consider the QoS issues related with service provisioning.

Several Web giants including Google, Microsoft and Amazon as well as start-up companies

have started providing Cloud Computing-related products. For example, Amazon Elastic Compute Cloud (EC2) [2] is a web service that provides resizable compute capacity in the cloud. EC2 offers a virtual computing environment enabling a user to run Linux-based applications. The user can either create a new Amazon Machine Image (AMI) containing the applications, libraries, data and associated configuration settings, or select from a library of globally available AMIs. Once the AMI is in place, the user can start, stop, and monitor instances of the uploaded AMIs. EC2 charges the user only for the resources that (s)he actually consumes (like instance-hours or data transfer), while Amazon Simple Storage Service (S3) charges for any data transfer (both upload and download).

2.4.1 Issues in Cloud Computing

Cloud vendors achieve 'extreme' scalability by assuming unlimited resource availability and using auto-scaling algorithms. Indeed, auto-scaling provides only a brute-force form of load-balancing, but although such technique can help considerably in alleviating overloads caused by the highly variable nature of the web traffic, it is important to stress that it cannot replace proper overload control: a good overload management scheme is essential for commercial servers, even with load-balancing, while for small web-servers (*e.g.*, single node) where load-balancing does not apply, or when not all servers can handle all web-pages, admission control becomes essential. Also, it must be noted that, even if auto-scaling is used, cloud providers might not be able to respond fast enough to increased capacity needs. For example, once the need for extra capacity has been detected, Amazon EC2 can take up to 10 minutes to launch a new instance. Customer's applications are completely unprotected, as there is no load balancing (all servers have already reached the saturation point) nor overload control in place, and thus it is 10 minutes of impaired performance, or perhaps even 10 minutes (at least) of downtime.

A closer look at the SLAs provided by most vendors shows that cloud providers guarantee almost nothing [1]. The main reason why cloud vendors do not offer strong SLAs is that the cost advantage of the cloud is based on shared resources. Although IBM is now pushing the idea of creating and running 'private' clouds for its customers [3], *i.e.*, clouds that are managed within the organization, the fact that many applications are running on a shared infrastructure increases the risk of catastrophe. While there is no current solution for the provision of performance guarantees on cloud infrastructures, the availability issues can be, if not entirely solved, at least mitigated with the adoption of two replication techniques. The first approach (multi-cloud) simply replicates the business infrastructure across multiple cloud providers, while the second is more complicated

as companies should backup their systems on a data center they control. If the cloud fails, they can move operations temporarily to their own backup, which may have all of the data needed or, depending on the application, a cache of the most active data.

Chapter 3

Allocation and Admission Policies for Single Jobs

This chapter presents a QoS model that can be employed in a service provisioning system where a cluster of servers is employed to offer different services to a community of users. The immediate motivation comes from the world of Web Services, but other multi–class hosting environments would fall in the same framework. With each service type is associated an SLA, formalizing the obligations of the users and the provider. In particular, a user agrees to pay a certain amount for each accepted and completed job, while the provider agrees to pay a penalty whenever the response time (or waiting time) of a job exceeds a certain bound. It is then the provider's responsibility to decide how to allocate the available resources, and when to accept jobs, in order to make the system as profitable as possible. Clearly, efficient policies that avoid over-provisioning are desirable.

The aim of this chapter is to propose and evaluate efficient and easily implementable policies for resource allocation and job admission. In more detail, this chapter will try to tackle the following issues:

1. Given a number of service types and a total number of available servers, together with the set of different demand and QoS contract parameters, how many servers should be allocated to each service type?

2. Given the number of servers allocated to a particular type of service, together with the corresponding demand and QoS contract parameters, what is the optimal queue length threshold, beyond which incoming jobs would not be accepted?

3. What is the effect of dependencies between the economic parameters? For example, it may be reasonable to charge higher prices for executing jobs whose contractual response time

bounds are lower. How does the form of that dependency affect the optimal server allocation and admission policy?

The used approach includes mathematical analysis, numerical simulations (in this chapter) and the design, implementation and performance assessment of a real hosting environment (next chapter). This model will be extended in Chapter 6 in order to deal with stream of requests.

3.1 The Model

Today's service provisioning systems are usually designed according to a three-tier software architecture. The first one translates end-user markup languages such as HTML or XML into and out of business data structures, the second tier (*i.e.*, the business logic tier) performs computation on business data structures while the third level provides storage functionalities. Requests traverse tiers via synchronous communication over local area networks and a single request may revisit tiers more than once. Business-logic computation is often the bottleneck for Internet services [110], and thus this chapter focuses on this tier. However, user-perceived performance depends also on disk and network workloads at other tiers. Front-end servers are not typically subject to a very high workload, and thus over-provision is usually the cheaper solution to meet service quality requirements. Moreover, different solutions exist to address some of the issues occurring at both the presentation and database tiers [37, 93], while McWherter et al. [73] have shown that smart scheduling can improve the performance of the database tier.

The model for the application tier, depicted in Figure 3.1, consists of N identical servers, which may be used to serve jobs belonging to m different types. Job types, which may for example include locally cached web accesses as well as complex workflows involving distributed transactions, have different QoS requirements (*i.e.*, some may be less tolerant of delays than others). Once allocated to a type of service, a server remains dedicated to jobs of that type only, until a subsequent re-allocation. In other words, a non-sharing server allocation policy is employed: n_i servers are assigned to jobs of type i ($n_1 + n_2 + \cdots + n_m = N$). Such a policy may deliberately take the decision to deny service to one or more job types (this will certainly happen if the number of services exceeds the number of servers, that is, if $m > N$).

Each server can execute only one job at any time, *i.e.* the system does not allow processor sharing. Jobs of type i are either waiting for service at queue i, or being served. The scheduling policy is FIFO, with no preemption, while servers allocated to queue i cannot be idle if there are

jobs of type i waiting. Jobs of type i arrive at rate λ_i, while required service times have mean $1/\mu_i$. Finally, an admission policy controlled by a set of thresholds is in operation: if there are K_i jobs of type i present in the system (waiting and in service), then incoming type i jobs are not accepted and are lost ($i = 1, 2, \ldots, m$).

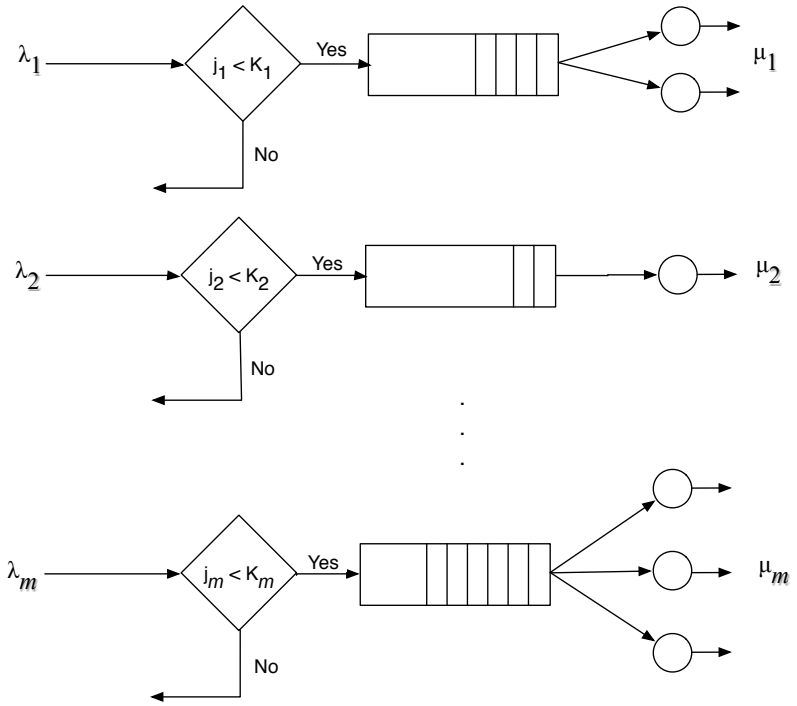


Figure 3.1: *QoS model for single jobs.*

For the purposes of this model, the quality of service experienced by an accepted job is measured either in terms of its response time, W (the interval between the job's arrival and completion), or in terms of its waiting time, w (excluding the service time). Whatever the chosen measure, it is mentioned explicitly in a service level agreement between the provider and the users. We assume that each such contract would include three clauses, namely charge, obligation and penalty.

Definition 3.1 (Charge)**.** *For each accepted and completed job of type i a user shall pay a charge of c_i. In practice this may be proportional to the average length of type i jobs or it may be related to the obligation q_i.*

Definition 3.2 (Obligation)**.** *The response time, W_i (or waiting time, w_i), of an accepted job of type*

i shall not exceed q_i.

Definition 3.3 (Penalty)**.** *For each accepted job of type i whose response time (or waiting time) exceeds q_i, the provider shall pay to the user a penalty of r_i.*

Such value penalizes the service provider to compensate the user for failure to meet the promised level of service. It is also a valuable input to the customer service selection process because a higher level of penalty may be indicative of the level of provisioning by the vendor and indicate suitability for more business critical functions.

In other words, in this model service type i is characterized by its *demand parameters*:

$$(\lambda_i, \mu_i),\ i = 1, 2, \ldots, m \tag{3.1}$$

and its *economic parameters*:

$$(c_i, q_i, r_i) = (charge, obligation, penalty) \tag{3.2}$$

In order to develop a meaningful framework for QoS control, it is necessary to have a quantitative model of user demand, service provision and admission policy. As a basic building block the $M/M/N/K$ queueing model will be used, augmented with the economic parameters of charges and penalties introduced above. As will become clear later in this chapter, under suitable assumptions about the nature of user demand, it is possible to evaluate explicitly the effect of particular server allocation and admission policies. Hence, a numerical algorithm for computing the optimal queue length threshold corresponding to a given partition of the servers among the service pools will be introduced. That computation is sufficiently fast to be performed on-line as part of a dynamic admission policy. Furthermore, it can be implemented in any system, but it might lose its optimality if the simplifying assumptions are not satisfied.

3.2 Performance Evaluation

In order to evaluate the system's performance, a utility function will be used. While different kinds of utility functions can be employed (*i.e.* [16, 19, 27, 28]), here the average revenue, R, obtained by the service provider per unit time is the considered metric. This value can be computed using the following expression:

$$R = \sum_{i=1}^{m} \gamma_i [c_i - r_i P(W_i > q_i)] \tag{3.3}$$

where:

- γ_i is the average number of type i jobs accepted per unit time, while
- $P(W_i > q_i)$ is the probability that the response time of an accepted type i job exceeds the obligation q_i.

If the service level agreements are expressed in terms of waiting times rather than response times, then W_i should be replaced by w_i.

In order to maximize the function described by equation (3.3) the service provider controls the *resource allocation* and the *job admission* policies. The first decides how to partition the total number of servers, N, among the m service pools. In other words, resource allocation algorithms assign n_i servers to jobs of type i, given the invariant $\sum_{i=1}^{m} n_i = N$. As mentioned before, under certain circumstances the allocation policy might deliberately decide to deny service to one or more job types.

The admission policy is defined by a set of thresholds, K_i $(i = 1, 2, \ldots, m)$: incoming jobs of type i are rejected if there are already K_i jobs in the system (waiting or executing). The extreme values $K_i = 0$ and $K_i = \infty$ correspond to accepting respectively none, or all, of the type i jobs.

Both the server allocations n_i and the admission thresholds K_i may, and should, change dynamically in response to changes in demand. The problem is how to do this in a sensible manner.

Finally, equation (3.3) uses a 'flat penalty' factor: as illustrated in Figure 3.2(a), if the job is executed within the obligation q the service provider does not pay any penalty, otherwise, no matter what the amount of the delay is, the provider must pay a penalty r. Such a model can be easily extended. For example, one could introduce penalties that are proportional to the amount by which the response time, or waiting time, exceeds the obligation q (see Figure 3.2(b)), as proposed in [123]. The effect of that would be to replace the term $P(W > q)$ in the cost function with $E(min(0, W - q))$. A similar analysis would apply.

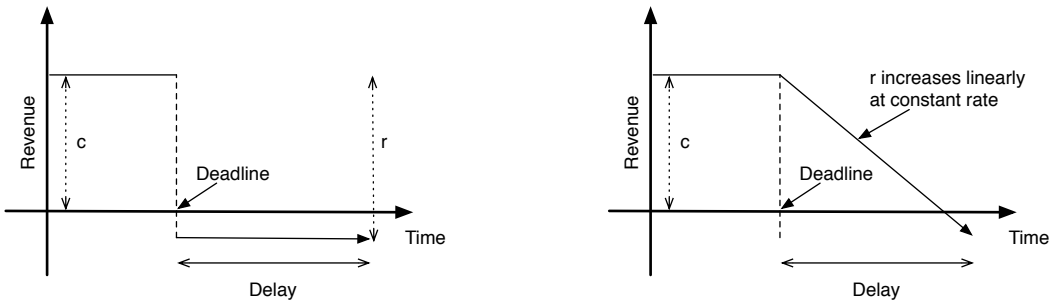


Figure 3.2: *Utility function with (a) flat penalties and (b) proportional penalties. In the first case as soon as the provider breaches the deadline, it must pay the whole penalty. In the second case the penalty is proportional to the delay (i.e. breaching the deadline by 10 seconds would be much more penalizing that failing to meet the SLA by 1 second).*

3.3 Heuristic Allocation Policies

The random nature of user demand and changes in demand pattern over time make capacity planning very difficult in the short time period and almost impossible in the long time period [65]. The well-documented 'Slashdot Effect' [91] shows that it is possible to experience more than 100–fold increases in demand when a site becomes popular [17] or when an exceptional event occurs (see Figure 3.3 [42]).

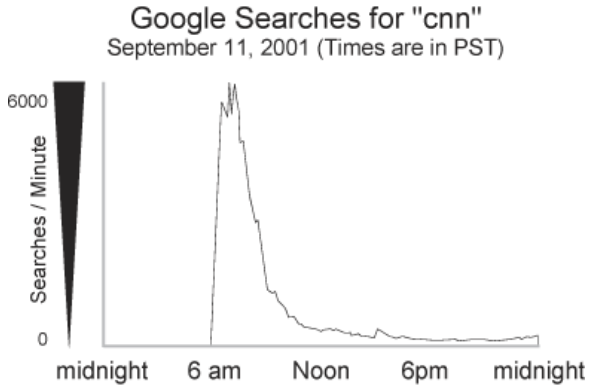


Figure 3.3: *Google searches for "cnn" on 09/11 [42].*

If servers are statically assigned, some services might be oversubscribed (*i.e.* unable to satisfy the promised QoS guarantees), while others might be underutilized (*i.e.* some servers would sit

idle). In such situations it could be advantageous to reallocate resources from one type to another, even at the cost of switching overheads [81].

The question that arises in that context is how to decide whether, and if so when, to perform such system reconfigurations. Posed in its full generality, this is a complex problem which does not always yield an exact and explicit solution. For this reason, as well as considering optimal policies for allocating servers, this chapter introduces two simple heuristic algorithms which, even though not optimal, perform reasonably well and are easily implementable. The proposed policies divide the system's lifetime into 'observation windows' (the interval between two consecutive policy invocations), such that a total of J jobs (of all types) arrive during a window. The idea is to monitor the traffic during the current window in order to make decisions about server allocations and job admissions during the next window. This technique is not new, but other scientists use time-based windows [16] or complex algorithms to detect traffic surges [14]. The use of events, instead, provides an elegant and simple implementation of 'adaptive windows': under heavy traffic conditions the allocation algorithm is executed more often than when the load is light.

Measured Loads heuristic From the statistics collected during a window, estimate the arrival rate, λ_i, and average service time, $1/\mu_i$, for each job type. For the duration of the next window, allocate the servers roughly in proportion to the traffic intensities, $\rho_i = \lambda_i/\mu_i$, and to a set of coefficients, α_i, reflecting the economic importance of the different job types. In other words, start by setting:

$$n_i = \left\lfloor N \frac{\rho_i \alpha_i}{\sum_{j=1}^{m} \rho_j \alpha_j} + 0.5 \right\rfloor \tag{3.4}$$

(adding 0.5 and truncating is the round-off operation). Then, if the sum of the resulting allocations is either less than N or greater than N, adjust the numbers so that they add up to N. When $\rho_i < n_i$ the system is *stable*, that is, the steady-state waiting time of requests entering the system is finite. When $\rho_i \geq n_i$, the system is *unstable* (or overloaded), and waiting/response times are unbounded.

Measured Queues heuristic From the statistics collected during a window, estimate the average queue sizes, L_i, for each job type. For the duration of the next window, allocate the servers roughly in proportion to those queue sizes and the coefficients α_i. That is, start by setting:

$$n_i = \left\lfloor N \frac{L_i \alpha_i}{\sum_{j=1}^{m} L_j \alpha_j} + 0.5 \right\rfloor \tag{3.5}$$

and then adjust the numbers so that they add up to N. If there are always requests in the systems, then the number of served customers is a Poisson process with rate μ. Hence, the expected number of jobs in the system at time t is at least $t(\lambda_i - \mu_i)$. If the mean service time multiplied by the number of available servers is less than the average interarrival time (*i.e.*, if $\lambda_i \geq n_i \mu_i$), than $\rho_i \geq n_i$ and L_i is unbound as $t \to \infty$ [89].

The intuition behind both of these heuristics is that more servers should be allocated to services which are *(i)* more heavily loaded and *(ii)* more important economically. The difference is that the first estimates the offered loads directly, while the second does so indirectly, via the observed average queue sizes. One might also decide that it is undesirable to starve existing demand completely. Thus, if the heuristic suggests setting $n_i = 0$ for some i, but there have been arrivals of that type, an attempt is made to adjust the allocations and set $n_i = 1$. Of course, as pointed out in Section 3.2, that is not always possible.

One could use different expressions for the coefficients α_i. Experiments have suggested that a good choice is to set $\alpha_i = r_i/c_i$ (the supporting intuition is that the higher the penalty, relative to the charge, the more important the job type becomes). If $r_i = c_i$, then $\alpha_i = 1$ and servers are allocated in proportion to offered loads (or queues).

Note that these heuristics may yield an allocation of $n_i = 0$ and an admission threshold $K_i = 0$ for some service types. If that is undesirable for reasons other than revenue, the allocations can be adjusted appropriately.

Finally, an allocation policy decision to switch a server from one pool to another does not necessarily need to take effect immediately. For example, in the implementation described in the next chapter, if a service is in progress at the time, it is allowed to complete before the server is switched (*i.e.*, switching is non-preemptive).

Greedy and Conservative Policies Under both allocation policies it may happen, either at the beginning of a window, or during a window, that the number of servers allocated to a pool is greater than the number of jobs of the corresponding type present in the system. In that case, one could

decide to return the extra servers to a 'spare pool' and to assign them temporarily to other job types. If that is done, the heuristic will be called *greedy*. The original policy, where no reallocation takes place until the end of the current window, will be called *conservative*. Thus, there are conservative and greedy versions of both the Measured Loads heuristic and the Measured Queues heuristic.

3.4 Admission Policies

Having decided how many servers to allocate to each pool, one should choose appropriate queue size thresholds for purposes of job admission. Unfortunately, the optimal value K_i depends not only on the number of servers allocated to queue i, and on the average inter-arrival and service times, but also on the distributions of those values. Therefore some simplifying assumptions about the arrival and service processes are made, and what appear to be the *best* thresholds under those assumptions are chosen. It is important to emphasize that such choices may be sub-optimal when the assumptions are not satisfied, but should nevertheless lead to reasonable admission policies.

According to the queueing theory terminology, a service center exhibits some service time distribution S,

$$S(x) = Pr[\text{service time} \leq x],$$

while client requests arrive at each service center according to an arrival process with a certain interarrival distribution A,

$$A(t) = Pr[\text{time between arrivals} \leq t]$$

The simplification consists of assuming that jobs of type i arrive according to an independent Poisson process with rate λ_i (it has been shown in [109] that the arrival rate at the application tier of e-commerce websites, *i.e.* the layer responsible for the creation of dynamic content, is indeed Poisson), and their required service times are distributed exponentially with mean $1/\mu_i$. These two assumptions allow a straightforward analysis of various aspects of the queueing system, such as the number of requests in the system or the waiting and service time distributions. Thus, for any fixed admission threshold K_i, allocation n_i and given sets of demand and economic parameters, queue i can be treated as an $M/M/n_i/K_i$ queue [78].

Denote by $p_{i,j}$ the stationary probability that there are j jobs of type i in the $M/M/n_i/K_i$ queue,

and by $W_{i,j}$ the response time of a type i job which finds, on arrival, j other type i jobs present. The average number of type i jobs accepted into the system per unit time, γ_i, is equal to:

$$\gamma_i = \lambda_i(1 - p_{i,K_i}) \tag{3.6}$$

Similarly, the probability $P(W_i > q_i)$ that the response time of an accepted type i job exceeds the obligation q_i, is given by:

$$P(W_i > q_i) = \frac{1}{1 - p_{i,K_i}} \sum_{j=0}^{K_i-1} p_{i,j} P(W_{i,j} > q_i) \tag{3.7}$$

By using the economic parameters described in equation (3.2) and equations (3.6) and (3.7) it is now possible to compute the average revenue, R_i, gained from type i jobs per unit time as:

$$R_i = \lambda_i[c_i - r_i P(W_i > q_i)] \tag{3.8}$$

The stationary distribution of the number of type i jobs present may be found by solving the balance equations:

$$p_{i,j} = \begin{cases} \rho_i p_{i,j-1}/j & \text{if } j \leq n_i \\ \rho_i p_{i,j-1}/n_i & \text{if } j > n_i \end{cases} \tag{3.9}$$

where $\rho_i = \lambda_i/\mu_i$ is the offered load for service type i. These equations can be solved iteratively, starting with $p_{i,0} = 1$, and then dividing each probability by their sum, in order to ensure that $\sum_{j=0}^{k_i} p_{i,j} = 1$.

The conditional response time distribution, $P(W_{i,j} > q_i)$, can be obtained as follows:

Case 1: $j < n_i$. There is a free server when the job arrives, in other words the job is executed immediately. The response time is distributed exponentially with parameter μ_i:

$$P(W_{i,j} > q_i) = e^{-\mu_i q_i} \tag{3.10}$$

Case 2: $j \geq n_i$. All servers allocated to queue i are currently busy, so the incoming job must wait for $j - n_i + 1$ departures before starting its own service. These occur at exponentially distributed interarrivals with parameter $n_i\mu_i$. In this scenario the conditional response time is

distributed as the convolution of an Erlang distribution with parameters $(j-n_i+1, n_i\mu_i)$, and an exponential distribution with parameter μ_i. According to [78], the Erlang density function with parameters (k,a) has the form:

$$f_{k,a}(x) = \frac{a(ax)^{k-1}e^{-ax}}{(k-1)!} \tag{3.11}$$

At this stage is possible to rewrite the conditional response time distribution as:

$$\begin{aligned} P(W_{i,j} > q_i) &= \int_0^{q_i} \frac{n_i\mu_i(n_i\mu_i x)^{j-n_i}}{(j-n_i)!} e^{-n_i\mu_i x} e^{-\mu_i(q_i-x)}\, dx \\ &+ \int_{q_i}^{\infty} \frac{n_i\mu_i(n_i\mu_i x)^{j-n_i}}{(j-n_i)!} e^{-n_i\mu_i x}\, dx \end{aligned} \tag{3.12}$$

By using some expressions for the Gamma integral [45] we obtain:

$$P(W_{i,j} > q_i) = \frac{e^{-\mu_i q_i} n_i^{j-n_i+1}}{(n_i-1)^{j-n_i+1}} + e^{-n_i\mu_i q_i} \sum_{k=0}^{j-n_i} \left[\frac{(n_i\mu_i q_i)^k}{k!} - \frac{n_i^{j-n_i+1}(\mu_i q_i)^k}{k!(n_i-1)^{j-n_i+1-k}} \right] \tag{3.13}$$

The right–hand side of equation (3.13) is not defined when $n_i = 1$. However, in that case the conditional response time $W_{i,j}$ has a simple Erlang distribution with parameters $(j+1, \mu_i)$:

$$P(W_{i,j} > q_i) = e^{-\mu_i q_i} \sum_{k=0}^{j} \frac{(\mu_i q_i)^k}{k!} \tag{3.14}$$

Other performance values If the SLA is formulated in terms of waiting time instead of response time, then R_i is given by an expression similar to equation 3.8, with $P(W_{i,j} > q_i)$ being replaced by $P(w_{i,j} > q_i)$ (where $w_{i,j}$ is the waiting time of a type i job which finds, on arrival, j other type i jobs present). Computing the conditional waiting time is more straightforward than computing the conditional response time. The conditional waiting time, $w_{i,j}$, of a type i job which finds j other type i jobs on arrival is:

- 0 if $j < n_i$, while
- Erlang distributed with parameters $(j-n_i+1, n_i\mu_i)$ if $j \geq n_i$

$$P(w_{i,j} > q_i) = e^{-n_i \mu_i q_i} \sum_{k=0}^{j-n_i} \frac{(n_i \mu_i q_i)^k}{k!} \tag{3.15}$$

The expressions discussed above, together with equation (3.8), enable the average revenue R_i to be computed efficiently and quickly. When that is done for different sets of parameter values, it becomes clear that R_i is a unimodal function of K_i. That is, it has a single maximum, which may be at $K_i = \infty$ for lightly loaded systems. We do not have a mathematical proof of this proposition, but have verified it in numerous numerical experiments. That observation implies that one can search for the optimal admission threshold by evaluating R_i for consecutive values of K_i, stopping either when R_i starts decreasing or, if that does not happen, when the increase becomes smaller than some ε. Such searches are typically very fast. Thus, having chosen a dynamic server allocation policy, a possible dynamic job admission policy is defined as follows:

Definition 3.4. *For each job type i, whenever its server allocation n_i changes, compute the 'best' admission threshold, K_i, by carrying out the search described above.*

There are quotation marks around the word 'best' because, as pointed out at the beginning of this section, that threshold may not be optimal if the exponential assumptions are violated. This admission policy can be applied in any system but is, in general, a heuristic. There is one exception to the above rule: if the allocation policy is greedy, and n_i changes because a server has been temporarily allocated to pool i as a result of not being needed elsewhere, then the threshold K_i is not recalculated. The rationale is that such a server may have to return to its original pool soon, so type i admissions should not be relaxed.

Note that, whenever $n_i = 0$ for some job type, then $K_i = 0$, that is jobs of that type are not accepted. If that is undesirable for reasons other than revenue, then either the allocations or the thresholds can be adjusted appropriately. For example, in the implementation that will be described in the next chapter, if there has been at least one arrival of type i during the current window, the resource allocator tries to assign at least one server to pool i for the next window.

3.5 Optimal Server Allocation

Rather than using a heuristic server allocation policy, one might wish to determine the *optimal* server allocation at the end of each window. The problem can be formulated as follows. For a given set of demand and economic parameters (the former are estimated from statistics collected during the previous window), find a server allocation vector, $(n_1, n_2, \ldots, n_m)$[1] which, when used together with the corresponding optimal admission threshold vector $(K_1, K_2, \ldots, K_m)$, maximizes the total average profit $R = R_1 + R_2 + \ldots + R_m$ (see equation (3.3)).

One way of achieving this is to try all possible server allocation vectors. For each of them, compute the corresponding optimal thresholds, evaluate the total expected revenue R, and choose the best. The number, s, of different ways that N servers may be allocated between m different service types is equal to the number of ways that the integer N can be partitioned into a sum of m components. This is equivalent to the number of ways that N indistinguishable balls may be allocated into m distinguishable boxes [108]. That number is given by:

$$s = \left(\begin{array}{c} N+m-1 \\ m-1 \end{array} \right) . \tag{3.16}$$

For each of these s server allocations, there is an optimal allocation threshold K_i for each service type i. For the purpose of computing those optimal thresholds, the queues can be decoupled, that is, queue i may be considered in isolation of the others. Furthermore, only the corresponding number of servers, n_i, is required. Thus, for each allocation set there is maximum achievable total average revenue R: choosing the largest of these yields the optimal server allocations as well as the optimal admission thresholds.

Unfortunately the exhaustive search method is feasible only when N and m are small. In fact, the complexity of determining the optimal threshold vector $(K_1, K_2, \ldots, K_m)$ for a given allocation set $(n_1, n_2, \ldots, n_m)$ is in the order of the number of services, $O(m)$. Hence, the complexity of the exhaustive search is on the order of $O(sm)$. The resulting pair of m-vectors is referred to as the *optimal configuration* of the system and is denoted by OC:

$$OC = [(n_1, n_2, \ldots, n_m), (K_1, K_2, \ldots, K_m)] \tag{3.17}$$

[1] As usual the invariant $\sum_{i=1}^{m} n_i = N$ holds.

The major part of the cost in computing the optimal configuration is governed by the number of server allocations that have to be examined and compared, *i.e.* by the value of s. That number can become very large when both N and m are large.

When the exhaustive search is too expensive to be performed on-line, a fast method for minimizing the cost can be employed. This can be justified by arguing that the revenue is a concave function with respect to its arguments $(n_1, n_2, ..., n_m)$. Intuitively, the economic benefits of giving more servers to pool i become less and less significant as n_i increases. On the other hand, the economic penalties of removing servers from pool j become more significant as n_j decreases. Such behaviour is an indication of concavity. One can therefore assume that any local maximum reached is, or is close to, the global maximum.

A fast search algorithm suggested by the above observation works as follows.

1. Start with some allocation, $(n_1, n_2, \ldots, n_m)$; a good initial choice is provided by the heuristic policies.

2. At each iteration, try to increase the revenue by performing 'swaps', *i.e.* increasing n_i by 1 for some i and decreasing n_j by 1 for some j ($i, j = 1, 2, \ldots, m$; $i \neq j$). Recompute K_i and K_j.

3. If no swap leads to an increase in R, stop the iterations and return the current allocation (and the corresponding optimal admission thresholds).

There are different ways of implementing the second step. One is to evaluate all possible swaps (there are $m(m-1)$ of them) and choose the best, if any. Another is to stop the current iteration as soon as a swap is found that improves the revenue, and go to the next iteration. In the worst case this would still evaluate $m(m-1)$ swaps, but in general the number would be smaller. The trade-off here is between the speed of each iteration and the number of iterations. Experiments have shown that the second variant tends to be faster than the first (but not always).

An algorithm of the above type reduces drastically the number of partitions that are examined. Its worst-case complexity is on the order of $O(zm^2)$, where z is the number of iterations; the latter is typically small. For example, in a system with $N = 50$, $m = 5$, the fast search algorithm found the optimal configuration in less than 0.5 seconds, whereas the exhaustive search took about 0.5 hours!

Unfortunately, there are some examples where the fast search algorithm terminates at a local maximum rather than the global one. However, those examples are rare and can be described as 'pathological': they involve situations where the globally best policy is to allocate 0 servers to a

pool and not accept any jobs of that type. Even in those cases, the local maxima found by the algorithm provide acceptable revenues.

The fast search algorithm can be performed dynamically, at the end of each window, instead of applying a heuristic server allocation policy. Furthermore, when both N and m are large, it is also possible to apply a 'compromise solution' whereby the number of iterations is bounded (*e.g.*, no more than 5 iterations). The resulting policy may be sub-optimal, but any improvements will be obtained at a small run-time cost.

3.6 About Charges and Penalties

Market forces usually imply that there is a relationship between the offered quality of service and the amount that the provider is able to charge for it. One might expect that the stronger the obligation, *i.e.* the lower the value of q_i, the higher the charge c_i that the users would be willing to pay. Then the question arises of what is the best QoS contract to offer for each service, *i.e.* the most profitable obligation q_i to undertake. To find out the exact nature of the relationship between q_i and c_i would require a market analysis which is beyond the scope of this dissertation. However, an idea of the possible trade-offs may be gained by assuming some simple dependency, such as a linear one. For example, when the performance measure is the waiting time, w, the relationship between q_i and c_i could have the following form (the index i is omitted for simplicity):

$$c = \begin{cases} c_1 - aq & \text{if } c_1 - aq > c_0 \\ c_0 & \text{otherwise} \end{cases} \tag{3.18}$$

The parameters c_1 and c_0 are the highest and lowest amounts, respectively, that can be charged for the service, while a is the slope at which the willingness to pay decreases with the weakening of the obligation. That relationship is illustrated in Figure 3.4.

If, in addition, the penalty r_i is assumed to be a function of the charge c_i, or of the pair (c_i, q_i), then the QoS contract triple (c_i, q_i, r_i) would be determined by the obligation q_i. There would be an optimal configuration for each vector of obligations $(q_1, q_2, \ldots, q_m)$, and one could search for the most profitable QoS contracts.

Some limited experimentation with the linear dependency described by equation 3.18 in the

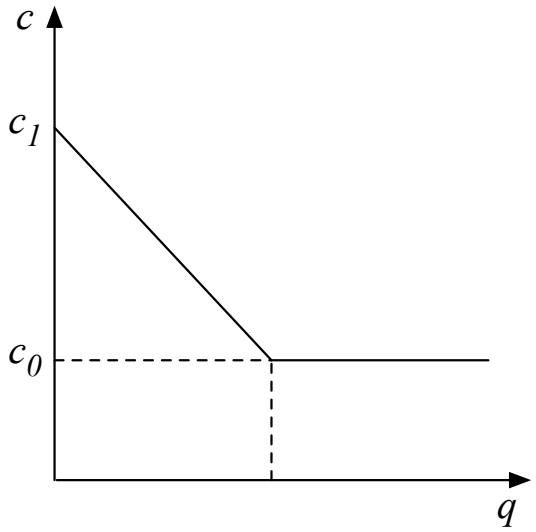


Figure 3.4: *Charge as function of obligation.*

context of a single service type has shown that the slope a plays a very important role: the steeper it is, the smaller the optimal obligation q_i.

3.7 Model Validation

Before implementing the framework on a real system some numerical experiments were carried out in order to *(i)* validate the algorithms presented earlier in this chapter and *(ii)* evaluate the benefits of determining the optimal system configuration. To reduce the number of variables, the following features are fixed:

- The QoS metric is the response time, W;
- The obligations undertaken by the provider are that jobs will complete within twice their average required service times, *i.e.* $q_i = 2/\mu_i$; and
- All penalties are equal to the corresponding charges: $r_i = c_i$, $\forall\, i$, that is if the response time exceeds the obligation, users get their money back.

Finally, in the following numerical experiments, the arrival processes are Poisson and the service times are exponentially distributed. However in the next chapter the system's behavior will be assessed with clustered arrivals and with non-exponential distributed service times.

The first experiment examines the effect of the admission threshold on the achievable revenue. A single service is offered on a cluster of 10 servers ($N = 10$, $m = 1$). The average job length and the charge per job are $1/\mu = 1.0$ and $c = 100$, respectively. The results are shown in Figure 3.5, where the revenue earned, R , is plotted against the admission threshold, K, for three different arrival rates.

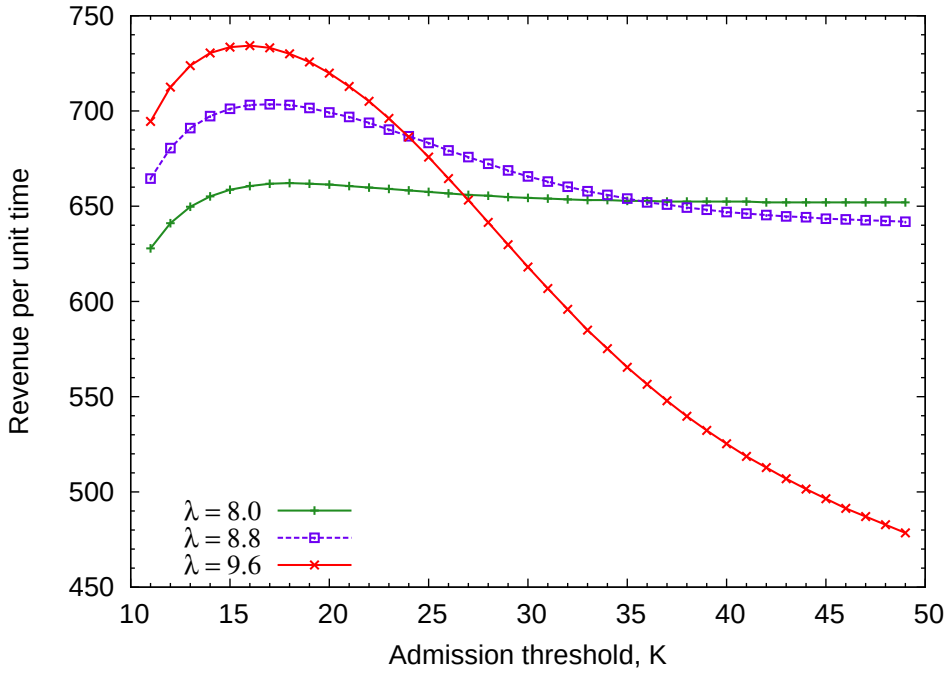


Figure 3.5: *Revenue as function of admission threshold. The heavier the load, the more important is to operate close to the optimum threshold.* $N = 10, m = 1, \mu = 1.0, c = r = 100$.

The figure illustrates the following points, of which the last is perhaps less intuitive than the others.

- In each case there is an optimal admission threshold.
- The heavier the load, the lower the optimal threshold but the higher the maximum achievable revenue.
- The heavier the load, the more important it is to operate at or near the optimum threshold.

Thus, when $\lambda = 8.0$, the optimal admission threshold is $K = 18$; however, much the same revenue would be earned by setting $K = 10$ or $K = \infty$. When the arrival rate is $\lambda = 8.8$, about 10% higher revenue is obtained by using the optimal threshold of $K = 17$, compared with the worst one

of $K = 1$. When the arrival rate increases further to $\lambda = 9.6$, the revenue drops very sharply if the optimal admission threshold of $K = 16$ is exceeded significantly.

The next experiments involve two service types with the aim to evaluate the effects of both resource allocation and admission control on the maximum achievable revenue. About resource allocation, apart from the last experiment, the switching policy is to do no switching at all: servers are statically allocated to the two queues in proportion to the average load. The second experiment concerns a 20-server system offering two services ($N = 20$ and $m = 2$). The 21 possible server allocations (n_1, n_2) are evaluated and compared, for three different pairs of arrival rates. In each case, the total offered load is $\rho_1 + \rho_2 = 15.0$, which means that the 20-server system is 75% loaded. The average service times and job charges are $1/\mu_1 = 1/\mu_2 = 10.0$ and $c_1 = c_2 = 100.0$, respectively. For each server allocation, the optimal pair of admission thresholds is determined and used.

Figure 3.6 shows the total revenue earned, $R = R_1 + R_2$, as a function of n_2. When the two arrival rates are equal, that is $\lambda_1 = \lambda_2 = 0.75$, the demand is symmetric and so the optimal allocation is $n_1 = n_2 = 10$. The optimal admission thresholds are $K_1 = K_2 = 19$. For asymmetric demands, as one might expect, it is better to allocate more servers to the more heavily loaded service. Thus, if $\lambda_1 = 0.5$ and $\lambda_2 = 1.0$, the optimal allocation is $n_1 = 7$, $n_2 = 13$, with admission thresholds set to $K_1 = 14$ and $K_2 = 24$ respectively. Lastly, when $\lambda_1 = 0.2$ and $\lambda_2 = 1.3$, the optimal allocation is $n_1 = 4$, $n_2 = 16$, with admission thresholds $K_1 = 9$ and $K_2 = 28$.

Finally, it is worth noting that the optimal server allocations are quite close to those suggested by the Measured Load heuristic, $(10, 10)$, $(7, 13)$ and $(3, 17)$ respectively.

The aim of the next two experiments is to evaluate the quality of the Measured Loads heuristic allocation policy. The quality Measured Queues heuristic cannot be measured via simulation, therefore its performance evaluation will be assessed in the next chapter.

Figure 3.7 shows the revenues obtained by the optimal policy and the heuristic, in the context of a 20-server system with two job types. Type 1 jobs are ten times longer, on the average, than type 2 ($\mu_1 = 0.02$ and $\mu_2 = 0.2$, respectively). The arrival rate for type 1 jobs is fixed to 0.2, while λ_2 varies from 0.4 to 1.8. The former corresponds to a total offered load of $\rho_1 + \rho_2 = 12$, *i.e.* the system is 60% loaded, while the latter means that the total offered load is $\rho_1 + \rho_2 = 19$, that is, the system is 95% loaded. The heuristic performs very well throughout. Its sub-optimality becomes

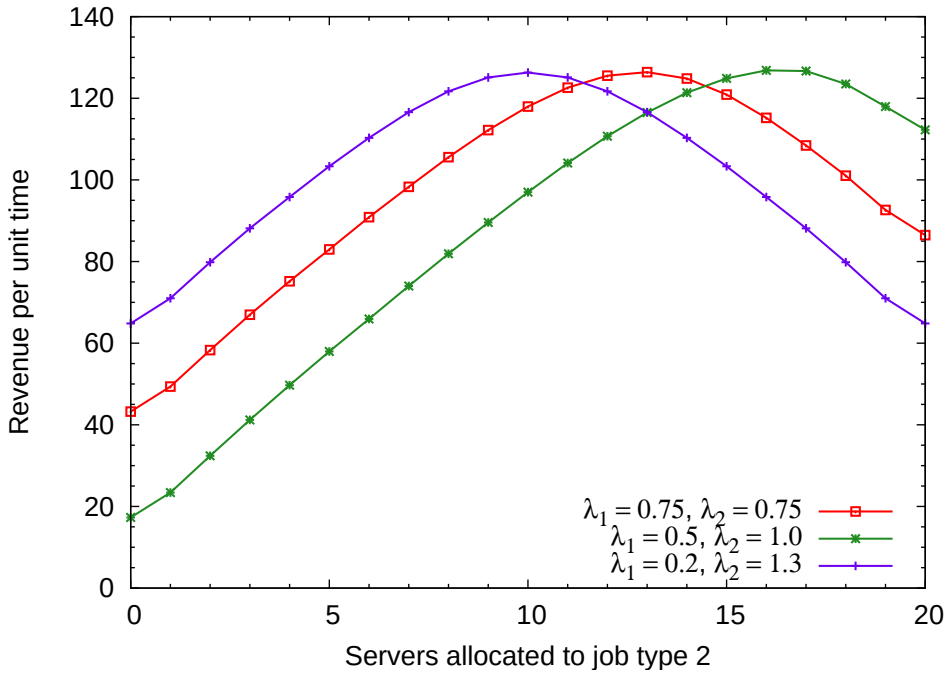


Figure 3.6: *Maximum revenue earned for different server allocations. When the demand is asymmetric it is better to allocate more servers to the service type that is more heavily loaded.* $N = 20, m = 2, \mu_i = 0.1, c_i = r_i = 100$.

noticeable only when the system is quite heavily loaded (see table 3.1).

In the next experiment, a 20-server system with 2 types of service was subjected to fluctuating demand controlled by a single parameter, λ. During a period of time of length 1000, jobs of type 1 and 2 arrive at rates $\lambda_1 = \lambda$ and $\lambda_2 = 10\lambda$, respectively. Then, during the next period of length 1000, the arrival rates are $\lambda_1 = 10\lambda$ and $\lambda_2 = \lambda$, respectively; and so on. The average service times for the two types are equal, $1/\mu_1 = 1/\mu_2 = 0.8$, as are the charges, $c_1 = c_2 = 100$.

Again, the aim is to compare the total revenues earned by the optimal and the heuristic configurations. In addition, a third policy which uses the same server allocations as the heuristic, but does not restrict admissions (*i.e.*, $K_1 = K_2 = \infty$), is included in the comparison. In all cases, it is assumed that, at the beginning of every new period, the demand parameters become known instantaneously, so that the server allocations and admission thresholds can be computed and applied during that period. It is worth pointing out that, in order to avoid the question of whether the system reaches steady state during each period, the comparisons were done by simulation.

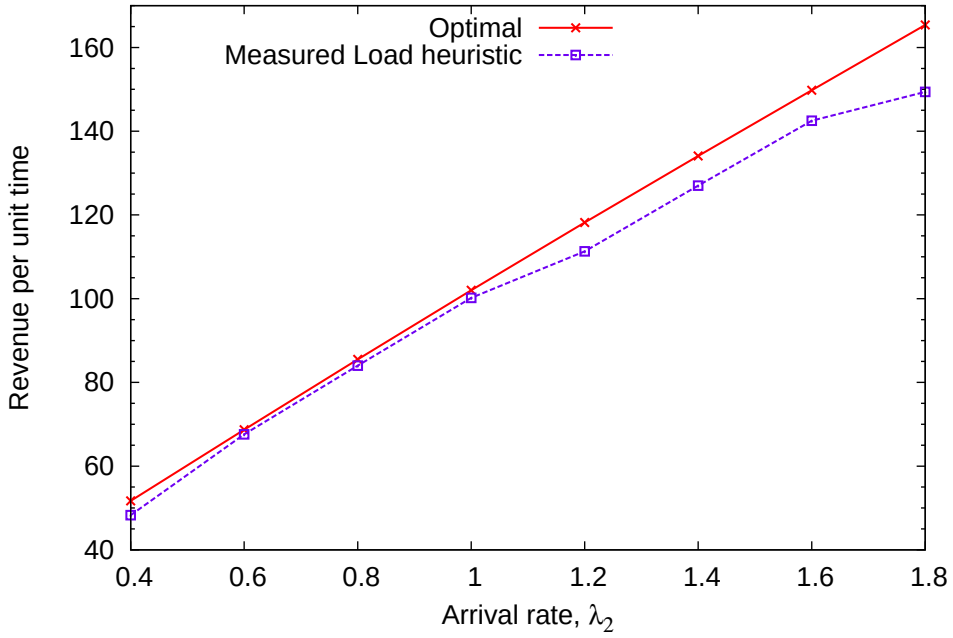


Figure 3.7: *Comparison between the optimal and the Measured Load heuristic policies. The sub-optimality of the heuristic becomes noticeable only when the system is heavily loaded.* $N = 20, m = 2, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$.

In Figure 3.8, the total revenue earned per unit time by the three policies is plotted against the offered load (which is equal to $11\lambda/\mu_1$). The near-optimality of the heuristic is rather remarkable. In contrast, the revenues earned by the unrestricted admission policy increase more slowly, and then drop sharply as the load becomes heavy. This example demonstrates that, by itself, a sensible server allocation is not enough; to yield good results, it should be accompanied by a sensible admission policy.

3.8 Summary

This chapter has introduced a quantitative framework where the performance of a service provisioning system is measured by the average revenue earned per unit time. Some easily implementable policies for dynamically adaptable service provisioning systems subject to QoS constraints and SLAs have been introduced. The presented experiments have demonstrated that policy decisions

	Optimal				MLH			
λ_2	n_1	n_2	K_1	K_2	n_1	n_2	K_1	K_2
0.4	14	6	28	16	17	3	38	6
0.6	13	7	24	17	15	5	31	11
0.8	12	8	21	19	14	6	28	12
1.0	11	9	18	20	13	7	24	14
1.2	10	10	15	22	13	7	24	12
1.4	9	11	13	23	12	8	21	14
1.6	8	12	11	25	11	9	18	15
1.6	7	13	9	26	11	9	18	14

Table 3.1: *Comparison between the optimal and the Measured Loads heuristic policies.*

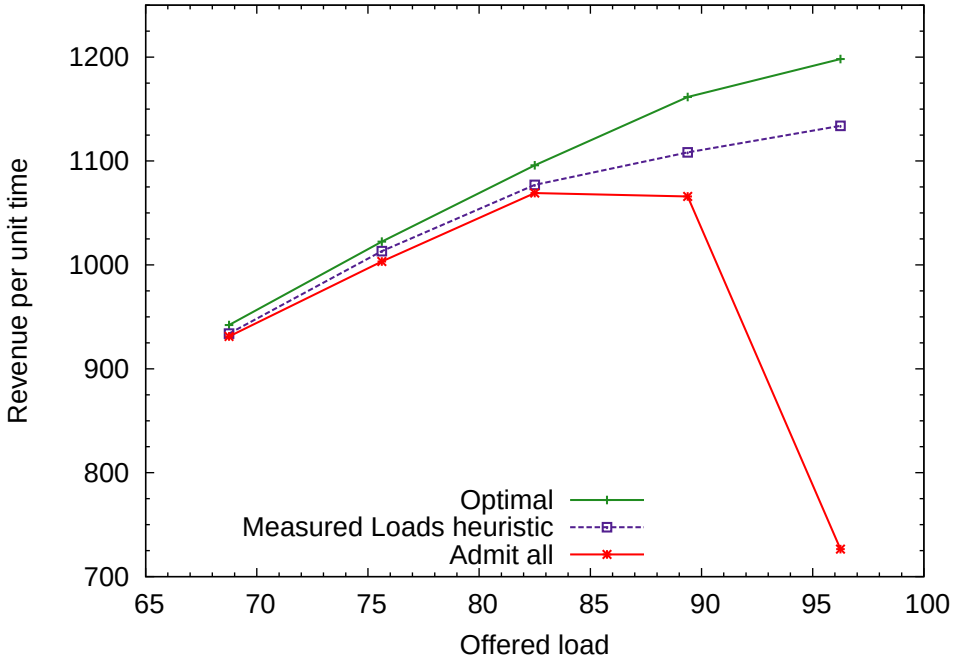


Figure 3.8: *Policy comparisons: revenue as function of load. The revenue earned by the unrestricted admission policy increases more slowly than the others, and drops very quickly as the loaded increases.* $N = 20, \mu_i = 0.8, c_i = r_i = 100$.

such as server allocations and admission control can have a significant effect on the earned revenue. Moreover, those decisions are affected by the contractual obligations between clients and the service provider in relation to quality of service.

Having made some simple assumptions about the nature of service demand, an algorithm to compute the optimal system configuration has been presented. When that computation becomes

too expensive to be performed on line, that is, when the numbers of offered and available servers are large, two simple heuristics have been proposed. Experimentation with the Measured Loads heuristic suggests that it is close to optimal, while the validation of the Measured Queues heuristic is delayed until the next chapter.

Chapter 4

SPIRE: A Next Generation Management System for Internet Data Centers

This chapter discusses the design and prototype implementation of SPIRE (Service Provisioning Infrastructure for Revenue Enhancement), a management system for enterprise data centers designed with a utility computing paradigm in mind [64, 116]. The core idea behind SPIRE is to group the available hardware resources into a common infrastructure and to share those resources across the provided services. Current Internet utilities use SLAs to help the data center promise what is possible to deliver and achieve the QoS goals [75, 120]. SPIRE, instead, does not make any assumption about the SLAs in operation (*i.e.*, whether the promised QoS levels are realistic or not); instead, it enforces the SLAs in place by monitoring income and expenditure dynamically and (i) migrating servers among hosted services as their demands change over time and (ii) selecting the requests that will be executed and the ones that will not.

A preliminary version of SPIRE has been presented in [72] and it has been deployed at British Telecom's Research and Development Laboratories, where it is used for experimental purposes. Also, it is subject to two European Patent applications.

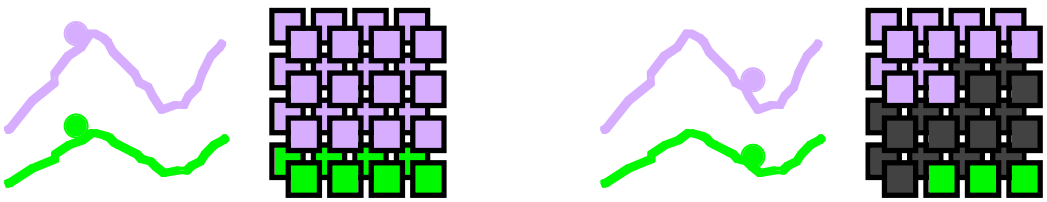
This chapter demonstrates the use of SPIRE to implement the framework defined in Chapter 3 for the execution of single jobs. The proposed middleware is flexible enough to be extended to handle streams of requests (Chapter 7).

4.1 Overview of SPIRE

4.1.1 Architecture of Commercial Data Centers

An enterprise hosting platform, or data center, is a collection of servers interconnected by high-speed, switched LANs that hosts content and runs third-party applications (or services) accessed over the Internet in return for payments. Service provisioning systems are very advantageous because they offer economies of scale for network, power, cooling, administration, security and surge capacity [29]. Different models can be used to design such platforms, the most widely-used ones being 'dedicated' and 'shared' architectures. As the name implies, the dedicated model is a hosting platform where servers are not shared [11, 86]: the entire provisioning system is dedicated to run a single application or, more often, each application runs on a subset of the available servers (as in the 'managed hosting' scenario [4, 74]), while each machine is allocated to at most one application at any given time. In other words, this model allocates resources at a granularity of entire machines.

If the contract allows the service provider to use the same server to run different applications, then a dynamic allocation scheme can be used. This means that, no matter what definition of load is used, every Δt time units the service provider can estimate the load for each application and change the amount of machines running the hosted services accordingly, as illustrated in Figure 4.1.



(a) Allocation during a peak load.

(b) Allocation during a low load period.

Figure 4.1: *Dynamic resource allocation. (a) During peak load periods all servers are allocated to the hosted applications and services while, (b) During low load periods only the necessary servers are allocated. The others are kept in a special 'virtual' pool (eventually running in energy-save mode)*

Depending on the software and hardware in operation, if a dedicated architecture is employed, any reallocation epoch might involve: (i) deallocation of the server from another customer, (ii) disk scrubbing, to avoid data leaks, (iii) OS/VM or application installation, and (iv) startup. Of course, performing these tasks takes time, from a few seconds to several minutes. This poses a lower bound

on the length of the configuration intervals, *i.e.*, how often reallocations can be made. Recent efforts have recognized the need to reduce the amount of time needed to reconfigure dedicated provisioning systems and have proposed several approaches aiming to mitigate the problem. Possible solutions include installing the needed OS/VM and applications from remote boot images, fast application switching or the use of virtual data centers, eventually associated to a 'reserve pool' of idle servers in energy-save mode to be used during switches [38, 46, 79].

The most widely used alternative, *i.e.*, the shared architecture, does not allocate entire servers. Instead, it runs multiple applications on each server and multiplexes the server resources among these applications. Apart from its low degree of security and isolation, the major drawback of this architecture is that if one wants to control the amount of resources (*i.e.*, CPU time) each job type can use, he/she must employ complex resource management algorithms such as proportional-share schedulers [23] to enforce the fine-grained allocation values (about 10% of the server capacity). The problem with such systems is that they are not flexible. In other words, changing the allocation by changing the size of the quanta and the amount of quanta each job type receives is not possible at runtime.

SPIRE follows the dedicated pattern because that is by far the most flexible way to implement adaptive data centers. In SPIRE, changing the number of resources running a certain application is simply a matter of routing requests: if the system is carefully designed, this task is extremely flexible and can be performed very quickly.

4.1.2 Design Challenges

The overall increase in traffic on the Internet causes a disproportional increase in client requests to popular Web Sites, especially in conjunction with special events [65]. System administrators continually face the need to increase the server capacity. An easy approach would be to mirror the information across the available servers, however such a model is not transparent to the users as they should manually choose a URL. Moreover, it does not allow the system to control the way incoming requests are distributed, *i.e.* it does not provide any load balancing mechanism. A better approach would consist in a distributed architecture capable of routing the jobs among the available servers in a flexible and transparent manner (*i.e.* capable of reacting to any change in the operational environment, such as the availability of new servers). This model can be implemented in different ways, for example (i) via DNS, (ii) through a dispatcher or (iii) via a two-level dispatching mechanism involving the DNS as well as the servers in the cluster. While all of these alternatives

have pros and cons (for a detailed comparison, see [24]), SPIRE has been designed according to the dispatcher model (or mediation service [52]) as this is by far the most flexible way to implement Service Oriented Architecture (SOA) solutions. The proposed middleware system hides the IT infrastructure from the clients and creates an illusion of a single system [21] by using a Layer-7 two-way architecture [20, 25]. The load balancer:

- Forwards packets in both directions, client-to-server and server-to-client (packet double-rewriting), as shown in Figure 4.2;
- Takes routing decisions using only the information available at the application layer of the OSI stack [83], such as target URL or cookie.

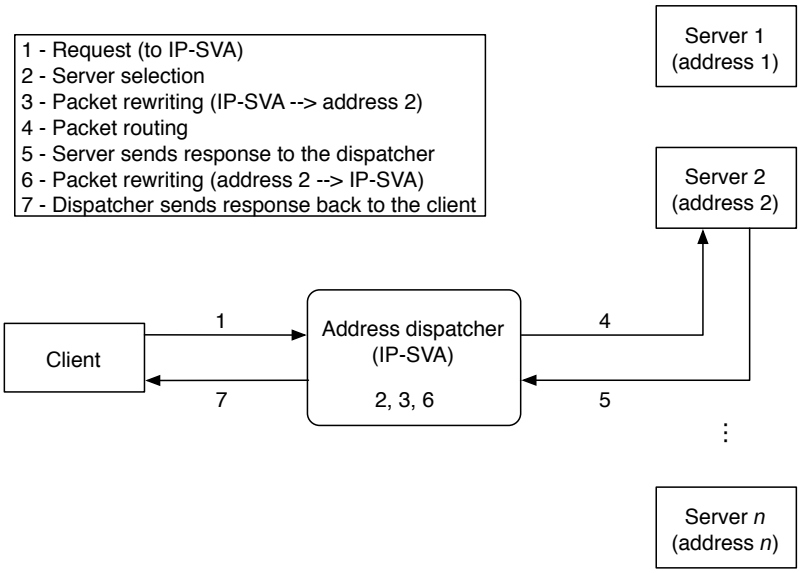


Figure 4.2: *Packet double-rewriting model: the dispatcher has a single, virtual IP address (IP-SVA) and rewrites packets in both directions (steps 3 and 6).*

The immediate consequence is that it becomes straightforward to add or remove servers because clients do not know where their requests will be executed. The main advantage of Layer-7 over Layer-4 load balancers is that Layer-7 load balancers offer good policy management functionalities, *i.e.* the ability to define, integrate with existing policies and enforce, during the runtime cycle, policies such as access control and service level agreement compliance. Layer-4, instead, perform essentially a content-blind dispatching, which is faster and easy to implement, but less efficient because the employed routing algorithms are essentially stateless.

A high level view of the resulting architecture is depicted in Figure 4.3. SPIRE uses a dedicated hosting model (see previous section), *i.e.*, the available machines are dynamically grouped into virtual pools, where each pool deals with demands for a particular service. All incoming requests are sent to the cluster controller (arrow 1) and handled by the appropriate Service Handler. There is one Service Handler for each of the provided services and for each performance level: if the same service is offered at different QoS levels (*e.g.*, gold, silver and bronze) the Service Handler will be instantiated at differentiated service levels. Each differentiated level will have its own SLA management function that strives to meet the level of service specified by the differentiation. Thus, each Service Handler instance includes the admission control policy (notifying users of rejections, arrow 7), scheduling policy and the collection of statistics through a profiler and the management of the corresponding pool of servers. The results of completed jobs are returned to the Controller, where statistics are collected, and to the relevant user (arrows 5 and 6).

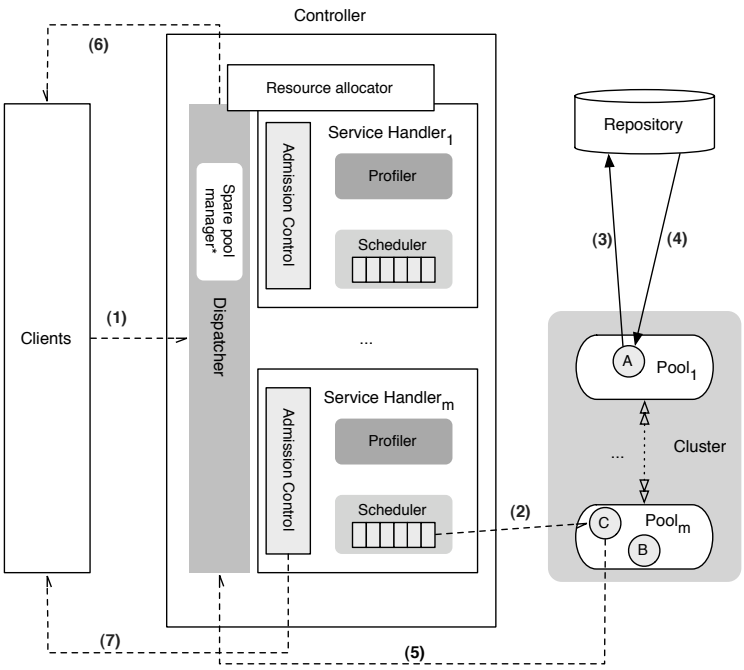


Figure 4.3: *Architecture overview. Dotted lines indicate asynchronous messages.*

If a deployment is needed, the service is fetched from a remote repository (arrows 3 and 4). This incurs a migration time of a few seconds, while a server is switched between pools. SPIRE tries to avoid unnecessary deployments, while still allowing new services to be added at runtime. In such cases, the appropriate Service Handlers are automatically created in the Controller. If there is

sufficient space on a server, deployed services could be left in place even when not currently offered on that server. One would eventually reach a situation where all services are deployed on all servers. Then, allocating a server to a particular service pool (handler) does not involve a new deployment; it just means that only jobs of that type would be sent to it. In those circumstances, switching a server from one pool to another does not incur any overhead.

4.2 API and Implementation

This section discusses the API and implementation details of both the controller and the servers. Section 4.2.1 illustrates the implementation of the request forwarding algorithm and explains how incoming requests are handled. Section 4.2.2 shows the implementation of the dispatcher, while Section 4.2.3 shows the algorithm used by SPIRE to deploy new services at runtime. The prototype has been implemented in Java and relies on the Apache Axis2 framework [99] to handle SOAP messages [114]. Even though SPIRE is transport agnostic, that is all the transports supported by Axis2 can be employed (HTTP, TCP, JMS [98] and POP3/SMTP, as well as user defined protocols), in the rest of this chapter it is assumed that the parties communicate using HTTP, as this is by far the most widely used protocol to exchange SOAP messages over the Internet.

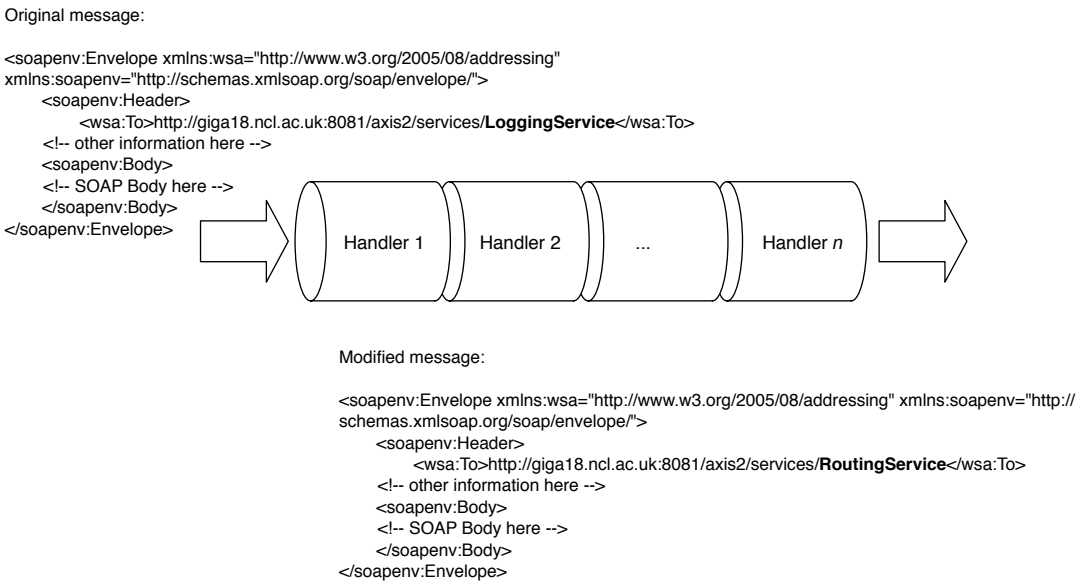


Figure 4.4: *Chain of handlers. The last part of the target endpoint has been modified from 'RoutingService' to 'LoggingService'.*

4.2.1 Packet Double-Rewriting

SPIRE's message forwarding algorithm uses the WS-Addressing[1] [112] information available in the SOAP header of incoming messages to take routing decisions and to dispatch them: the current implementation uses a custom Axis2 handler placed on the controller's chain in order to change the addressing information of the incoming messages and redirect them to the mediation service. Axis2 is a pure SOAP processing engine that does not understand data bindings, transports or Web Service Description Language (WSDL) [111]. Its main functionality is to process and deliver messages to a certain destination, or endpoint. In order to customize the default behavior it is possible to write modules, *i.e.* collections of handlers [57, 58]. The concept of message interceptor, or handler according to the Axis2 terminology, is a widely used concept in messaging systems: its task is to intercept the messaging flow (Figure 4.4) and do whatever task it is assigned to it, such as message validation or content enrichment.

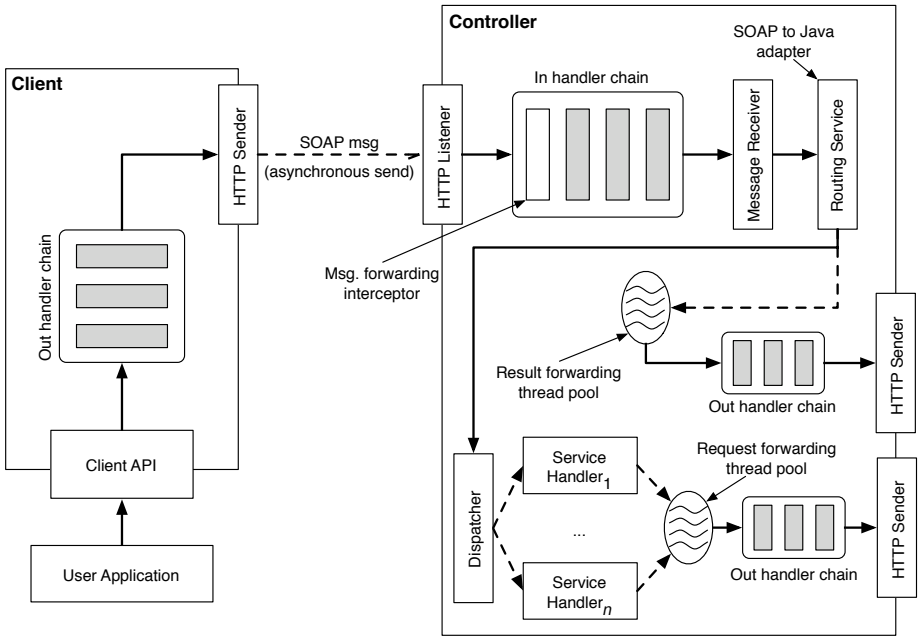


Figure 4.5: *Axis2-based implementation of the Layer-7 load balancer. The packet double-rewriting is performed by the first message interceptor.*

Figure 4.5 illustrates the load balancer's architecture. The HTTP listener initializes the execution of incoming messages. Before reaching the target service every message passes through

[1] WS-Addressing provides transport-neutral mechanisms to address messages and Web Services.

the input sequence of interceptors. Each Axis2 handler has an `invoke()` operation that takes a `MessageContext` object[2] as argument. The first handler in the input chain, `RouterDispatcher`, is the interceptor in charge of performing the message forwarding. Its `invoke()` implementation is outlined in Figure 4.6. First, the handler gets the SOAP header and retrieves the message type. Only messages coming from other SPIRE components (servers and repository) have a header portion specifying the type of message the system is trying to handle (see Figure 4.7). This means that, if there is no message type, the received message is a client request. Then (not shown), the original target service, operation and SOAP action are saved into the MessageContext, together with the arrival time-stamp, and the message is redirected to the mediation service by replacing the above values with the ones identifying the mediation service: `RoutingService`, `forward` and `urn:forward`, respectively.

```
public InvocationResponse invoke(MessageContext msgCtx) throws AxisFault {
  // get the SOAP header
  SOAPHeader header = msgCtx.getEnvelope().getHeader();
  // scans the SOAP header
  final String msgType = SOAPHeaderConstants.getQospMsg(header);
  if (msgType == null) {
    // received client request
    try {
      clientRequest(msgCtx);
    } catch (AxisFault) {
      // handle/redirect error to the inFault chain
    }
  } else {
    // internal message
    if (MessageType.RESULT.value().equalsIgnoreCase(msgType)) {
      // received result, remove the WSA_RELATES_TO header portion
    }
  }
  return InvocationResponse.CONTINUE;
}
```

Figure 4.6: *`RouterDispatcher` implementation of the `invoke` operation.*

After passing through the interceptors' pipe, the `MessageReceiver`[3] uses Java reflection [92] to find the target class and method, then it invokes the business logic of the mediation service passing as argument the SOAP body of the received message. All the operations exposed by the

[2] `MessageContext` objects are the placeholder of SOAP messages and related properties. They are stored into thread-local variables.

[3] Different types of message receiver exist, *i.e.* to handle In-Only, In-Out, asynchronous operations, *etc*.

`RoutingService` use the `RawXMLInOnlyMessageReceiver`. In other words:

1. All operations have an *In-Only* Message Exchange Pattern (MEP) [115], *i.e.* they are all defined as one-way operations, even though the client expects a response (the packet rewriting algorithm will take care of delivering the result of the computation to the client as soon as it becomes available);

2. The mediation service works at the XML level. In other words it does not need any WSDL nor intermediate layers to marshal and unmarshal the message content. Moreover, the mediation service uses exclusively the information contained into the message header to take routing and admission control decisions. This implies, for example, that SPIRE can handle encrypted messages. Finally, since the SOAP body is not needed, the XML tree of the SOAP body is never built because the parsing model in operation (StAX) allows for on-demand building of the object tree (*i.e.* the object tree is built only when it is needed) [100].

```
public enum MessageType {
    // forward operation
    FORWARD("Forward"),
    // result operation
    RESULT("Result"),
    // used to add new services at runtime (i.e. to create a new ServiceHandler)
    NEW_SERVICE("New service"),
    // used to dynamically add new nodes to the cluster
    NEW_NODE("New node"),
    // used to dynamically deploy new services
    DOWNLOAD("Download");
}
```

Figure 4.7: *`MessageType` enum.*

Figure 4.8 shows the API for the controller: (*i*) `RoutingService` is a stateful web service used as a layer to validate the XML information contained into the SOAP header of the received messages, to translate them into Java objects, and to forward the results back to the client, (*ii*) `Request` holds all the necessary information needed to execute a request, and (*iii*) `Response` contains the data needed to update the statistic and QoS values. All `Request` and `Response` objects have a unique identifier and store the name of the service they are dealing with.

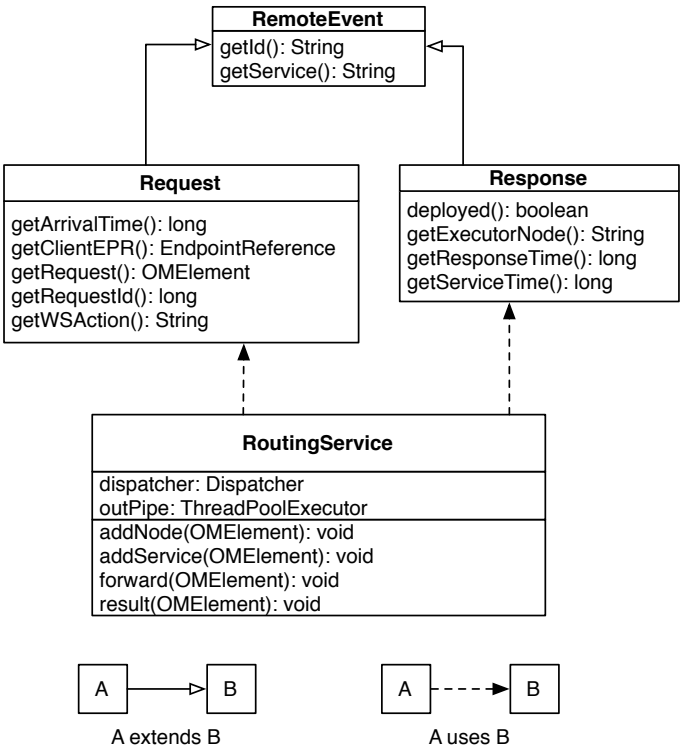


Figure 4.8: *`RoutingService` API.*

RoutingService is the entry point to the controller. It exposes four operations that take as argument an `OMElement` object, which is nothing else than the SOAP body of the received message[4]:

1. `addNode()` is used to add servers to the cluster managed by the controller. Even though this kind of functionality is not new (especially in Grid environments), adding a new node involves updating many runtime parameters, mainly because the system capacity has increased. As a result, the admission control policy might be relaxed. Two scenarios are possible, (i) the controller is already running and a new server is added to the cluster, for example in response to a traffic surge if the cluster management does not want to lose too much revenue, and (ii) the provisioning infrastructure is already in place and the controller is (re)started. In the first case, at server's start-up, each machine automatically sends a SOAP message to the `RoutingService` (the endpoint of the `RoutingService` is a 'well-known' address specified into a configuration file). Received messages look like the one shown in Figure A.1

[4]If needed, it is always possible to retrieve the whole message, including its header, via the `MessageContext`.

(page 142). In the second case, instead, multicast communications are employed. The controller first starts a network listener (both TCP and UDP are supported), then it sends a multicast message to a 'well-known' multicast group (again, the group name is specified in a shared configuration file). All servers run a multicast listener waiting for messages sent to that group. When a machine receives a `BootstrapProtocol` object (see Figure 4.9) containing the controller coordinates, it replies by sending to the controller a `BootstrapProtocol` instance including its IP address and port.

```
public interface BootstrapProtocol extends java.io.Externalizable {
    String getHost();
    int getPort();
    void setHost(String host);
    void setPort(int port);
}
```

Figure 4.9: *`BootstrapProtocol` interface.*

2. `addService()` is the operation invoked by the repository in order to add new services. Messages like the one shown in Figure A.2 (page 143) are sent to the controller every time the repository server is started or a new service is added to the repository. As a result, the controller creates a new `ServiceHandler` for each service with the QoS values specified in the message.

3. `forward()` is the operation invoked once the message has been redirected to the controller. The business logic of the method simply creates a new `Request` object containing the client request and some information stored into the `MessageContext` and passes it to the `Dispatcher`. The way such requests are handled is discussed into the next section.

4. `result()` handles result messages coming from the servers (see Figure A.3, page 144). This method first extracts the `qosp` block from the SOAP header and creates a `Reponse` instance with the values it contains. Such header portion holds both time and service-related informations. The former includes the arrival timestamp and the service time, while the latter specifies which machine executed the request, whether a service deployment was carried out, the message identifier of the request and the SOAP action of the service. Then it (i) removes the `qosp` header-block from the received message, (ii) forwards it to the client (the request

identifier is used by the client to correlate the request and the response), and (*iii*) passes the `Response` object to the `Dispatcher`.

SPIRE uses the Stage Event Driven Architecture (SEDA) [118] to perform I/O operations. The result as well as the request forwarding pipes use a dedicated thread pool with a blocking working queue and a bounded number of worker threads (Figure 4.10). This model provides a simple mechanism for backpressure: if the worker threads on the output pipe are temporarily unable to cope with the flow of responses, the output pipe will use one of the threads of the listener queue. This implementation is quite primitive and, if the input queue of the HTTP listener is full, it might end up rejecting responses. A more sophisticated approach overcoming the aforementioned limitation would consist in decoupling the input and the output queues. Such option can be implemented by exposing two endpoints to handle the request and response flows. This would result in two different, unrelated, queues.

It is worth stressing that results do not need to be redirected to the mediation service. In fact, as shown in the `wsa:To` block in Figure A.3, servers send such messages to the mediation service, while the final destination is specified into the `wsa:ReplyTo` block. This means that the destination of messages coming from servers is not changed at runtime. Instead, they are simply forwarded to the destination specified by the `wsa:ReplyTo` block.

```
outPipe = new ThreadPoolExecutor(1, 5, 10, TimeUnit.SECONDS,
    new ArrayBlockingQueue<Runnable>(20, true),
    new ThreadPoolExecutor.CallerRunsPolicy());
```

Figure 4.10: *Implementation of the output pipe providing a simple backpressure mechanism.*

Request instances store the received requests as well as the time they entered the system, the client endpoint (used to send the response), the message identifier (used by the client to correlate the request and the response), the object identifier (used by SPIRE) and the target service and operation.

Response objects are used by SPIRE to update statistical information used to enforce the SLAs in operation via the allocation and admission policies. Unlike `Request` instances such objects do not store any XML nor routing information. As a matter of fact the result is sent back using the

output pipe while `Responses` are handled by `ServiceHandlers`. The state of these objects is composed by the object identifier, the type of request executed, the server that executed the job, the service and response time and a boolean value indicating whether a service deployment was needed to execute the request.

4.2.2 Requests Dispatching and Routing

The dispatching API, illustrated in Figure 4.11, implements the functionalities exposed by the `RoutingService`. It allows the system to add nodes and services at runtime, handle incoming requests and result messages, and dynamically change the amount of servers allocated to each queue. This section first gives an overview of the functionalities provided by each component, then it discusses the information flow of the four operations exposed by the `RoutingService`. It is worth mentioning that at this stage there is no trace of XML, only `Request`, `Response` and `String` objects are used.

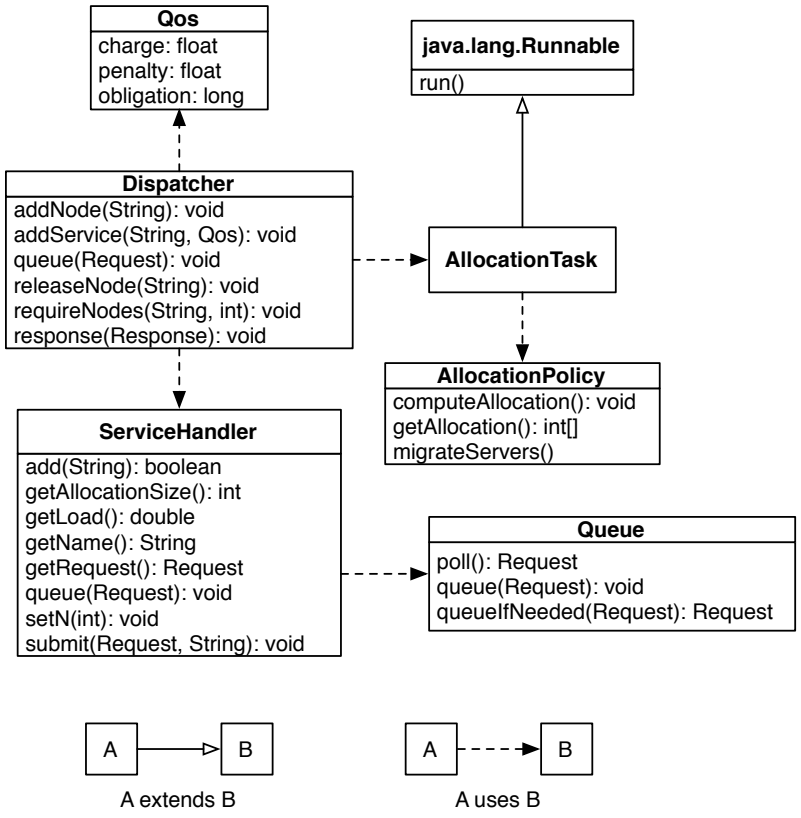


Figure 4.11: *Dispatching API. The `Request` and `Response` interfaces are defined in Figure 4.8.*

Once the mediation service has validated and translated a message into the appropriate Java object, it passes it to the `Dispatcher`. The `Dispatcher` interface is the entry point for the dispatching API. It implements the business logic exposed by the `RoutingService`, implements the server reallocation algorithm in cooperation with the `ServiceHandler`s, and stores the `ServiceHandler`s (each instance represents the state of a queue). Such objects can be kept in different ways; the SPIRE implementation uses a concurrent hash table where the keys are the service names and the values are the `ServiceHandler`s. This approach has two main advantages compared to other alternatives, (i) it allows for inserting and searching entries in $O(1)$ time [34], while (ii) multiple threads can query and modify the hash table at the same time. `Qos` instances store the economic parameters of each service, as defined in Section 3.1. The `AllocationTask` is an asynchronous task used to run the allocation policy. It computes the new allocation vector, $(n_1,\ldots,n_m)$ and updates the n_i values in operation. The `AllocationPolicy` interface defines methods to change the way servers are allocated to service pools. The `computeAllocation()` operation uses an allocation algorithm to construct a new allocation vector at the end of each configuration window. Concrete classes implement the heuristic as well as the optimal allocation policies defined in Section 3.3 and 3.5. The `migrateServers()` operation is used to put the new n_i values in operation. Finally, `ServiceHandler` is by far the most important interface in the dispatching API as it represents a job type. It contains the queue storing the jobs waiting for execution, a profiler collecting the statistic information used to take resource allocation and admission control decisions, and the admission control module.

The remainder of this section discusses the use of the dispatching API to handle the four operations exposed by the mediation service and analyzes the computational cost of each of them.

Add node The argument of the method used to add new nodes is the machine's identifier (IP address and listening port), which is extracted by the mediation service from the incoming SOAP message (see Figure A.1). The new server is added to the pool of available machines, then the `Dispatcher` tries to allocate it to a service pool. The same method is invoked also every time a server is released from a service pool. Since the execution of client requests is non-preemptive, *i.e.* the execution of a client request cannot be interrupted, the switch of a server from a service pool to another one is generally not instantaneous. For this reason the `Dispatcher` keeps track of which job type needs at least one server, while each `ServiceHandler` keeps track of how

many servers are currently allocated and how many servers should be allocated to it. The value of the latter might differ depending on the type of allocation policy employed. If a conservative allocation policy is used, that value is n_i, while if a greedy algorithm is in operation, the number of servers that should be allocated to queue i is $min(n_i, p_i)$, where p_i is the number of 'pending jobs' for queue i (the number of pending jobs for a queue is defined is defined as the sum of the number of jobs queueing and the number of jobs in execution). As shown in Figure 4.12, if no queue needs the spare server, an aggressive method which queries all the queue states is used. Of course different metrics can be employed. SPIRE uses the current queue sizes and the n_i values to decide where to allocate the spare capacity provided by the extra server. If after this stage the node allocator is still unable to allocate the machine, the machine is temporarily 'allocated' to the spare pool. The cost of this operation is $O(m)$, where m is the number of provided services, because the `allocateIdleNode()` method visits all the `ServiceHandlers` in the worst case scenario (this method is declared as *protected* and thus it is not shown in Figure 4.11).

```
void allocateIdleNode(String node) {
    if (isRequiredEmpty()) {
        // apparently no service pool needs an extra server
        if (! tryAggressiveAllocation(node)) {
            addNodeToSparePool(node);
        }
    } else {
        // at least one service pool needs the machine
        allocateNodeToHandler(node);
    }
}
```

Figure 4.12: *Skeleton implementation of the* `allocateIdleNode` *operation.*

Each `ServiceHandler` has two logical pools, one containing the idle servers, the other storing the busy ones. Every time a new machine is added to queue i, the threshold K_i must be recomputed because of the increased capacity. Then, if the queue storing jobs waiting for execution is empty, the machine is added to the idle pool, otherwise one client request is forwarded to the new server using the request forwarding pipe, while the server is marked as busy (*i.e.*, added to the busy pool).

Add service is used to add a new service to the system. This method is invoked using the service name i and the demand parameters (c_i, q_i, r_i) as arguments (such values are extracted from

`addService` messages by the `RoutingService`, as discussed in Section 4.2.1), and allows the `Dispatcher` to instantiate `ServiceHandler` *i*. After this step, the system is ready to execute requests of type *i*. This operation is executed in constant time.

Queue checks whether a periodical system evaluation is needed, then passes the request to the appropriate `ServiceHandler`. The related activity diagram is shown in Figure 4.13.

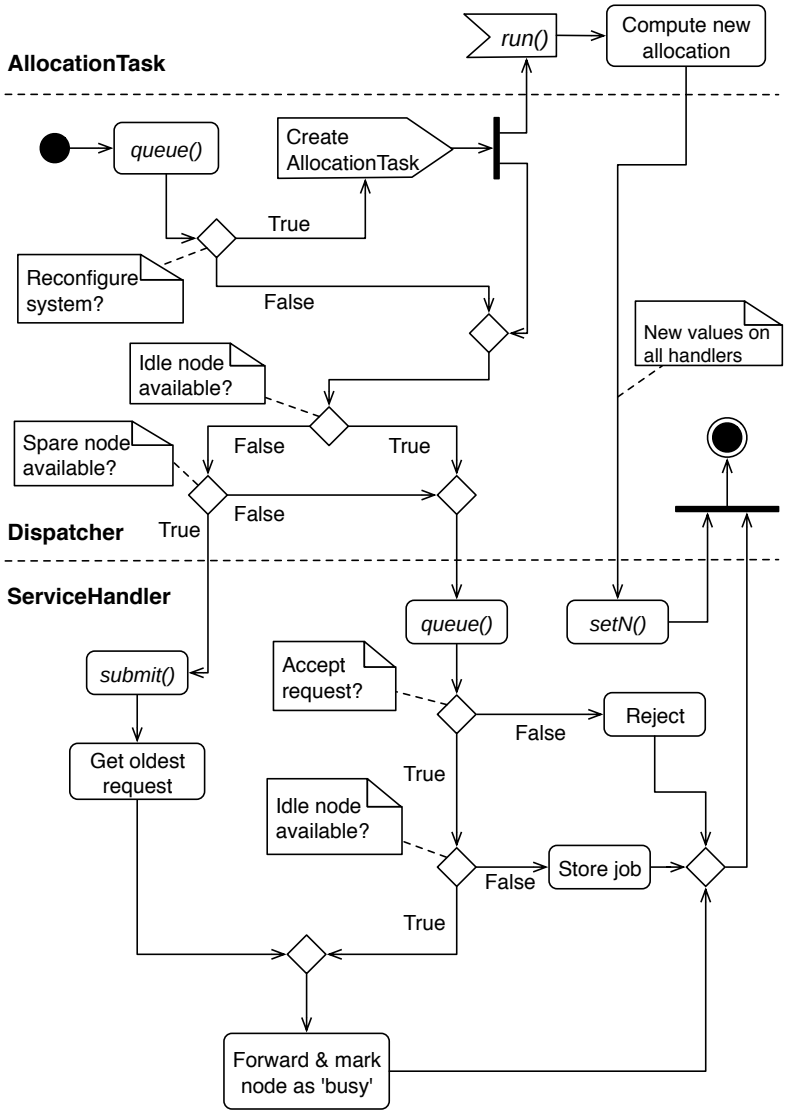


Figure 4.13: *Activity diagram for the queue event. When invoked on queue i, the* `setN()` *operation sets the new n_i value in operation and computes the new threshold, k_i.*

If the system needs to be reconfigured, a dedicated thread pool with a single worker executes

the allocation policy, as illustrated in Figure 4.14. Having a single worker thread running the allocation policies guarantees that consecutive invocations are executed sequentially; in other words, one allocation run does not start until the previous one has completed. The `AllocationTask` computes the new allocation vector $(n_1, \ldots, n_m)$ according to the allocation policy, then sets the new n_i and computes the new thresholds K_i. As mentioned above, the switch of a machine from one queue to another is instantaneous only if it sits idle: if a server is currently used, it will be reallocated at job completion. Since each `Request` object stores the target service name, identifying the `ServiceHandler` is trivial because, as described earlier in this section, the states of the queues are stored into a hash table. Once the target `ServiceHandler` has been retrieved, two possible scenarios are possible:

1. Queue i has at least one idle server, or there are no servers in the spare pool nor in the idle pool of queue i: the `queue()` method is invoked;

2. There are no servers in the idle pool of queue i, but the spare pool has at least one machine available: the `submit()` operation is used.

```
public void queue(Request request) throws RequestRejectedException,
    NoSuchServiceException {
    if (isEvaluationNeeded()) {
        ServiceHandler[] handlers = getServiceHandlers();
        int nodes = getRegisteredNodes();
        AllocationTask task = AllocationPolicyFactory.createAllocationTask(
        handlers, nodes);
        this.threadPool.execute(task);
    }

    ServiceHandler handler = getHandler(request);
    handler.queue(request);
}
```

Figure 4.14: *Skeleton implementation of the `Dispatcher.queue` operation. The `getHandler` method throws a `NoSuchServiceException` if the required service is not provided.*

The `queue()` method first evaluates whether the request should be rejected. If that is the case, the request is discarded and the client notified with an error message, otherwise (i) if there is a spare node available the request is forwarded immediately using the request forwarding pipeline (and the server is put into the busy pool), (ii) else it joins the queue. No matter what decision is

taken, the system updates the arrival rate information, while every time a `Request` is forwarded, a copy of it is saved on the controller. Even though fault tolerance support is beyond the scope of this dissertation, the latter provides a hook for employing a failure detection algorithm: in such a scenario, as soon as the system finds out that the server has crashed, it could forward the request somewhere else.

Unlike the `queue()` operation, `submit()` never rejects requests. Such method takes as second argument the identifier of the machine where the request is going to be forwarded. In order to maintain the guarantee that requests are handled in a FIFO order a special queue providing an atomic `queueIfNeeded()` method is used. Such operation requires a `Request` object as argument and returns the same object if the queue is empty, otherwise it returns the first element and queues the argument at the bottom. Once the job to execute is found, it is forwarded as before and the server is marked as busy. Queue events are handled in $O(1)$ time (excluding the cost of computing the new allocation and threshold vectors), given that waiting requests are stored in a linked list or a similar data structure.

Response updates the statistical information and eventually reallocates the server that executed the request. As shown by the activity diagram depicted in Figure 4.15, the `Dispatcher` retrieves the target `ServiceHandler` and invokes the `response()` method passing the response as argument. As for the `queue()` operation, retrieving the target handler is immediate because `Response` objects contain the name of the service.

The `ServiceHandler` first updates the average response and service times values, the charges and eventually the penalties, then it checks whether the server should be returned to the `Dispatcher` or not; as previously discussed, this decision depends not only on the current n_i and number of servers allocated to queue i, but also on the size of the queue, if a greedy policy is in operation. If that is the case, the machine identifier is removed from the busy-pool and the `releaseNode()` method is invoked (and eventually the server is allocated to a different pool). If instead the machine should be kept in the same service pool, (i) if there is at least one request waiting into the queue, the oldest request is executed, otherwise (ii) the server identifier is removed from the busy pool and stored into the idle pool. In the current implementation responses are handled in constant time, excluding the server re-allocation cost (if any), that requires $O(m)$ time in the worst case scenario (see the discussion at the beginning of this section).

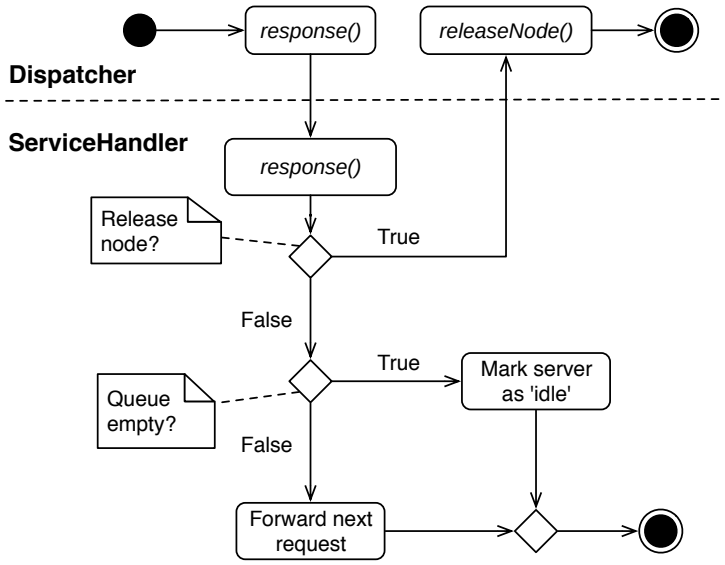


Figure 4.15: *Activity diagram for the response event. The* `ServiceHandler.response()` *operation updates the statistic information, while the* `Dispatcher.releaseNode()` *method tries to reallocate the server.*

4.2.3 Dynamic Service Deployment and Request Execution

The architecture of the server machines is quite simple because it does not include any 'intelligence'. As in the case of the load balancer, the idea is to use some custom handlers to perform all the non-standard tasks: the input chain includes two custom interceptors to save some state information and to deploy the target service, if needed, while the output chain uses one custom handler to restore the saved values as well as to compute the service time. Figure 4.16 shows the actions taken every time a job is going to be executed.

The first handler of the input chain determines whether the target service is already deployed or if it needs to be fetched from a remote repository. This has been implemented in two versions, as a TCP server and as a web service using the Message Transmission Optimization Mechanism (MTOM) [113] (of course, more complex solutions can be employed too). If the target service is not available in the Axis2 configuration, the handler gets the service archive from the repository

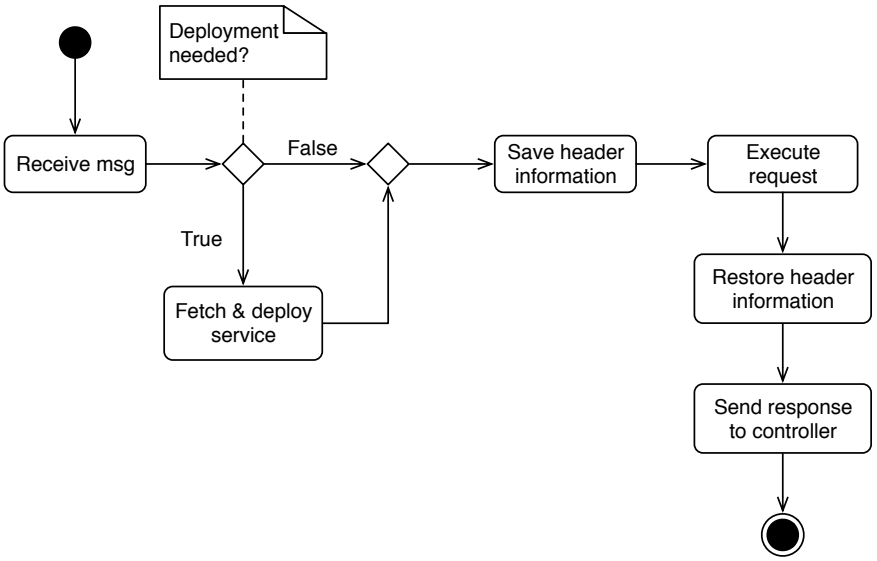


Figure 4.16: *Request execution: activity diagram.*

and deploys it, *i.e.*, copies the service archive into the folder where deployed services are stored and waits until the engine updates its internal configuration (a simple polling mechanism is used). This requires between 5 and 10 seconds plus the time needed to copy the file across the network.

At the end of the input handler's chain comes the second custom interceptor, used to save the state information. Unlike what happens in the load balancer, where the routing values are saved into the `MessageContext`, in this case the state must be saved into the `OperationContext` as the input `MesasgeContext` is valid only during the input flow (the `OperationContext`, instead, is the same for input as well as the output flow). The saved informations include the *qosp* block, whose structure is shown in Figure A.4 (page 145). This embodies the information saved into the `Request` object, as defined in Figure 4.8.

After the request is executed, the custom interceptor running into the output chain adds to the response SOAP header the state information previously saved, together with the service time and a boolean value indicating whether a deployment request was issued to the repository (see the *qosp* block in Figure A.3). Finally, the response is sent back to the controller.

4.3 Summary

This chapter has presented the architecture of SPIRE, a data center management system designed with a utility computing paradigm in mind. Central to the system is the controller (which can be replicated for availability and scalability purposes), a Layer-7 two-way load balancer implementing the allocation and admission policies introduced in Chapter 3. Settings such as the use of a conservative or greedy allocation policy or the use of the admission control are configurable at runtime via Java Management Extensions (JMX) [95]. Next chapter demonstrates the use of the presented framework to optimize the performance of a data center, while Chapter 7 extends SPIRE in order to deal with streams of requests.

Chapter 5

Experiments with Single Jobs

This chapter describes the experiments carried out on SPIRE in order to evaluate the effects of the server allocation and job admission policies described in Chapter 3. As in Section 3.7, in order to reduce the number of variables, the following values are kept fixed:

- The QoS metric is the response time, W;
- The obligations undertaken by the provider are that jobs will complete within twice their average required service time, that is, $q_i = 2/\mu_i$;
- All penalties are equal to the corresponding charges, *i.e.* $r_i = c_i$. In other words, if the response time exceeds the obligation, users get their money back;
- A 20-server system is used. The cluster is composed of machines running Linux version 2.6.14 and Sun JDK 1.5.0_04. Axis2 version 1.1 is deployed on the Tomcat servlet engine [101]. The used version was 5.5.

Unlike the model validation of Chapter 3, experiments described here were carried out on the SPIRE system: CPU-bound jobs were generated, queued and executed on real servers; messages were sent and delivered by a real network. Apart from the network delays, the main difference between the simulations discussed in Chapter 3 and the experiments that will be introduced shortly is that messages are subject to random processing overheads, which cannot be controlled. Also, it could not be guaranteed that the computers were dedicated to these tasks; there could be random demands from other users. The connection between the load generator and the controller is provided by a 100 Mb/sec Ethernet network, while the servers of the cluster are connected to the controller via a 1 Gb/sec Ethernet network. The average round trip time (RTT) between nodes and the controller

is 0.258 ms, while the one between the client and the controller is 0.558 ms. Nevertheless, since both the servers and the network are shared, unpredictable delays due to other users are possible.

Random interarrival intervals were generated by client processes while service times were randomly generated at the server nodes. A variety of parameter values were tried, in order to subject the system to different loading conditions. Also, different distributions of job lengths and interarrival times were introduced, in order to test the robustness of the proposed algorithms. Finally, except when indicated explicitly, the 'conservative' versions of the server allocation policies are used (see Section 3.3).

5.1 Performance when Exponentiality Assumptions are Satisfied

The first experiment is similar to the simulation illustrated in Figure 3.6. Two service types were deployed. The three pairs of arrival rates used in this experiment were the same as in figure 3.6: $\lambda_1 = \lambda_2 = 0.75$; $\lambda_1 = 0.5$, $\lambda_2 = 1.0$; and $\lambda_1 = 0.2$, $\lambda_2 = 1.3$. The average service times were also the same: $1/\mu_1 = 1/\mu_2 = 10$. However, because of the factors mentioned above, one should not expect a precise match between the numerical predictions and the observations of the real system.

In Figure 5.1, the total revenues earned are plotted against n_2, the number of servers allocated to service 2. Each point in the figure corresponds to a separate run of the system, during which about 1,000 jobs of each type arrived and were completed. These plots have the same general characteristics as the ones in Figure 3.6. The maximum achievable revenue is again about 130 per unit time, and the server allocations that achieve it are the same.

The next experiment involves a comparison between static and dynamic configuration policies. The parameters are the same as for Figure 3.7: a 20 server system with two job types, the arrival rate for type 1 is kept fixed at $\lambda_1 = 0.2$, while λ_2 takes different values, ranging from 0.4 to 1.8. The average service times for the two types are $1/\mu_1 = 50$ and $1/\mu_2 = 5$ respectively (*i.e.*, type 1 jobs are on the average 10 times longer than type 2). Thus, the offered load of type 1 is 10, while that of type 2 varies between 2 and 9; the total system load varies between 60% and 95%. The charges for the two types are equal: $c_1 = c_2 = 100$.

Again, each point is obtained from a separate run of the system. Under the 'Static Oracle' policy, which somehow knows the values of all parameters, at time 0 of each run, the Measured Loads heuristic is used to compute the server allocations n_1 and n_2, and the admission thresholds

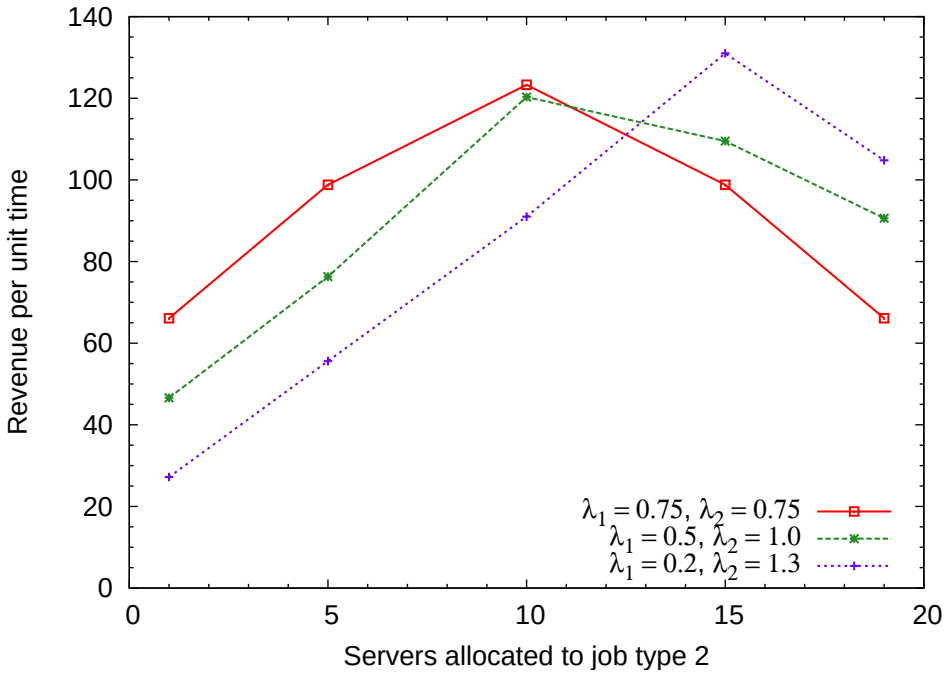


Figure 5.1: *Observed revenues for different server allocations. As in Figure 3.6, when the demand is asymmetric it is more profitable to allocate more servers to the more heavily loaded queue.* $N = 20, m = 2, \mu_i = 0.1, c_i = r_i = 100$.

K_1 and K_2; thereafter, the configuration remains unchanged. Obviously, the Static Oracle policy could not be used in practice, since the demand parameters are not usually known in advance; it is included here only for purposes of comparison.

The dynamic allocation policy divides a run into configuration intervals called 'windows', such that a total of J jobs (of all types) arrive during a window. Demand statistics are collected, providing new estimates for λ_i and μ_i ($i = 1, 2, \ldots, m$) by the end of each window. Those estimates are used to allocate servers to job types according to the Measured Load heuristic described by equation (3.4). Reallocations of busy servers take place upon service completions (*i.e.*, job services are not interrupted). The new allocations remain valid for the duration of the next window. Whenever the number of servers allocated to type i changes, the optimal admission threshold K_i is recomputed. At time 0, before any statistics have been collected, servers are allocated in a FIFO order (that is, to the job type which needs them) and an arbitrary admission policy (*e.g.*, $K_i = \infty$) is adopted. Clearly, some such dynamic policy would have to be used in practice, since (i) the demand parameters are not usually known in advance and (ii) those parameters may change with time.

The observed revenues obtained by the static and dynamic heuristics are illustrated in Figure 5.2. Three different window sizes were tried, containing $J = 30$, $J = 60$ and $J = 90$ jobs, respectively.

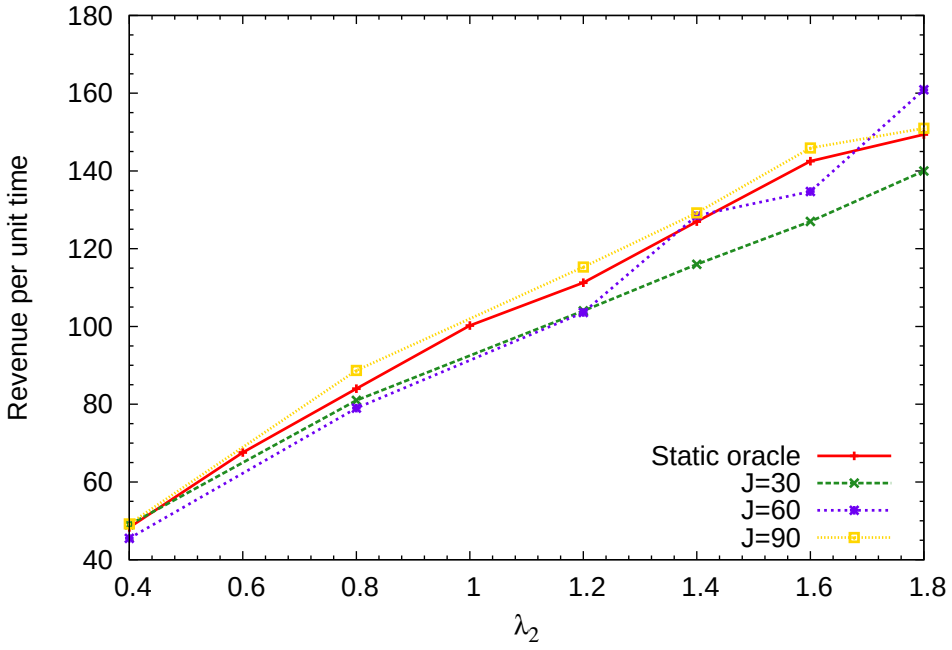


Figure 5.2: *Observed revenues for static and dynamic Measured Load heuristic.* $N = 20, m = 2, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$.

The most notable feature of the figure is the close agreement between the observed revenues and those predicted by the model (Figure 3.7). This increases our confidence in the practical applicability of the proposed server allocation heuristic. Moreover, the dynamic policies are able to estimate and use the current values of the demand parameters. The effect of the different window sizes is not very pronounced. The shortest windows ($J = 30$) probably do not produce sufficiently accurate estimates, and therefore have the worse performance. The $J = 60$ and $J = 90$ windows are better, the latter consistently out-performing the static policy.

The following four experiments use the same settings as before and compare three operating policies:

1. The Static Oracle policy;
2. The dynamic Measured Loads heuristic;
3. The dynamic Measured Queues heuristic.

As in the previous experiment, for both dynamic heuristics, at the start of a run, before any statistics have been collected, servers are allocated on demand as jobs arrive; the initial admission policy is to accept all jobs (*i.e.*, $K_i = \infty$).

The average revenues obtained per unit time are plotted against the arrival rate for type 2, λ_2. Each point represents a separate system run, during which at least 1,000 jobs of type 1 are processed (the number of type 2 jobs is larger, and increases with λ_2). In the case of the Measured Load heuristic, the corresponding 95% confidence intervals are also shown. They confirm the well-known observation that the confidence intervals tend to grow with the load. Those of the other policies are of similar size and are omitted in order not to clutter the graphs.

Figures 5.3, 5.4, 5.5 and 5.6 illustrate the behavior of the system for four different window sizes, $J = 30$, $J = 60$, $J = 90$ and $J = 150$.

When the window size is small, the dynamic Measured Loads heuristic performs slightly worse than the other two policies. This is probably due to the fact that the arrival and service rate estimates obtained during each window are not sufficiently accurate. The Measured Queues heuristic appears to react better to random changes in load.

The effect of increasing the window size is not very pronounced. The Measured Loads heuristic is now, if anything, slightly better than the Measured Queues heuristic, and both outperform the Static Oracle policy. However, the differences are not statistically significant. Corresponding points are mostly within each other's confidence intervals.

It is perhaps worth mentioning that the relative insensitivity of the system performance with respect to the window size is, from a practical point of view, a very good feature. It means that the decision of what window size to choose is not too critical.

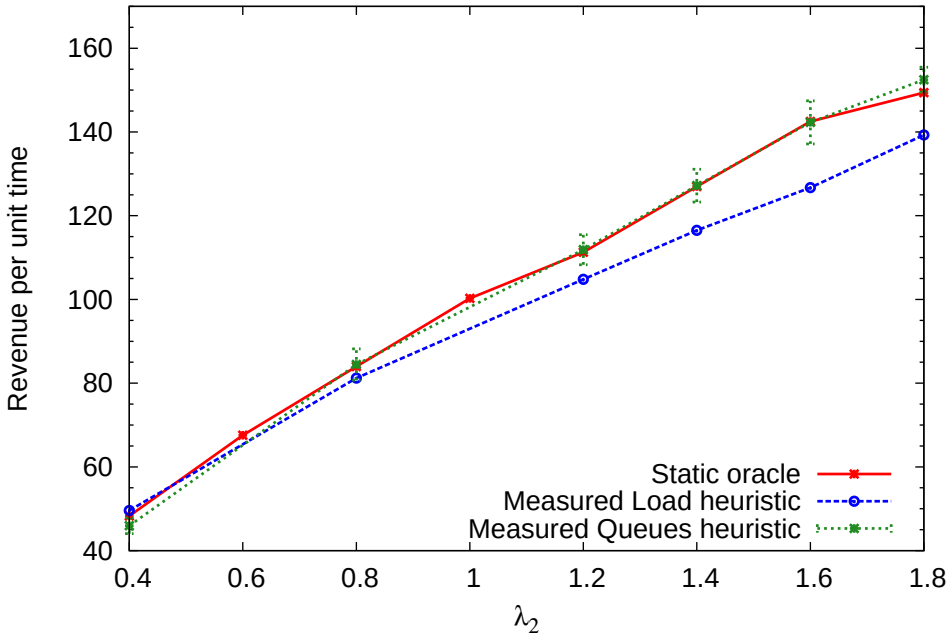


Figure 5.3: *Dynamic policies in SPIRE: window size 30 jobs.* $N = 20, m = 2, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$

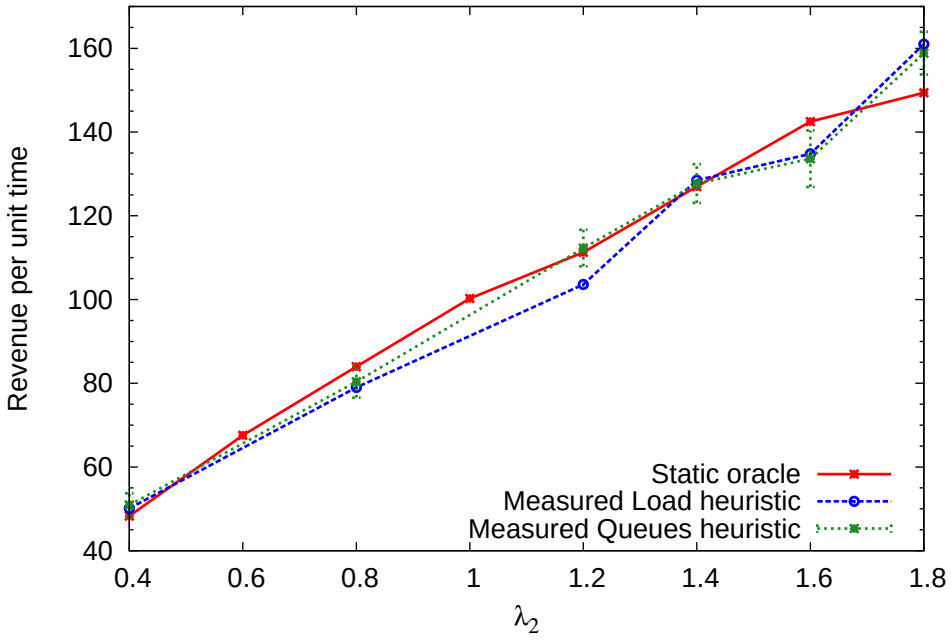


Figure 5.4: *Dynamic policies in SPIRE: window size 60 jobs.* $N = 20, m = 2, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$

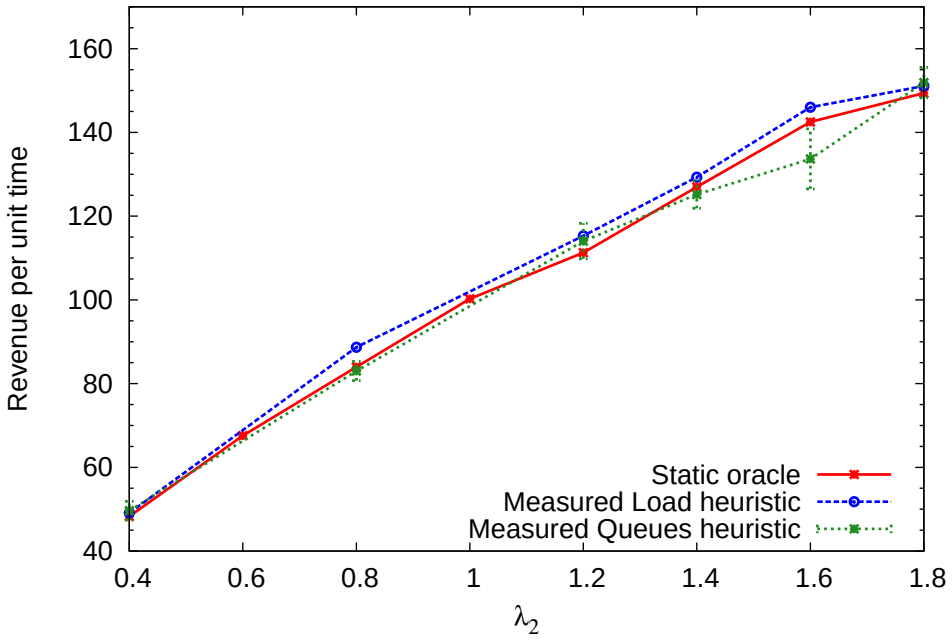


Figure 5.5: *Dynamic policies in SPIRE: window size 90 jobs.* $N = 20, m = 2, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$

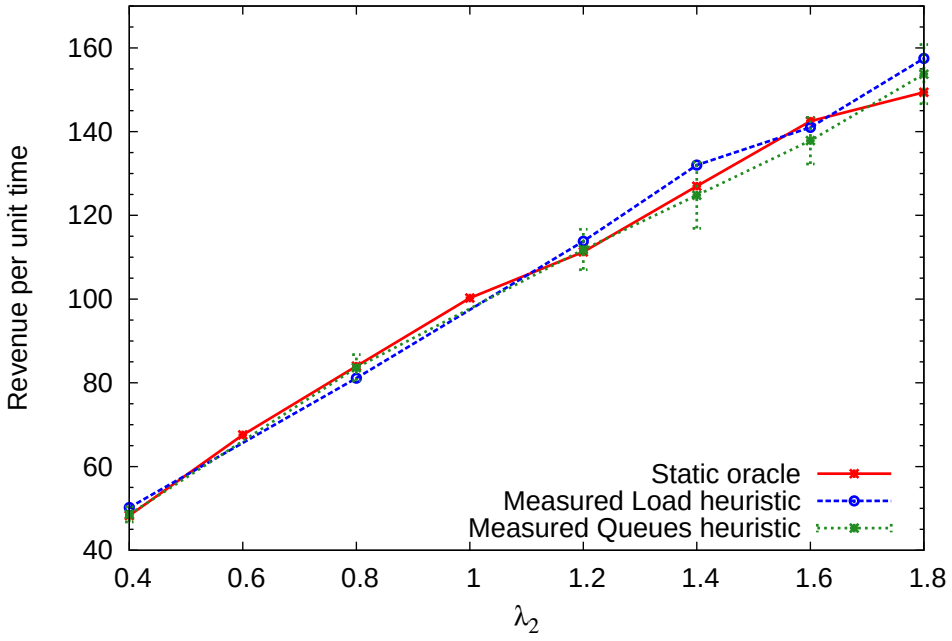


Figure 5.6: *Dynamic policies in SPIRE: window size 150 jobs.* $N = 20, m = 2, \lambda_1 = 0.2, \mu_1 = 0.02, \mu_2 = 0.2, c_i = r_i = 100$

5.2 Performance without Exponentiality Assumptions: Bursty Arrivals

In the next experiments, the arrival process of type 2 is no longer Poisson, but consists of bursts. More precisely, the observation period is divided into alternating periods of lengths 180 seconds and 60 seconds, respectively. During the longer periods, $\lambda_2 = 0.4$ jobs/sec, while during the shorter periods it is about 5 times higher, $\lambda_2 = 1.8$ jobs/sec. The other demand parameters are the same as before.

Figure 5.7 illustrates the revenues obtained by the Measured Loads and Measured Queues heuristics, for three different window sizes. For purposes of comparison, an Oracle policy which knows the values of all parameters and makes optimal allocations at the beginning of each period was also evaluated. Its performance is of course independent of the window size.

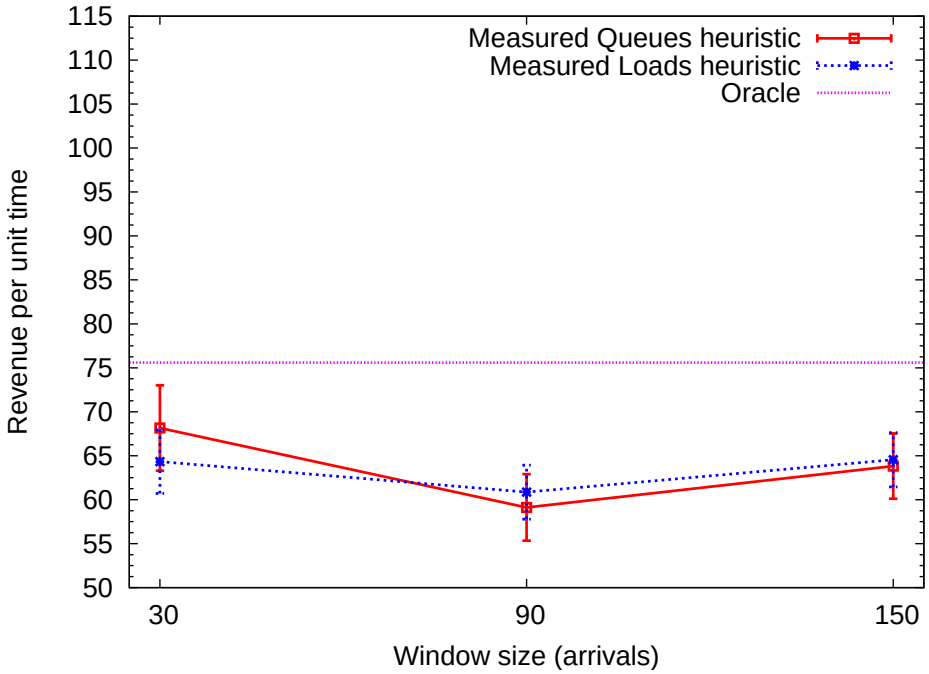


Figure 5.7: *Bursty arrivals: different window sizes; equal charges. $\lambda_2 = 0.4$ during the longer periods and 1.8 during the shorter ones.*

Because of the frequent changes in user demand, shorter windows are now better than long ones. The two heuristic policies perform similarly, with the values inside each other confidence interval. The revenues achieved by both heuristics are within about 20% of the ideal (and unrealizable) revenue of the Oracle policy.

To examine the effect of applying different charges to different services, the experiment was repeated with the same demand parameters, but doubling the charge (and the penalty) for type 1 jobs: $c_1 = 200$, $c_2 = 100$. Note that this change might affect the allocation heuristics since $\alpha_i = c_i$. The results are shown in Figure 5.8.

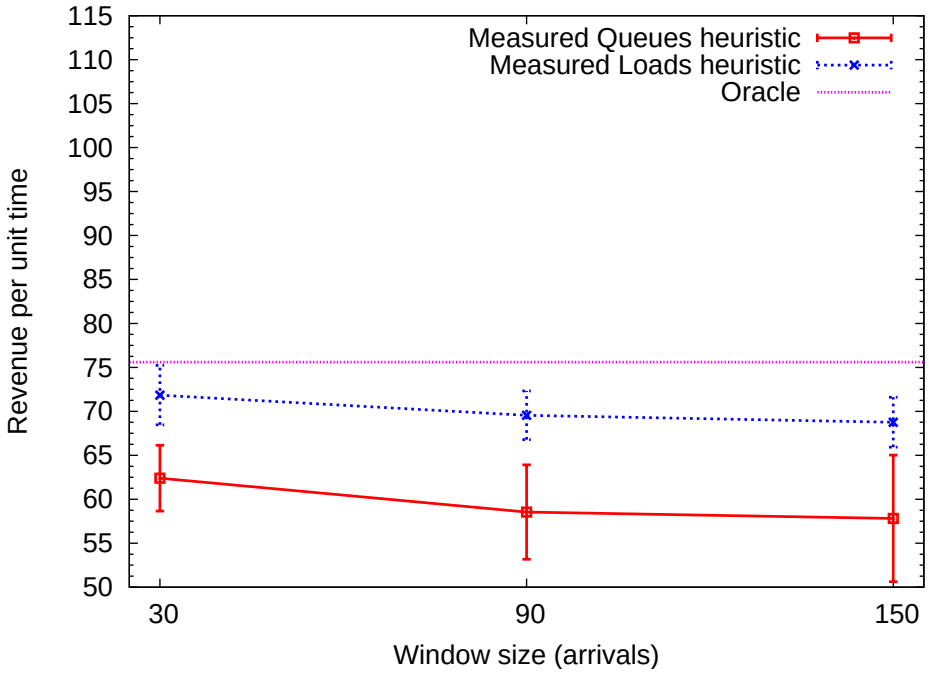


Figure 5.8: *Bursty arrivals: different window sizes; unequal charges, $\alpha_i = c_i$. $\lambda_2 = 0.4$ during the longer periods and 1.8 during the shorter ones.*

The observed behavior is now different: the Measured Load heuristic consistently outperforms the other policy, and there is not much difference in using short configuration intervals. Finally, the revenues achieved by both policies are within about 25% of the Oracle's revenues.

The differences in observed revenue can be better understood from Figure 5.9. The figure shows the number of servers executing type 2 jobs over time by changing the window size, J, and the allocation policy. As already discussed the dependence between the allocation decisions and the maximum achievable revenue is very tight since the threshold K_i is computed every time the number of servers allocated to a queue changes.

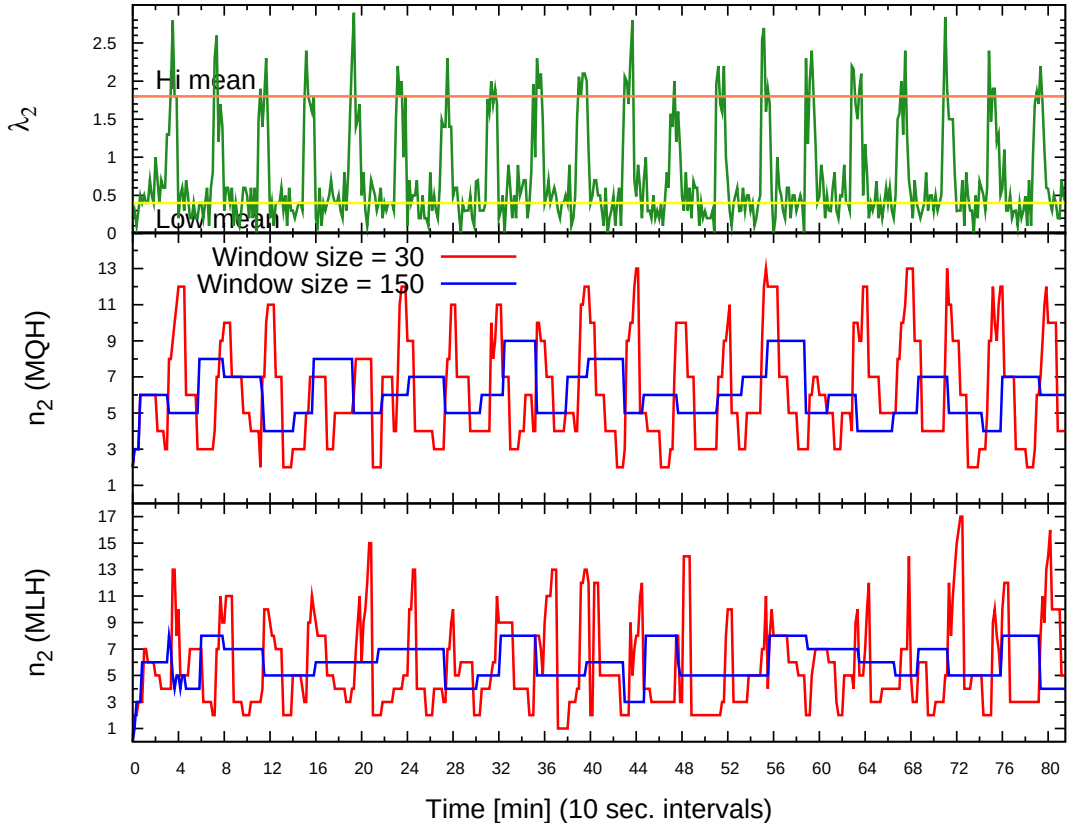


Figure 5.9: *Bursty arrivals: arrival rate versus server allocation. The first part of the figure illustrates the arrival rate for type 2 jobs, λ_2, the second part shows different allocation decisions taken by the Measured Queues heuristic and the last part shows the number of servers allocated to queue 2 by the Measured Loads heuristic.*

The same experiments were repeated by changing the rate at which type 2 jobs arrive during busy periods. During the longer periods, $\lambda_2 = 0.4$ jobs/sec (as before), while during the shorter periods it is about 12 times higher, $\lambda_2 = 5$ jobs/sec. The other demand parameters are the same as before. Note that if the higher arrival rate was maintained throughout, the system would be saturated.

As might be expected in view of the highly variable demand, shorter windows are now significantly better than long ones. It is also interesting to note that the Measured Loads heuristic consistently outperforms the Measured Queues one. When the charges are the same for both the job types, as in Figure 5.10, the differences are not large. The largest difference between the two heuristics is about 5%; the largest loss of revenue due to using a window whose size is too large is about 20%. Moreover, the revenues achieved by both heuristics are within about 25% of the

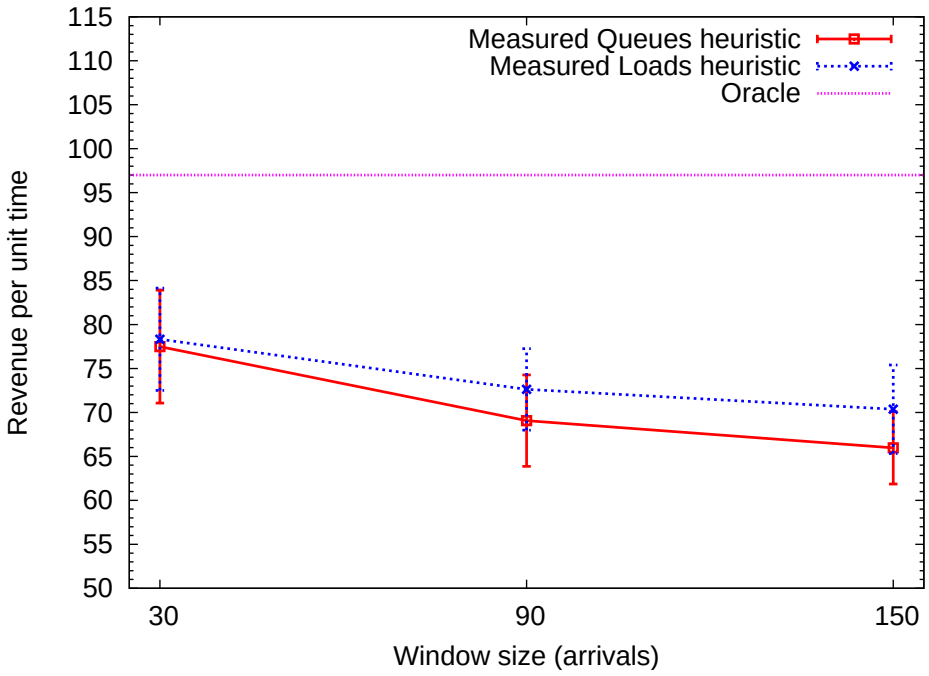


Figure 5.10: *Bursty arrivals: different window sizes; equal charges.* $\lambda_2 = 0.4$ *during the longer periods and 5.0 during the shorter ones.*

Oracle's revenue. In the second case, however, the difference is bigger, with the Oracle performing about 35% better than the Measured Queues heuristic.

Finally, the same experiment was run by changing the α_i parameters from c_i to $r_i/c_i = 1$. This change does not affect the allocation heuristics compared to the case when all charges are the same, but it may affect the admission policies.

The obtained behavior is similar to that in Figure 5.10. Figure 5.12 shows that revenues are higher, in line with the increased charges for type 1 jobs. Again, shorter windows are better than longer ones, but not by much. It is a little surprising that the Measured Queues heuristic now appears to outperform the Measured Loads one. However, the differences are rather small; the corresponding points are well within each other's confidence intervals. The revenues achieved by both policies are within about 30% of the Oracle's revenues.

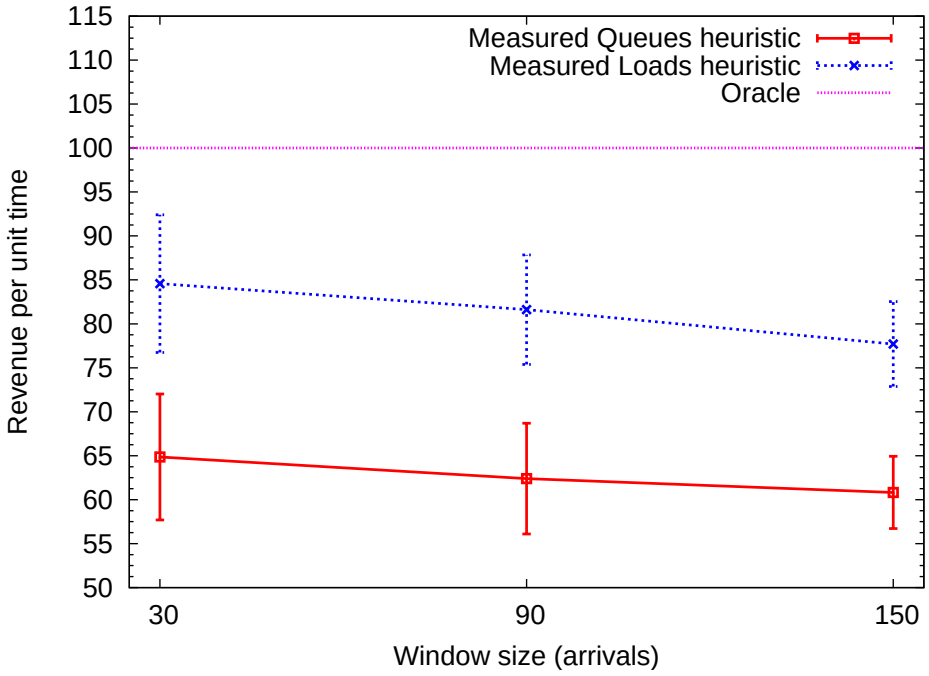


Figure 5.11: *Bursty arrivals: different window sizes; unequal charges,* $\alpha_i = c_i$. $\lambda_2 = 0.4$ *during the longer periods and 5.0 during the shorter ones.*

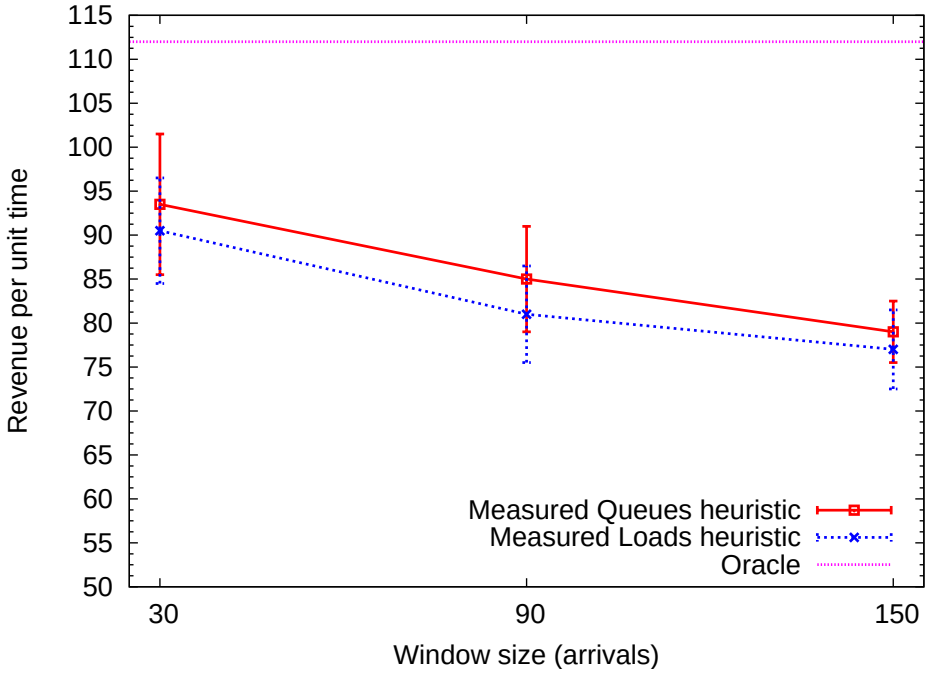


Figure 5.12: *Bursty arrivals: different window sizes; unequal charges,* $\alpha_i = 1$. $\lambda_2 = 0.4$ *during the longer periods and 5.0 during the shorter ones.*

It is interesting to observe the effect of replacing the conservative versions of the allocation policies with the greedy ones (*i.e.*, unused servers are temporarily reallocated to other pools before the end of the current window).

Figure 5.13 and 5.14 show that there is a marked improvement in the performance of both heuristics. The revenues they achieve are now within about 10% of those of the Oracle policy. More importantly, both policies are now even less sensitive with respect to the window size. The relative differences between the best and the worst average revenues are less than 5% (the confidence intervals are quite large, but that is explained by the variability of demand).

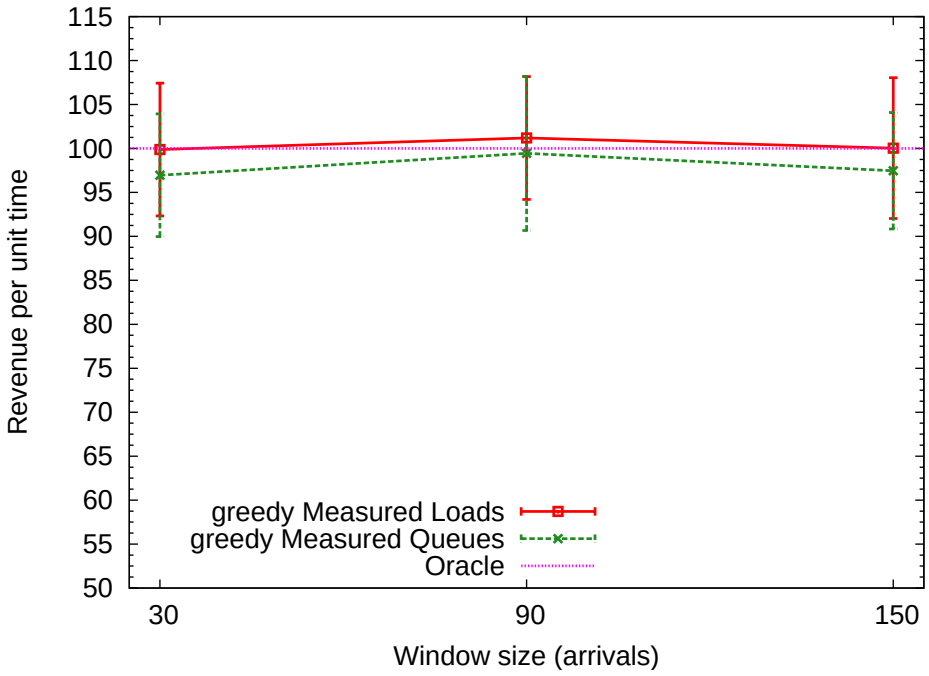


Figure 5.13: *Greedy policies, bursty arrivals: unequal charges, $\alpha_i = c_i$. Other parameters as in Figure 5.12.*

The reason why the greedy versions of the heuristics are less affected by the window size is that a 'wrong' server allocation can be adjusted as soon as its symptoms appear, without waiting for the end of the window.

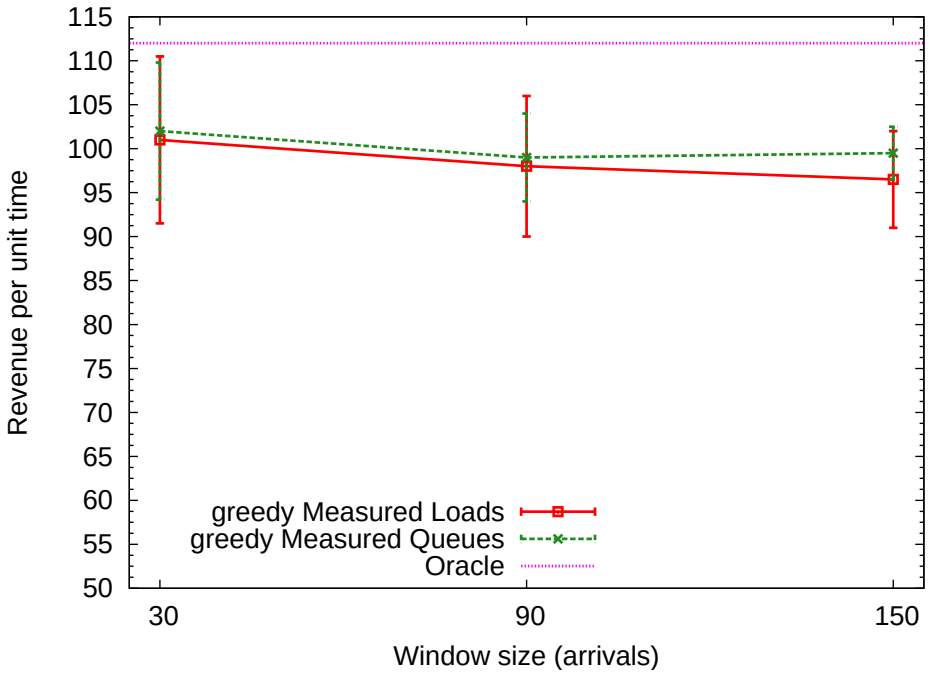


Figure 5.14: *Greedy policies, bursty arrivals: unequal charges,* $\alpha_i = 1$. *Other parameters as in Figure 5.13.*

Other performance metrics In some cases, values other than the average revenue per unit time might be of interest. A possible example is the rate at which type i jobs are rejected. For the conservative heuristics that rate, X_i, is given by:

$$X_i = \lambda_i p_{i,K_i} \tag{5.1}$$

where p_{i,K_i} is the probability that a type i request is rejected. There is no mathematical formula for the greedy versions. The trace illustrated in Figure 5.15 compares the reject rate of the conservative and greedy Measured Loads heuristic as well as the ideal and unrealizable rate at which the Oracle policy would reject jobs. It is clearly visible that, during peaks, the greedy implementation rejects less jobs, and thus it can achieve higher revenues.

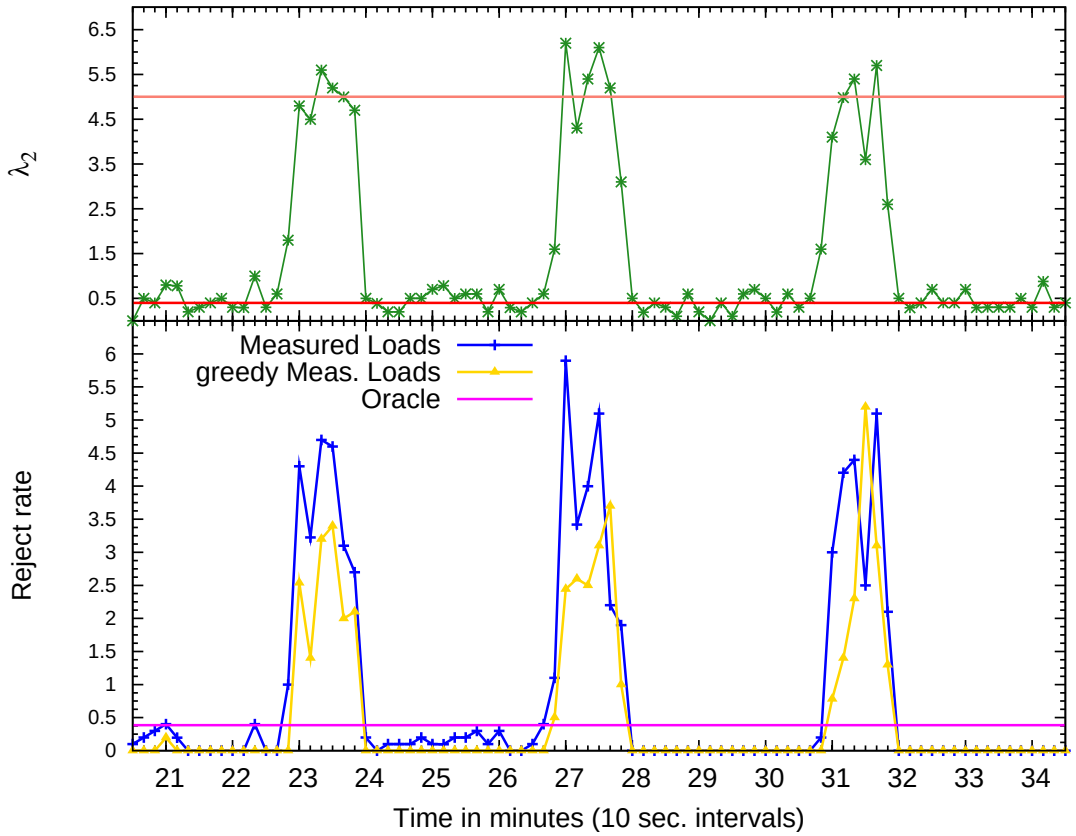


Figure 5.15: *Bursty arrivals: reject rate comparison for the conservative and greedy implementations of the Measured Loads heuristic and the oracle.*

5.3 Performance without Exponentiality Assumptions: Hyperexponential Distributed Service Times

The next experiment departs from the assumption that all service times are distributed exponentially. Now the service time distributions are two-phase hyperexponential. Type 1 jobs have mean 32.9 with probability 0.7 and mean 90 with probability 0.3; the overall average service time is 50 seconds, as before. The corresponding parameters for type 2 are mean 3.71 with probability 0.7, and mean 8 with probability 0.3; the overall average service time is 5 seconds. These changes increase the variances of the service times to about 6, while preserving the averages. In this experiment the greedy heuristics have been used because they have demonstrated to consistently outperform their conservative counterparts. Since the window size is not important when the greedy policies are in

operation, only a window size of 150 jobs was used. Note that the Measured Loads heuristic for allocating servers is not affected significantly by the service time distribution, since it is based on averages only. However, one might expect that the admission decisions will now be 'wrong' more often, since the thresholds are still calculated assuming exponentially distributed service times, and that the revenues will drop as a consequence.

In fact, Figure 5.16 shows that this is not the case. Not only have the revenues not decreased, compared with those obtained when the distribution is exponential, but they have increased slightly. In other words, the procedures for making admission decisions under both heuristics are sufficiently robust to cope with violations of the distributional assumptions. The slight increases in revenue are probably explained by a reduction in the penalties paid because most jobs are shorter.

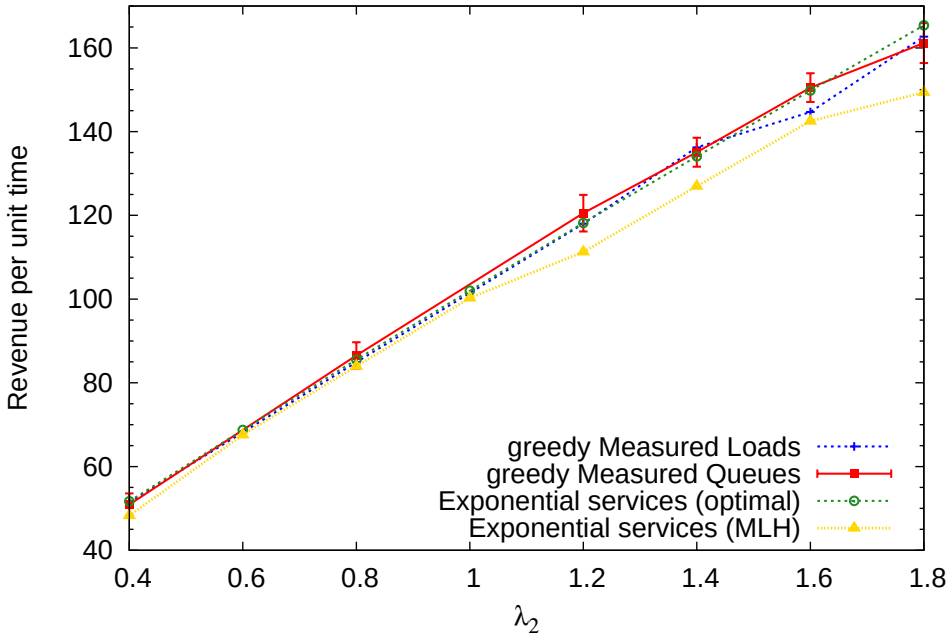


Figure 5.16: *Hyperexponential service time distribution.* $N = 20, \lambda_1 = 0.2, c_1 = c_2 = 100$

5.4 Optimal Policy Evaluation

The aim of the final experiment of this chapter is to evaluate the extent to which revenues can be improved by carrying out the dynamic optimization described in Section 3.5, rather than using a heuristic allocation policy. The usual 20-server cluster now offers 3 types of services, with average service times $1/\mu_1 = 40$, $1/\mu_2 = 10$ and $1/\mu_3 = 4$, respectively. The arrival rates for types 1 and 2 are fixed, at $\lambda_1 = 0.175$ and $\lambda_2 = 0.6$, while that of type 3 increases from $\lambda_3 = 0.25$ to $\lambda_3 = 1.5$. Thus, the total offered load is increased from moderate, 14, to heavy, 19. The charge/penalty values differ between the three types: $c_1 = r_1 = 500$, $c_2 = r_2 = 250$, $c_3 = r_3 = 100$. Finally, the system reallocates servers every 90 arrivals, *i.e.*, $J = 90$.

The 'Optimal' policy estimates the demand parameters during each window, and uses the fast search algorithm of Section 3.5 to find the optimal allocations and thresholds that will apply during the next window. The resulting revenues are compared, in Figure 5.17, with those of the Measured Loads heuristic which in this case allocates servers in proportion to the observed loads λ_i/μ_i (and then computes the best thresholds, K_i).

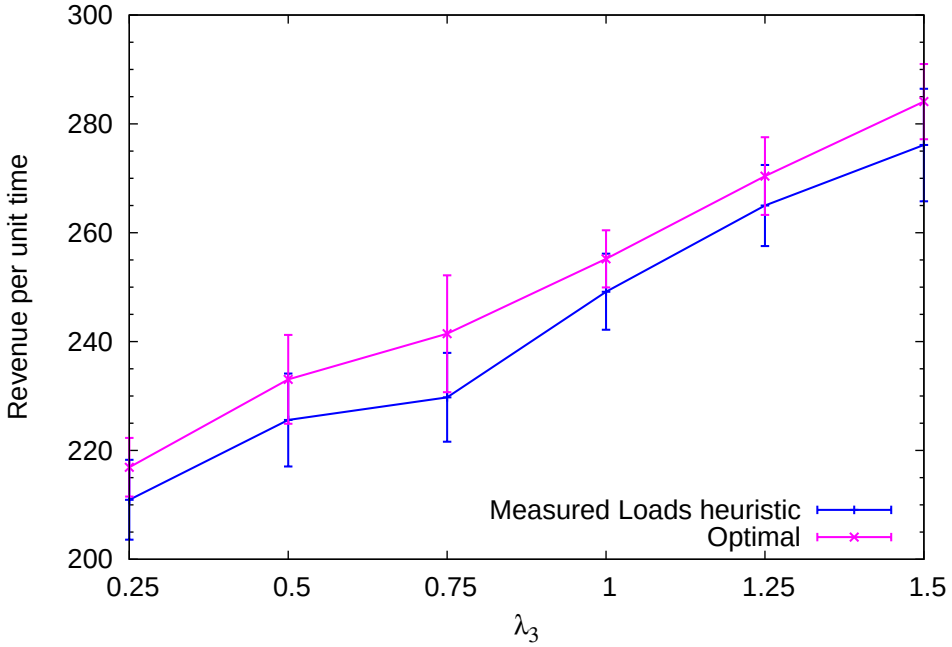


Figure 5.17: *Three job types: optimal and heuristic policies.* $N = 20, \rho_1 = 7, \rho_2 = 6, \rho_3 \in [1,6]$, $J = 90$.

This experiment confirms that it is feasible to search for the optimal configuration on-line: the fast search algorithm took about 30 milliseconds at the end of each window. That policy does yield consistently higher revenues than the heuristic. On the other hand, we also note that the relative improvement is not very great; it is on the order of 5% or less. One could therefore interpret these results also as a confirmation that the Measured Loads heuristic is a pretty good policy for the control of a service provisioning system.

5.5 Summary

This chapter has evaluated the dynamic policies for server allocation and job admission that were introduced in Chapter 3 using the SPIRE system discussed in Chapter 4. These algorithms have been shown not only to perform well (there is some evidence that they are close to the optimal), but also to be robust. An important feature from a practical point of view is that these heuristics are not very sensitive with respect to the window size that is used in their implementation. Moreover, they are able to cope with bursty arrivals and non-exponential service times.

The Measured Queues heuristic appears to offer no significant performance advantage, compared to the Measured Loads heuristic. Since the estimation of arrival rates and average service times is in any case necessary for the computation of admission thresholds, one could ignore the queue sizes and operate just the Measured Loads heuristic.

Chapter 6

Admission Policies for Service Streams

The model introduced in Chapter 3 applies charges, obligations, penalties and admission policies to individual jobs. The major drawback of that model is that service providers do not usually make QoS promises to every job they accept. Moreover, if a user is trying to execute several jobs, he/she would not know in advance which job will be accepted and which one will not. Again, in E-business systems such as Amazon or eBay, requests coming from the same customer are related and thus they can be grouped into sessions[1]. If an admission control model like the one discussed in Chapter 3 is in operation and the system is heavily loaded, only some requests would be accepted for processing (*i.e.*, there would be a large number of broken or incomplete sessions). Since requests are essentially dropped at random, all clients connecting to the highly-loaded system would be likely to experience connection failures, even though there may be enough capacity on the system to serve all requests properly for a subset of clients. Finally, since active sessions can be aborted at any time, this could lead to an inefficient use of resources because aborted sessions do not perform any useful work, but they 'waste' system resources.

For the reasons mentioned above, the previous model is now generalized in order to deal with streams of requests. The notion of 'stream' is defined in the next section and, after that, the words 'session' and 'stream' will be used interchangeably.

The aim of this chapter is to propose and evaluate efficient and easily implementable admission heuristics for service streams that aim to maximize the average revenue received per unit time. The core idea is the same as before, that is, customers still pay to use computer or storage resources, however now SLAs apply to job streams, not single requests. The proposed policies are based on *(i)* dynamic estimates of traffic parameters, and *(ii)* models of system behavior. The emphasis of

[1]For the sake of simplicity, a session can be defined as a semi-permanent interaction. Such a conversation can be stateful as well as stateless.

the latter is on generality rather than analytical tractability. Thus, interarrival and service times do not have to be exponentially distributed. Instead, they are allowed to have general distributions with finite coefficients of variation. To handle the resulting models, it is necessary to use approximations. However those approximations should lead to policies that perform well and can be used in real systems.

6.1 The Model

The provider has a cluster of N identical computers, which are used to offer m different services numbered $1, 2, ..., m$. A user request for service i is referred to as a 'stream of type i'.

Definition 6.1 (Stream)**.** *A stream of type i is a collection of k_i jobs, submitted at the rate γ_i jobs per second.*

If a stream is accepted, all jobs in it will be executed, *i.e.* session integrity is required. Session integrity is a critical metric for commercial web services. From a business perspective, the higher the number of completed streams, the higher the revenue is likely to be, while the same does not apply to single jobs. Apart from the penalties resulting from the failure to meet the promised QoS standards, streams that are broken or delayed at some critical stages, such as checkout, could mean loss of revenue for the service owners. From a customer's point of view, instead, breaking session integrity would generate a lot of frustration because the provided service would appear as not reliable [30].

A stream which has been accepted but not yet completed is said to be 'currently active'. If L_i streams of type i are currently active, then the total current arrival rate of type i jobs is $\lambda_i = L_i \gamma_i$.

Throughout this chapter, empirical distributions will be characterized using two metrics, the mean and the squared coefficient of variation. The squared coefficient of variation of a distribution is a measure of its variability and it is defined as the variance divided by the squared mean:

$$cv^2 = \frac{\sigma^2}{\mu^2} \tag{6.1}$$

The advantage of using the squared coefficient of variation as a measure of variability, rather than the variance or the standard deviation, is that it is normalized by the mean, and so allows comparison of variability across distributions with different means.

The service times of type i jobs are independent and identically distributed random variables (i.i.d.), that is, each has the same probability distribution as the others and all are mutually inde-

pendent. The average service time and squared coefficient of variation are b_i ($b_i = 1/\mu_i$) and cs_i^2 respectively, while the squared coefficient of variation of the interarrival intervals is ca_i^2. Thus, the demand of type i when L_i streams are active is characterized by the 4-tuple:

$$(\lambda_i, ca_i^2, b_i, cs_i^2), \qquad i = 1, 2, \ldots, m. \tag{6.2}$$

The Quality of Service experienced by an accepted stream of type i is measured by the observed average waiting time, W_i:

$$W_i = \frac{1}{k_i} \sum_{j=1}^{k_i} w_j, \tag{6.3}$$

where w_j is the waiting time of the jth job in the stream (the interval between its arrival and the start of its service). It is worth emphasizing that the right-hand side of equation (6.3) is a random variable; its value depends on every job that belongs to the stream. Hence, even if all interarrival and service times are distributed exponentially, one would have to include quite a lot of past history into the state descriptor in order to make the process Markov. This remark explains why some of the approximations that follow are really unavoidable.

One could also decide to measure the QoS by the observed average response time, taking also the job lengths into account. The question of whether to use response time or waiting time as the QoS measure is largely one of marketing. From the point of view of the provider, waiting time is better because there is less uncertainty associated with it (lower variance), and the admission policy is simpler. On the other hand, users might prefer a SLA based on response times.

The service level agreements are similar to the ones defined in Section § 3.1, but are now stated in terms of streams and average waiting times:

Definition 6.2 (Charge)**.** *For each accepted and completed stream of type i a user shall pay a charge of c_i. In practice c_i depends on the stream's length k_i, the submission rate, γ_i, and the obligation, q_i.*

Definition 6.3 (Obligation)**.** *The observed average waiting time, W_i, of an accepted stream of type i shall not exceed q_i.*

Definition 6.4 (Penalty)**.** *For each accepted stream of type i whose waiting time exceeds q_i, the service provider shall pay to the user a penalty of r_i.*

As before, the service provider decides how many servers to allocate to each queue, while the admission control is slightly different: now it decides which streams are accepted and which ones are not. As usual, once allocated, a server is dedicated to serving jobs of the corresponding type only, until a subsequent re-allocation. The above system model is illustrated in Figure 6.1.

Both the resource allocation heuristics introduced in Section 3.3 can be used. However, since the Measured Queues heuristic offers no significant performance advantage, compared to the Measured Loads heuristic, and the estimation of arrival rates and average service times is in any case needed by the admission control algorithms that will be presented later in this chapter, from now on the queue sizes will be ignored and only the Measured Loads heuristic will be used.

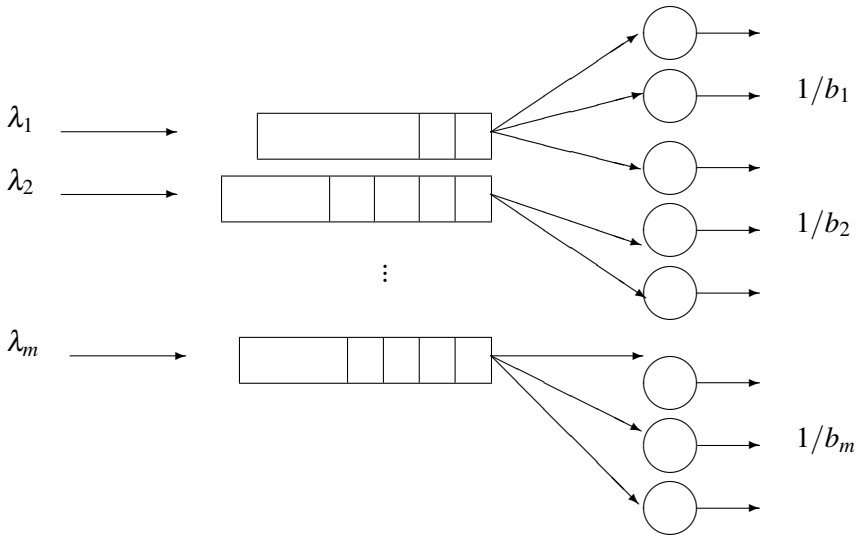


Figure 6.1: *System model for service streams.* $\lambda_i = L_i\gamma_i$*; sq. coeff. of var.* ca_i^2, cs_i^2.

The server allocation policy is invoked at stream arrival and stream completion instants. The admission policy is invoked at stream arrival instants. It must decide whether the incoming stream should be accepted or rejected. Of course, the allocation and admission policies are coupled: admission decisions depend on the allocated servers and vice versa. Moreover, they should be able to react to changes in user demand.

6.1.1 Assumptions

In order to make the model simpler, it is assumed that the time it takes to reallocate a server from one queue to another is negligible. That is certainly the case if all services are deployed on all servers, so that a reallocation does not involve a new deployment. The cost function may also include the cost of switching servers. However, if those switches were not instantaneous, then the analysis would become more complicated.

During the intervals between consecutive policy invocations, the numbers of active streams remain constant. Those intervals, which will be referred to as 'observation windows', are used by the controlling software in order to collect traffic statistics and obtain current estimates of the average values λ_i and b_i, and coefficients of variation ca_i^2 and cs_i^2. These values are used by the policies at each decision epoch.

Finally, it is assumed that the observation windows are reasonably large compared to the inter-arrival and service times, *i.e.*, that enough jobs arrive and are served during a window to provide good traffic estimates and to enable the system to be treated as having reached steady state.

6.2 Performance Evaluation

Equation (3.3), which is used to compute the average revenue received per unit time in the single jobs scenario, is now replaced by:

$$R = \sum_{i=1}^{m} a_i[c_i - r_i P(W_i > q_i)], \tag{6.4}$$

where:

- a_i is the average number of type i streams that are accepted into the system per unit time, and
- $P(W_i > q_i)$ is the probability that the observed average waiting time of a type i stream, (see equation (6.3)), exceeds the obligation q_i.

As described in Section 3.2, the objective of the resource allocation and job admission policies is to maximize the value of R. As before no assumptions about the values of charges and penalties are made, but the problem is interesting mostly if $c_i \leq r_i$. Otherwise one could guarantee a positive (but not optimal) revenue by accepting all streams, regardless of loads and obligations.

One could introduce penalties that are proportional to the amount by which the response time, or waiting time, exceeds the obligation q. The effect of that would be to replace the term $P(W > q)$ in the cost function with $E(min(0, W - q))$. A similar analysis would apply.

6.3 Admission Policies

Consider the subsystem associated with service i, with a given number of streams accepted and hence a given job arrival rate, λ_i. Suppose that n_i servers have been allocated to queue i. The offered load is $\rho_i = \lambda_i b_i$, and the stability condition is $\rho_i < n_i$. If the interarrival intervals and service times could be assumed to be distributed exponentially, then that subsystem could be modelled as an $M/M/n_i$ queue (see, for example, [78]). The average waiting time of a job would be given by:

$$w_{M/M/n} = \frac{b_i}{n_i - \rho_i} P(J \geq n_i), \tag{6.5}$$

where J is the number of type i jobs present, and so $P(J \geq n_i)$ is the probability that an incoming type i job will have to wait. That probability is given by the Erlang-C formula (or Erlang delay formula):

$$P(J \geq n_i) = \frac{n_i \rho_i^{n_i}}{n_i!(n_i - \rho_i)} p_0, \tag{6.6}$$

with p_0 being the probability of an empty type i subsystem:

$$p_0 = \left[\frac{n_i \rho_i^{n_i}}{n_i!(n_i - \rho_i)} + \sum_{j=0}^{n_i - 1} \frac{\rho_i^j}{j!} \right]^{-1} \tag{6.7}$$

If the Markovian assumptions are not satisfied, then the appropriate queueing model becomes $GI/G/n_i$, for which there is no exact solution. However, an acceptable approximation for the average waiting time, $\beta_i = w_{GI/G/n}$, is provided by the formula (see [119]):

$$\beta_i = \frac{ca_i^2 + cs_i^2}{2} w_{M/M/n}, \tag{6.8}$$

where ca_i^2 and cs_i^2 are the squared coefficients of variation of the interarrival intervals and service times of type i, respectively.

Moreover, when the system is heavily loaded, the waiting time in the $GI/G/n_i$ queue is approximately exponentially distributed [119]. Since the variance of the exponential distribution is equal

to the mean, the waiting time variance can also be approximated by β_i. Hence, the observed average waiting time of a stream, which according to equation (6.3) involves the sum of k_i waiting times, can be treated as being approximately normally distributed with mean β_i and variance β_i/k_i. That approximation appeals to the central limit theorem and ignores the dependencies between individual waiting times.

Based on the normal approximation, the probability that the observed average waiting time exceeds a given value, x, can be estimated as:

$$P(W_i > x) = 1 - \Phi\left(\frac{x - \beta_i}{\sqrt{\frac{\beta_i}{k_i}}}\right), \tag{6.9}$$

where β_i is given by equations (6.6), (6.7) and (6.8). The $\Phi(\cdot)$ function appearing in equation (6.9) is the cumulative distribution function of the standard normal distribution (mean 0 and variance 1). It can be computed very accurately by means of a rational approximation [6].

If $\rho_i \geq n_i$ (violating the stability condition), then it is natural to set $\beta_i = \infty$ and $P(W_i > x) = 1$ for any value of x.

The quality of the approximation expressed by equation (6.9) will depend on how well the implied assumptions are satisfied, namely *(i)* the load is heavy, and *(ii)* there is a large number of jobs per stream. If the former condition is not verified, that is, if the system is lightly loaded, there is no need of a clever admission policy; all incoming streams would be admitted. The latter, instead, ensures that any dependencies between the waiting times within a stream can be neglected.

For the following, it will be convenient to indicate explicitly the dependence of equation (6.9) on the parameters λ_i, k_i and n_i by introducing the notation:

$$P(W_i > x) = g_i(x; \lambda_i, k_i, n_i), \tag{6.10}$$

where $g_i(\cdot)$ stands for the right-hand side of (6.9). The other parameters, b_i, ca_i^2 and cs_i^2 are also involved, but do not need to be acknowledged explicitly.

6.3.1 Current State Heuristic

The state of subsystem i at any given point in time is specified by:

1. The number of streams, L_i, currently admitted;

2. For stream t, the number of jobs already completed, ℓ_t, and the average waiting time, u_t, achieved over those jobs; $t = 1, 2, \ldots, L_i$.

These values, as well as the parameter estimates, are available since traffic is monitored. Now let v_t be the average waiting time over the remaining $k_i - \ell_t$ jobs in stream t. A penalty r_i will be payable if the overall average waiting time for stream t exceeds the obligation q_i:

$$\frac{u_t \ell_t + v_t (k_i - \ell_t)}{k_i} > q_i, \tag{6.11}$$

or

$$v_t > \frac{q_i k_i - u_t \ell_t}{k_i - \ell_t} \tag{6.12}$$

Denote the right-hand side of expression (6.12) by $q_{i,t}$. Consider now an admission decision epoch for service i, that is, an instant when a new stream of type i is offered:

Case 1 If the new stream is rejected and no other action is taken, the expected total revenue from the current streams can be estimated as:

$$c_i L_i - r_i \sum_{t=1}^{L_i} g_i(q_{i,t}; \lambda_i, k_i - \ell_t, n_i) \tag{6.13}$$

Case 2 The alternative is to accept the new stream, possibly in conjunction with a reallocation of servers from other queues to queue i. As illustrated in Figure 6.2, such a decision would bring in an additional charge of c_i, but will also increase the job arrival rate at queue i by γ_i. This implies that there will be:

1. A possible penalty to pay for the new stream;
2. Different probabilities of paying penalties for the existing streams of type i, because of the new stream and eventually the added servers;
3. Different probabilities of failing to meet the promised QoS for the existing streams of the types that lost servers, if a reallocation occurred, because removing servers increased the effective load on those subsystems.

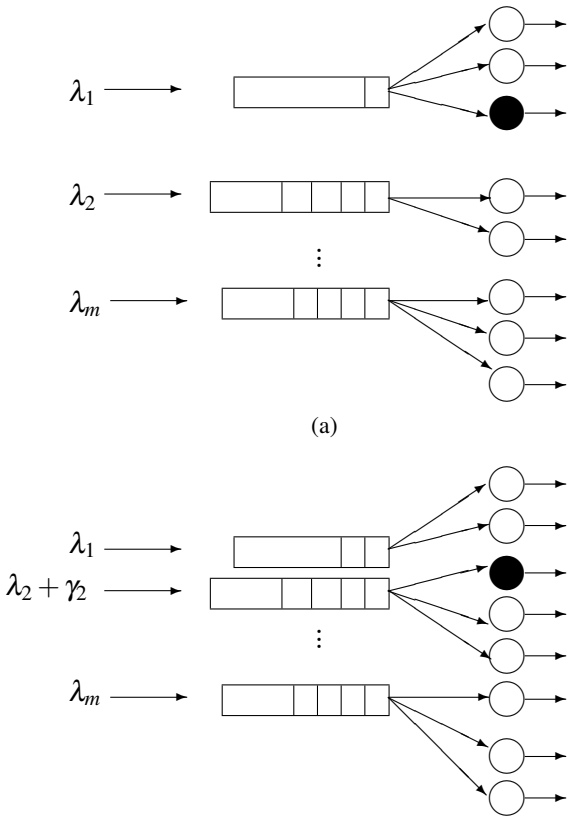


Figure 6.2: *(a) Original system configuration. (b) A new stream of type 2 has been accepted. As a result* (i) *the arrival rate for type 2 has increased to* $\lambda_2+\gamma_2$ *and* (ii) *one server has been removed from the first pool and is now dedicated to serve job of type 2.*

Denote by n'_j the new number of servers that queue j would have after a reallocation ($j = 1,2,\ldots,m$; $n'_1+n'_2+\ldots+n'_m = N$). The expected *change* in revenue resulting from a decision to reallocate and accept the new stream can be expressed as:

$$\Delta R = c_i - \overbrace{r_i g_i(q_i;\lambda_i+\gamma_i,k_i,n'_i)}^{\text{Expected penalty 1}} - \underbrace{\sum_{j=1}^{m} r_j \sum_{t=1}^{L_j} \Delta g_j(\cdot_t)}_{\text{Expected penalties 2, 3}}\ , \tag{6.14}$$

where $\Delta g_j(\cdot_t)$ is the change in the probability of paying a penalty for stream t at queue j. At queue i, that change is given by:

$$\Delta g_i(\cdot_t) = g_i(q_{i,t}; \lambda_i + \gamma_i, k_i - \ell_t, n_i') - g_i(q_{i,t}; \lambda_i, k_i - \ell_t, n_i),$$

while at other queues the change involves only the server reallocation; the arrival rates remain the same:

$$\Delta g_j(\cdot_t) = g_j(q_{j,t}; \lambda_j, k_j - \ell_t, n_j') - g_j(q_{j,t}; \lambda_j, k_j - \ell_t, n_j)$$

Equation (6.14) ignores the effect that the admission of a new stream might have on the coefficient of variation of the interarrival intervals. That effect is indeed negligible when the streams are close to Poisson, or when the number of currently active streams is large.

The above discussion suggests that, at stream arrival instants of type i ($i = 1, 2, \ldots, m$), the following policy may be adopted:

1. Invoke the Offered Loads allocation heuristic to determine the numbers of servers, n_j' ($j = 1, 2, \ldots, m$), that the queues would need if the new stream was accepted.

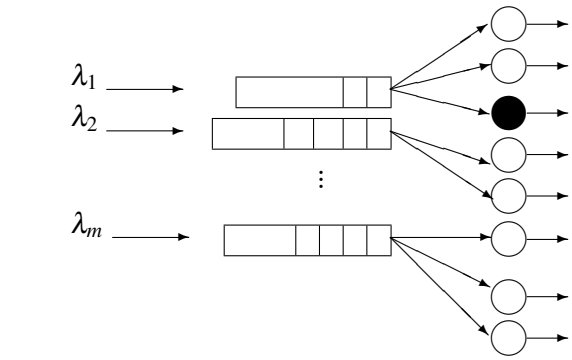
2. Evaluate the expected change in revenue in accordance with equation (6.14). If it is positive, carry out the server reallocation and accept the incoming stream. Otherwise, reject the new stream and leave the server allocation as it was.

This policy will be referred to as the 'Current State' admission heuristic.

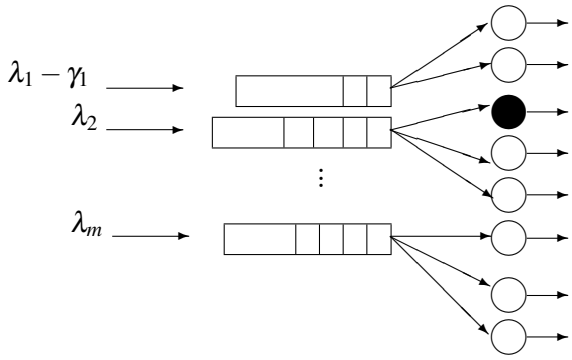
At instances of stream completion, the question of admission does not arise, but that of server reallocation does (since the offered load of one type is reduced). The Offered Load allocation heuristic is invoked and any switching of servers indicated by it is carried out, as shown in Figure 6.3.

The application of Offered Loads allocation and Current State admission implies that, if a stream arrives into an otherwise empty system, all servers are allocated to it. Then, if a stream of a different type arrives, several servers are switched to the other queue, etc.

One can easily relax the assumption that all streams of type i have the same job arrival rate, γ_i, and the same number of jobs, k_i. There is no problem in evaluating expressions (6.14) *and making allocation and admission decisions if those quantities vary from stream to stream, as long as each*



(a)



(b)

Figure 6.3: *(a) Original system configuration. (b) A stream of type 1 has completed. As a result the arrival rate for type 1 has decreased by γ_1 and, according to the Measured Load heuristics, one server has been removed from pool 1 and allocated to pool 2.*

incoming stream announces its arrival rate and number of jobs in advance. The charges, obligations and penalties could also vary from stream to stream.

Figure 6.4 shows how the use of the Current State heuristic can improve revenues, compared with not having any admission control policy in operation and accepting all submitted streams. When not stated otherwise, the following features are kept fixed between the experiments that will be presented in this chapter:

- The cluster consists of 20 servers and two types of services are offered, that is, $n = 20$ and $m = 2$;
- The obligations undertaken by the provider are that the average observed waiting time of the

jobs in a stream should not exceed their average required service time, *i.e.* $q_i = b_i$;

- All penalties are equal to the corresponding charges: $r_i = c_i$. In other words, if the average waiting time exceeds the obligation, the user that submitted the stream gets his or her money back.

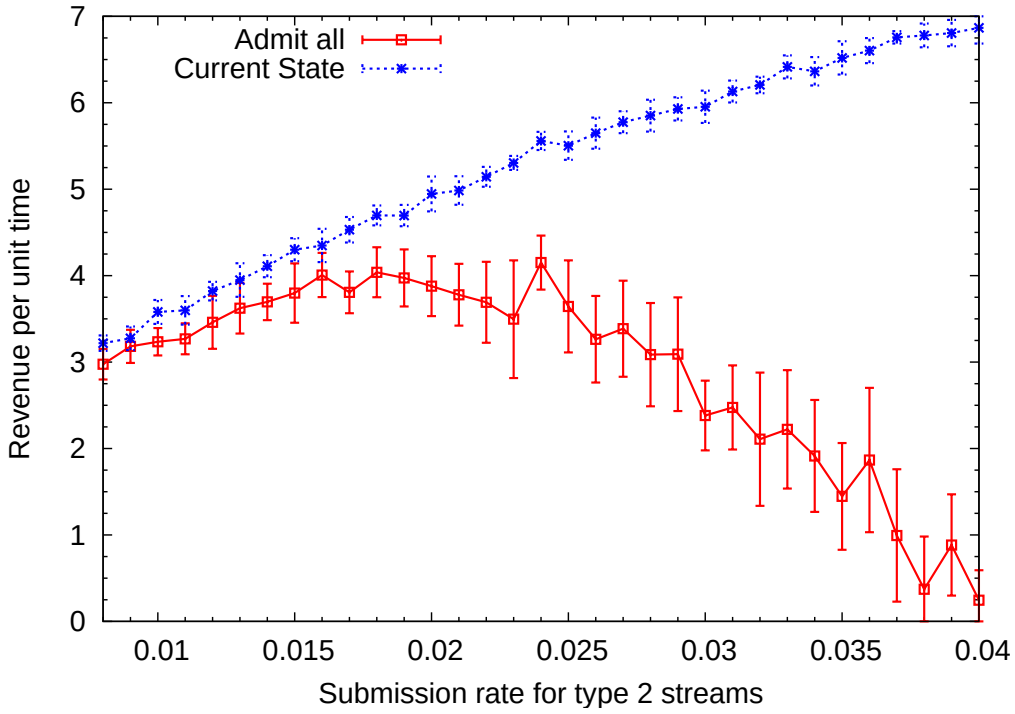


Figure 6.4: *Benefits of operating an admission policy.* $\delta_1 = 0.02, \gamma_1 = 0.2, \gamma_2 = 0.4, b_1 = 10, b_2 = 5, c_1 = 100, c_2 = 200$

The demand parameters of type 1 streams are: $\gamma_1 = 0.2$ (job arrival rate), $b_1 = 10$ (average service time), $k_1 = 50$ (number of jobs in a stream), $c_1 = 100$ (charge and penalty). Type 1 streams are submitted at rate $\delta_1 = 0.02$. The corresponding parameters for type 2 are $\gamma_2 = 0.4$, $b_2 = 5$, $k_2 = 50$ and $c_2 = 200$. However, the offered load of type 2 is increased in different runs, by reducing the average interval between stream submissions from 125 down to about 25 (that is, the submission rate δ_2 increases from 0.008 to 0.04). All interarrival, service and inter-stream intervals are distributed exponentially.

Each point corresponds to a simulation run of 110,000 time units, divided into 11 portions of 10,000 time units each for the purpose of computing a 95% confidence interval (the Student's t-distribution was used). As one might expect, at light loads there is not much difference between

having an admission policy and not having one, since nearly all streams are accepted anyway. However, when the system becomes more heavily loaded, the lack of an admission policy begins to have an increasingly significant effect. Whereas the revenues obtained by the Current State heuristic continue to increase throughout with a roughly constant slope, those of the 'Admit all' policy increase more slowly at first, and then quickly drop to near 0.

Figure 6.5 evaluates the effect of service time variability on performance. As well as the exponentially distributed service times ($cs_i^2 = 1$), the model was run with constant service times ($cs_i^2 = 0$) and with hyperexponential service times ($cs_i^2 > 1$). The average service times we kept the same as before, $b_1 = 10$ and $b_2 = 5$. The hyperexponential distribution had two phases: type 1 service times had a mean of 2 with probability 0.8 and a mean of 42 with probability 0.2; for type 2, the means were 1 with probability 0.8 and 21 with probability 0.2. The corresponding squared coefficients of variation are $cs_1^2 = cs_2^2 = 6.12$.

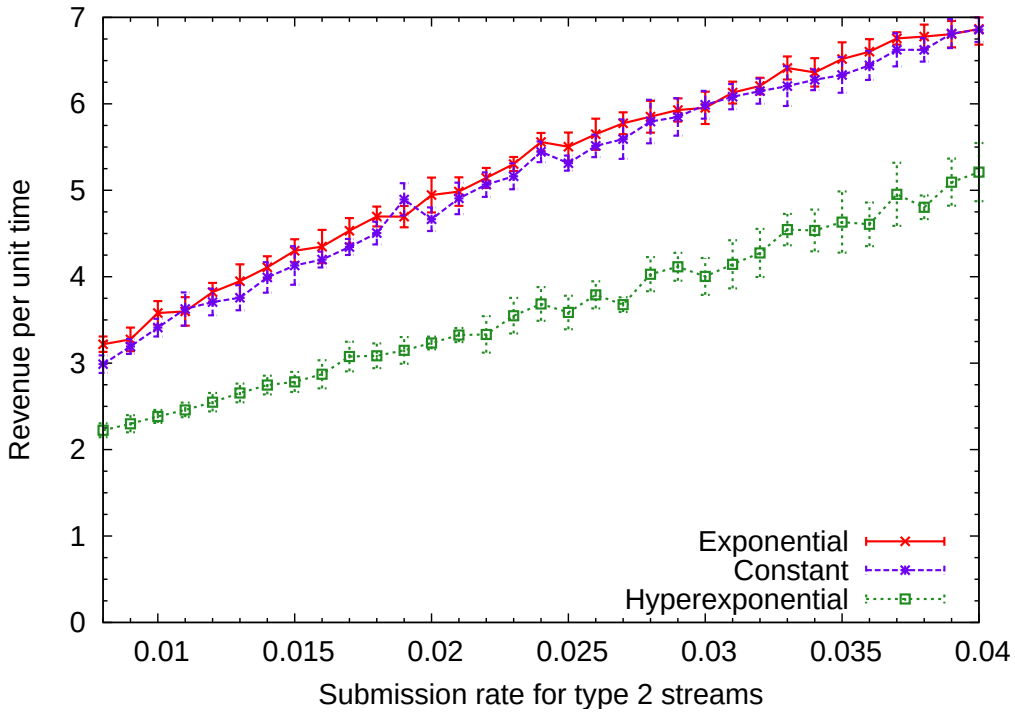


Figure 6.5: *Performance of the Current State heuristic with service times with different coefficients of variation. Hyperexponential: $cs_1^2 = cs_2^2 = 6.12$; Other parameters as in Fig. 6.4.*

One expects the performance to deteriorate when the variability of the demand increases, since the system becomes less predictable and it is more difficult to make correct admission decisions. In

fact, Fig. 6.5 shows that almost identical revenues are obtained when service times are distributed exponentially, as when they are constant. However, in the hyperexponential case, which was deliberately chosen to have very high variability, the revenues are indeed lower.

6.3.2 Policy Improvement

The decisions made by the coupled Offered Load allocation and Current State admission heuristics may well be sub-optimal. Consider, for example, an arrival instant of type i, when the Offered Load heuristic tries to accommodate the incoming stream. The application of expression (3.4) may err either in being too generous to queue i (thus increasing the likelihood of paying penalties in other queues), or not being generous enough (missing out on revenues from queue i). Therefore, it could be worthwhile trying to get closer to the 'optimal' server allocation at each decision instant, by carrying out one or more 'policy improvement' steps. At arrival instants, these have the following form:

1. Start with the allocation, $(n'_1, n'_2, \ldots, n'_m)$, that the Offered Loads heuristic would make if the new stream was accepted. Evaluate the corresponding expected change in revenue, ΔR, given by equation (6.14).

2. Try the $(m-1)$ switches where a server is moved from one of the other queues to queue i, and the $(m-1)$ switches where a server is moved from queue i to one of the other queues. In each case, evaluate expression (6.14) with the new vector $(n'_1, n'_2, \ldots, n'_m)$ and choose the best change; call that value $new\Delta R$.

3. If $new\Delta R > \Delta R$, set $\Delta R = new\Delta R$ and $(n'_1, n'_2, \ldots, n'_m)$ to the corresponding allocation, and repeat step 2; otherwise stop.

4. If ΔR is positive, carry out the server allocation $(n'_1, n'_2, \ldots, n'_m)$, and accept the incoming stream. Otherwise reject the new stream and leave the allocation as it was.

This procedure implements an optimization algorithm of the 'hill-climbing' variety (see Fig. 6.6), a search technique that works as follows. Starting from the current configuration $(n_1, n_2, \ldots, n_m)$, examine all *neighbor* configurations and move to the one with the highest expected revenue, ΔR. A neighbor configuration is defined as one in which one of the allocation vector values is changed by $+1$ or -1. So, for example, the neighbor configurations of (n_1, n_2, n_3) are $\{(n_1-1, n_2+1, n_3)$, (n_1-1, n_2, n_3+1), (n_1+1, n_2-1, n_3), (n_1+1, n_2, n_3-1), (n_1, n_2-1, n_3+1), $(n_1, n_2+1, n_3-1)\}$.

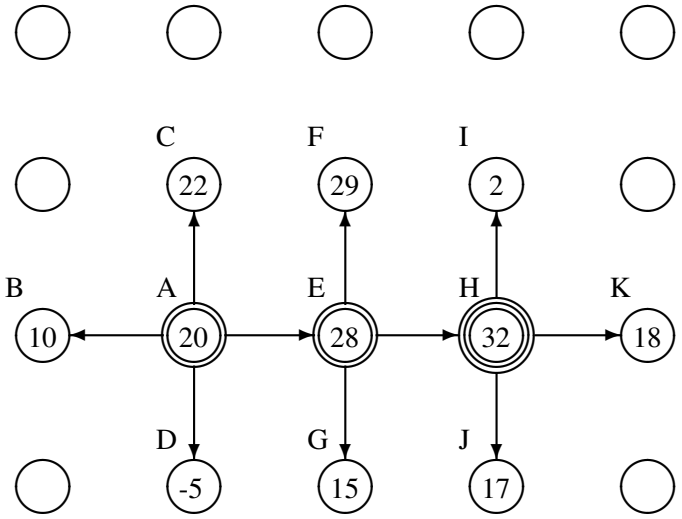


Figure 6.6: *Hill-climbing. The algorithm starts at 20 and stops at 32.*

The proposed algorithm economizes the computation by examining only $2(m-1)$ switches, *i.e.* $2(m-1)$ neighbouring allocations at each iteration, rather than all the $m(m-1)$ possible ones. The intuition is that a switch between a pair of queues neither of which was affected by a change in arrival rate is unlikely to be very advantageous.

A similar policy improvement algorithm can be applied at instants of stream completion of type i. Since the arrival rate in queue i has decreased, a sensible server reallocation, *e.g.* as indicated by the Offered Loads heuristic, would consist of removing a number of servers from queue i and assigning them to other queues. A question then arises whether that number is perhaps too large, or maybe not large enough.

Given the current server allocation, $(n_1, n_2, \ldots, n_m)$, and a proposed reallocation, $(n'_1, n'_2, \ldots, n'_m)$, the expected change in revenue can be estimated by an expression similar to (6.14):

$$\Delta R = -\sum_{j=1}^{m} r_j \sum_{t=1}^{L_j} \Delta g_j(\cdot_t), \tag{6.15}$$

where

$$\Delta g_j(\cdot_t) = g_j(q_{j,t}; \lambda_j, k_j - \ell_t, n'_j) - g_j(q_{j,t}; \lambda_j, k_j - \ell_t, n_j)$$

Note that the first two terms in the right-hand side of expression (6.14) are now absent, and there is no change in the arrival rates to be considered; the arrival rate in queue i has already been decremented appropriately.

The policy improvement steps carried out at the instants when a stream of type i is completed have the following form:

1. Set $\lambda_i = \lambda_i - \gamma_i$;
2. Apply the Offered Loads heuristic to produce an allocation vector $(n'_1, n'_2, \ldots, n'_m)$, then evaluate the corresponding expected change in revenue, ΔR, using expression (6.15);
3. Try the $(m-1)$ switches where a server is moved from one of the other queues to queue i, and the $(m-1)$ switches where a server is moved from queue i to one of the other queues. In each case, evaluate expression (6.15) with the new allocation vector $(n'_1, n'_2, \ldots, n'_m)$ and choose the best change, $new\Delta R$;
4. If $new\Delta R > \Delta R$, set $\Delta R = new\Delta R$ and $(n'_1, n'_2, \ldots, n'_m)$ to the corresponding allocation, and repeat the previous step; otherwise stop;
5. If ΔR is positive, carry out the server allocation $(n'_1, n'_2, \ldots, n'_m)$, that is, set $(n_1, n_2, \ldots, n_m)$ to $(n'_1, n'_2, \ldots, n'_m)$. Otherwise leave the allocation as it was.

The Current State admission heuristic does not make any assumption about any parameter, however it requires a rather detailed knowledge of the states of all active streams in the system. Its computational demands include evaluations of the right-hand side of equation (6.9) for every active stream in every queue. It may therefore be desirable to design simpler heuristics that allow admission decisions to be taken faster.

With that in mind, the following sections present two simple heuristics that can be employed if queues are subject to constant loads, or if the rates at which streams are submitted are known.

6.3.3 Long-Run Heuristic

The Current State heuristic does not require all streams belonging to the same job type i to have the same arrival rate, γ_i, and number of jobs, k_i. If those assumptions are satisfied, the following heuristic policy could be used.

Suppose that queue i is subjected to a constant load of L_i streams (*i.e.*, as soon as one stream completes, a new one replaces it) and has n_i servers allocated to it. Since each stream consists of k_i jobs submitted at rate γ_i, the average period during which a stream is active is roughly k_i/γ_i while, from Little's theorem, the rate at which streams are initiated is $L_i\gamma_i/k_i$. Under these conditions, the expected revenue earned by queue i per unit time would be:

$$R_i = \frac{L_i\gamma_i}{k_i}[c_i - r_i g_i(q_i; L_i\gamma_i, k_i, n_i)], \tag{6.16}$$

where $g_i(\cdot)$ stands for the right-hand side of expression (6.9).

The above observations imply that, if over a long period, the numbers of active streams in the system are given by the vector $L = (L_1, L_2, \ldots, L_m)$, and the server allocation is given by the vector $n = (n_1, n_2, \ldots, n_m)$, the total expected revenue earned per unit time can be computed using formula (6.4), where the average number of type i streams accepted per unit time, a_i, is replaced by $L_i\gamma_i/k_i$, while $L_i\gamma_i$ is used as arrival rate parameter in function $g_i(\cdot)$:

$$R(L,n) = \sum_{i=}^{m} \frac{L_i\gamma_i}{k_i}[c_i - r_i g_i(q_i; L_i\gamma_i, k_i, n_i)] \tag{6.17}$$

The idea is to use equation (6.17) not only to compute the expected average revenue earned per unit time, but as a basis for an admission heuristic too. Suppose that the current system configuration is given by the vectors L and n, and a new stream of type i is offered. If that stream is accepted and a server reallocation $n' = (n'_1, n'_2, \ldots, n'_m)$ is carried out, then the change in expected revenue may be estimated as:

$$\Delta R = R(L + e_i, n') - R(L,n), \tag{6.18}$$

where e_i is the ith m-dimensional unit vector, that is, a vector where the ith element is 1 and the other $(m-1)$ elements are 0, *i.e.* $L = (2,3,4)$, $L + e_2 = (2,4,4)$.

At stream arrival instants of type i, the new admission policy would operate as follows:

1. Invoke the Offered Loads allocation heuristic to determine the numbers of servers, n'_i ($i = 1, 2. \ldots, m$), that the queues would have if the new stream was accepted;

2. Evaluate the expected change in revenue, using equation (6.18). If it is positive, carry out the server reallocation and accept the incoming stream. Otherwise, reject the new stream and leave the server allocation as it was.

This policy will be referred to as the 'Long-Run' heuristic. The only state information it requires is the current numbers of active streams in each queue, together with the current server allocation. It is worth stressing that here the assumption that γ_i and k_i are the same for all streams of type i is important and cannot be easily relaxed. At stream completion instants, the server reallocation policy is the same as for the Current State heuristic discussed in Section 6.3.1.

The Long-Run heuristic can be optimized by applying one or more policy improvement steps, as described in Section 6.3.2. At stream arrival instants, the only change in the hill-climbing algorithm is that equation (6.18) is used in the place of expression (6.14). The steps applied at stream completion instants of type i are the same as those in Section 6.3.2, except that step 1 is replaced by:

1. Set $L_i = L_i - 1$,

while equation (6.15) is replaced by:

$$\Delta R = R(L, n') - R(L, n) \tag{6.19}$$

6.3.4 Threshold Heuristic

The Current State heuristic does not require or make use of the rates, δ_i, at which streams of different types are submitted. Yet one may expect that those parameters could play a role in admission decisions. If that is the case, the following heuristic can be used.

If streams of type i are submitted at rate δ_i and each such stream consists of k_i jobs of average length b_i each, then the 'potential' offered load of type i (*i.e.*, if all streams are accepted), is $\phi_i = \delta_i k_i b_i$. Suppose that servers are allocated to service types in proportion to these loads, according to equation (3.4) but replacing ρ_i with ϕ_i. Having fixed those allocations, the different services can be decoupled and considered in isolation of each other.

The idea behind the proposed admission heuristic is similar to the one presented in Section § 3.4, meaning that is based on a vector of thresholds, $(M_1, M_2, \ldots, M_m)$. If, at the moment when a stream of type i is submitted, there are fewer than M_i active type i streams, the new stream is accepted, otherwise it is rejected. The problem is how to choose those thresholds.

Assuming that the stream submission processes are Poisson, and bearing in mind that the average 'duration' of a type i stream is k_i/γ_i, the number of active streams of type i, for a given threshold M_i, can be modeled as the number of calls in an Erlang system with M_i trunks and traffic intensity

$\sigma_i = \delta_i k_i / \gamma_i$. The steady state probability, $p_{i,j}$, that there are j active streams of type i, is given by (see [78]):

$$p_{i,j} = \frac{\sigma_i^j}{j!} p_{i,0} \ ; \ j = 0, 1, \ldots, M_i, \tag{6.20}$$

where

$$p_{i,0} = \left[\sum_{s=0}^{M_i} \frac{\sigma_i^s}{s!} \right]^{-1} \tag{6.21}$$

Since the Erlang model is insensitive to the distribution of call times, there is no need to worry about the distributions of stream durations.

Now, the average revenue that is obtained per unit time from type i services can be estimated as:

$$R_i = \sum_{j=0}^{M_i} p_{i,j} \delta_i [c_i - r_i g_i(q_i; j\gamma_i, k_i, n_i)], \tag{6.22}$$

where $g_i(\cdot)$ is the probability of paying a penalty for a type i stream when there are j such streams active (the job arrival rate is $j\gamma_i$) and n_i servers have been allocated; that probability is given by expression (6.9).

The threshold M_i is chosen so as to maximize the right-hand side of equation (6.22). According to some numerical experiments carried out, R_i is computed for different threshold values, it is a unimodal function of M_i. That is, it has a single maximum, which may be at $M_i = \infty$ for lightly loaded systems. That observation implies that one can search for the optimal admission threshold by evaluating R_i for consecutive values of M_i, stopping either when R_i starts decreasing or, if that does not happen, when the increase becomes smaller than some ε. Such searches are typically very fast.

This admission policy will be referred to as the 'Threshold' heuristic. It is a very economical policy to implement, in terms of computational overheads. This is because it is essentially a static policy: the server allocations and the corresponding admission thresholds are computed only once, for a given set of demand parameters. The system is still monitored, and if one or more of those parameters are observed to have changed, the allocations and thresholds are recomputed. A possible implementation is to use observation windows of size J, such that a total of J streams of all m types arrive during a window, as described in Chapter 3.

Figure 6.7 compares the performance of the Current State, Threshold and Long-Run admission

heuristics. The setting is the same as in Figure 6.4, with the same demand parameters and lengths of simulation runs.

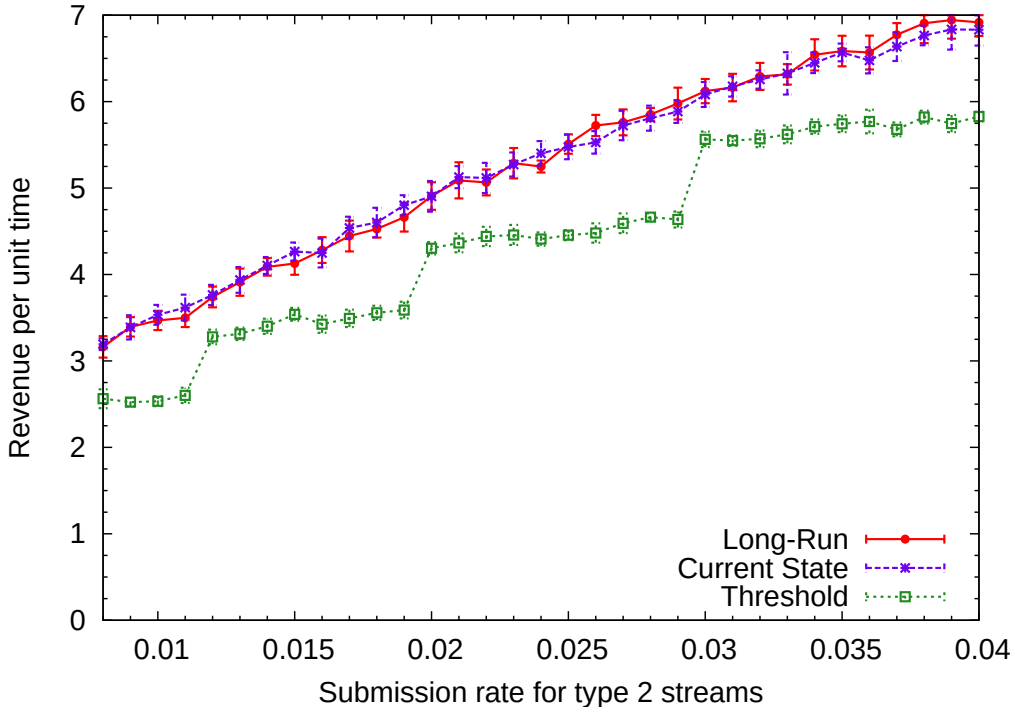


Figure 6.7: *Comparison of different heuristics. Demand parameters as in Fig. 6.4.*

The Long-Run heuristic performs slightly better than the Current State, with the points within each other confidence intervals, while the simple Threshold heuristic performs quite close to the more complicated policies. The revenues obtained are fairly close and the confidence intervals are of similar size. This suggests that the Threshold and Long-Run heuristics might be a suitable choice for practical applications. Finally, the line showing the revenues obtained using the Threshold policy has a characteristic step shape. This is because for adjacent points the Measured Load heuristic policy allocates the same amount of servers to each queue and, as a result, the chosen thresholds are similar, if not the same.

The experiments carried out so far have shown the feasibility of the proposed approach. The aim of the next set of simulations is to assess the scalability of the proposed policies. The system under study is composed of 100 servers (*i.e.* $n = 100$), while the data center provides 8 job types

(*i.e.* $m = 8$) whose parameters are summarized in Table 6.1. As effect of increasing the rate at which streams of type 8 are submitted, δ_8, the total load ρ varies between 32.8% and 94%.

Index	b_i	γ_i	k_i	δ_i	c_i	ρ_i
1	1.0	2.0	50	0.100	5.0	2.5
2	2.0	1.0	100	0.100	10.0	20.0
3	5.0	0.5	60	0.025	8.0	15.0
4	1.0	5.0	50	0.200	6.0	2.0
5	1.0	0.1	10	0.100	2.0	10.0
6	1.0	2.0	100	0.100	11.0	5.0
7	1.5	2.0	50	0.200	15.0	7.5
8	1.0	1.0	80	0.010 ... 0.400	11.0	0.8 ... 32.0

Table 6.1: *Experiment settings.* $m = 8, n = 100, c_i = r_i$.

Figure 6.8 shows the total revenue earned by the system per unit time by all the presented policies against the submission rate δ_8. As in Figure 6.4, when the system is lightly loaded there is almost no difference between operating an admission policy or admitting all streams. However, as the load increases, the revenues obtained using the Admit All policy increase more slowly than the others, and as soon as the load gets to 70% ($\delta_8 = 0.15$) the lack of an admission control policy shows its negative effects, with the revenues dropping to 0 as the system approaches the saturation point. Surprisingly, both the Current State and the Long-Run heuristics perform as well as their optimized versions (the statistical differences are negligible). This is because when m and n are big all the approximations carried out by those policies are more accurate, and so there is little or no space for further optimizations. Finally, the Threshold policy performs a little worse, even though the revenues obtained by such a policy are still acceptable. On the other hand, as mentioned earlier in this chapter, it is important to emphasize that the Threshold policy is very fast and easy to implement.

The achievable revenues are influenced by two factors, namely the success rate and the reject rate. A policy can under perform (i) because it rejects too many streams, and so it misses potential revenues, or (ii) because it does not reject enough, and so it ends up failing to meet the promised QoS levels and thus paying penalties.

Figures 6.9 and 6.10 analyze the success rate and reject rate, respectively. According to Figure 6.9 the Threshold policy never fails, however if does not perform as good as the other algorithms

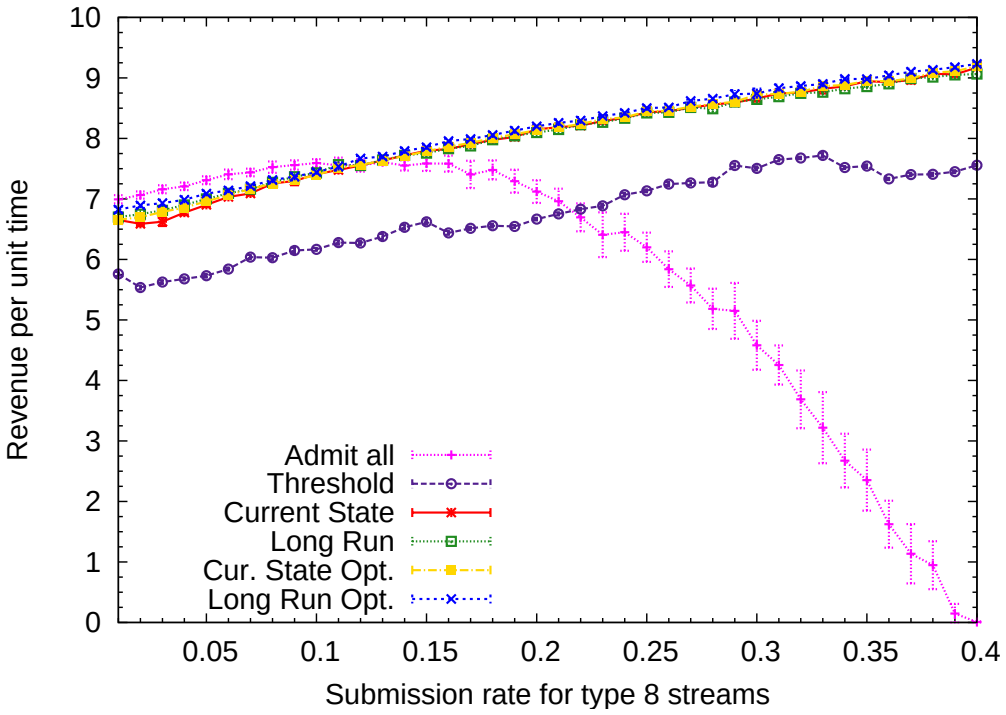


Figure 6.8: *Performance comparison of different policies. Parameters as in Table 6.1.*

because it rejects too many streams (it does not distinguish a stream that has just begun from a stream that is almost at the end). When the system is lightly loaded the Admit All policy achieves slightly better revenues than all the other algorithms, in spite of some failures, because it does not miss any revenue. The other policies perform similarly even though they behave differently, that is, some reject a bit more streams but almost never fail, while others accept more streams and fail to deliver what promised a bit more often.

Threshold Policy and Window Size So far the Threshold policy has been used with a static threshold. The parameters were estimated at the beginning and the algorithms chose how to partition the available servers and the threshold values for each queue. This is of course not a realistic scenario, since one of the requirements is that the policy should be able to react to changes in user demand. Figure 6.11 shows the revenues obtained by the Threshold policy using different values for the parameter J (the other settings are the one described in Table 6.1): longer windows perform better than shorter ones because they produce better parameters estimates, leading to more accurate approximations. Section 8.4 experiments with windows of different sizes in a more realistic scenario where the load changes periodically.

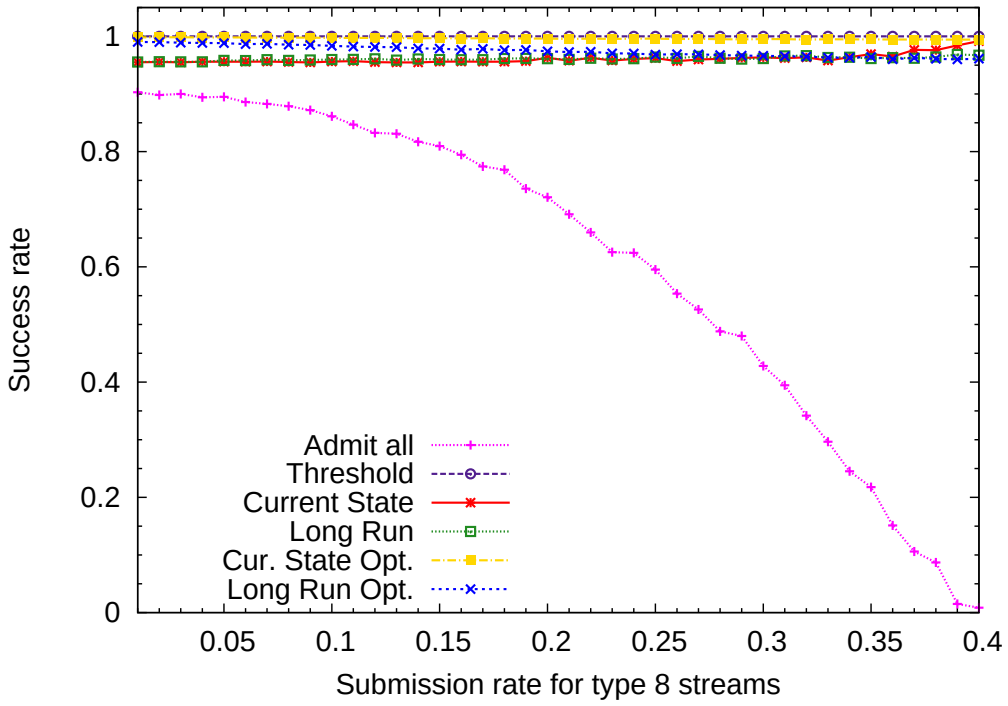


Figure 6.9: *Success rate comparison of different policies. Parameters as in Table 6.1.*

Tightening the SLAs The presented policies don't make any assumption about the relative magnitude of the parameters. The last set of simulations were carried out in order to try to understand how the algorithms' behavior changes as the SLAs become more strict . So far the target average waiting times, q_i, were set to be the same as the average service times, b_i. Now $q_i = b_i/2$, while all the other parameters are the same as before.

Figure 6.12 shows that the obtained revenues are now slightly lower than before, while the revenues obtained by the Admit All policy drop earlier and faster than before. The success rates of the various policies (Figure 6.13) are similar to the previous scenario because now the reject rates are higher, that is, the algorithms are more conservative (Figure 6.14).

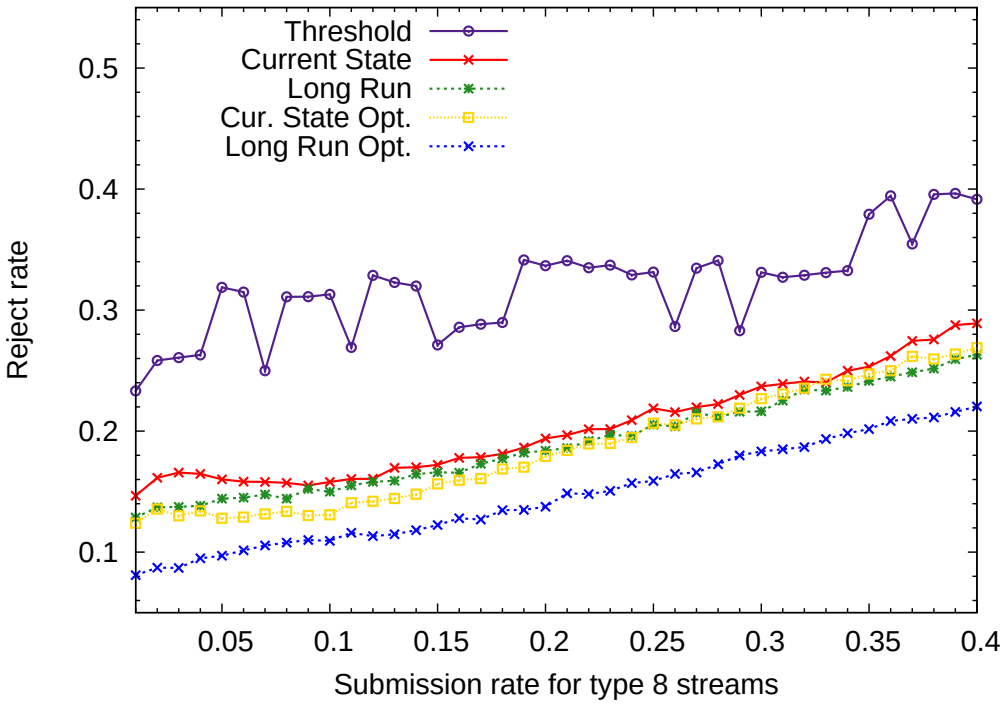


Figure 6.10: *Reject rate comparison of different policies. Parameters as in Table 6.1.*

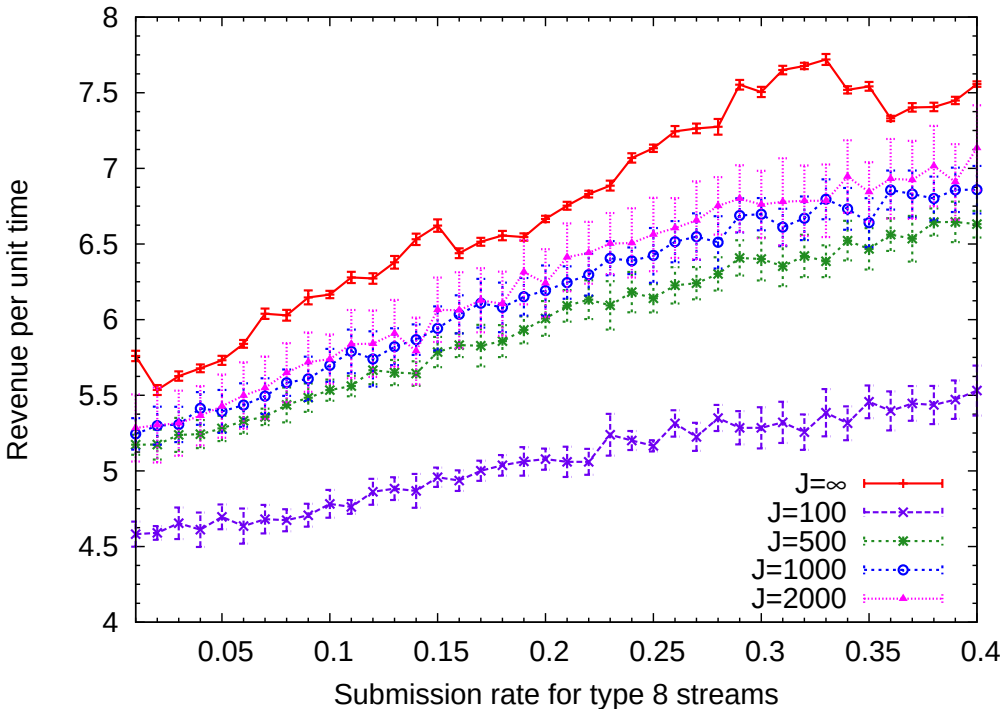


Figure 6.11: *Performance of the Threshold heuristic with different J values. Parameters as in Table 6.1.*

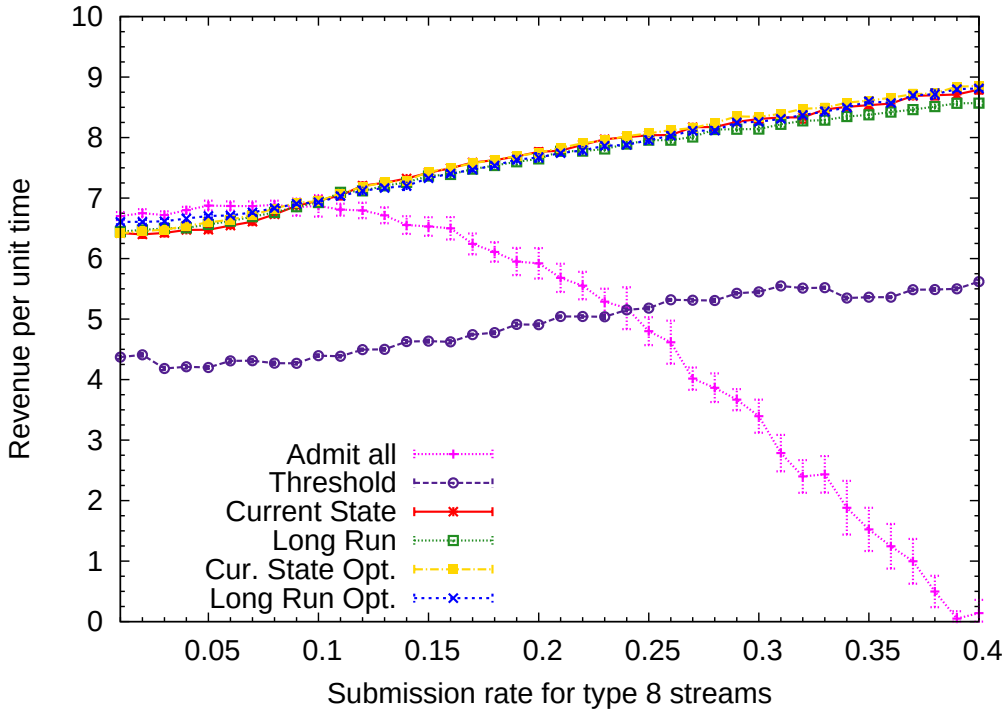


Figure 6.12: *Performance comparison of different policies.* $q_i = b_i/2$, *other parameters as in Table 6.1.*

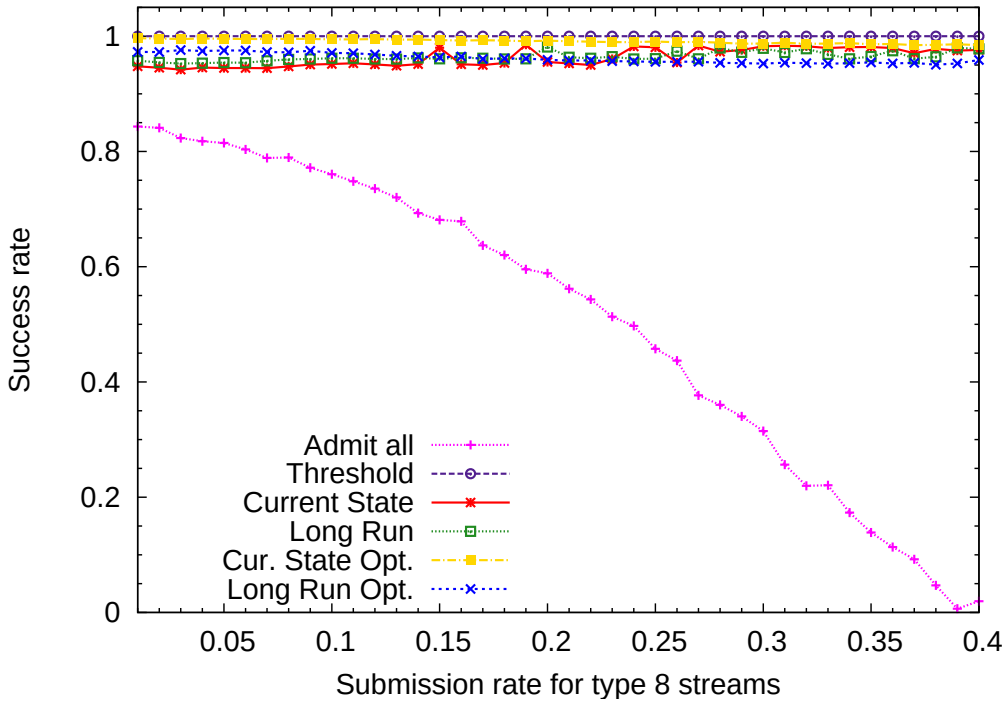


Figure 6.13: *Success rate comparison of different policies.*

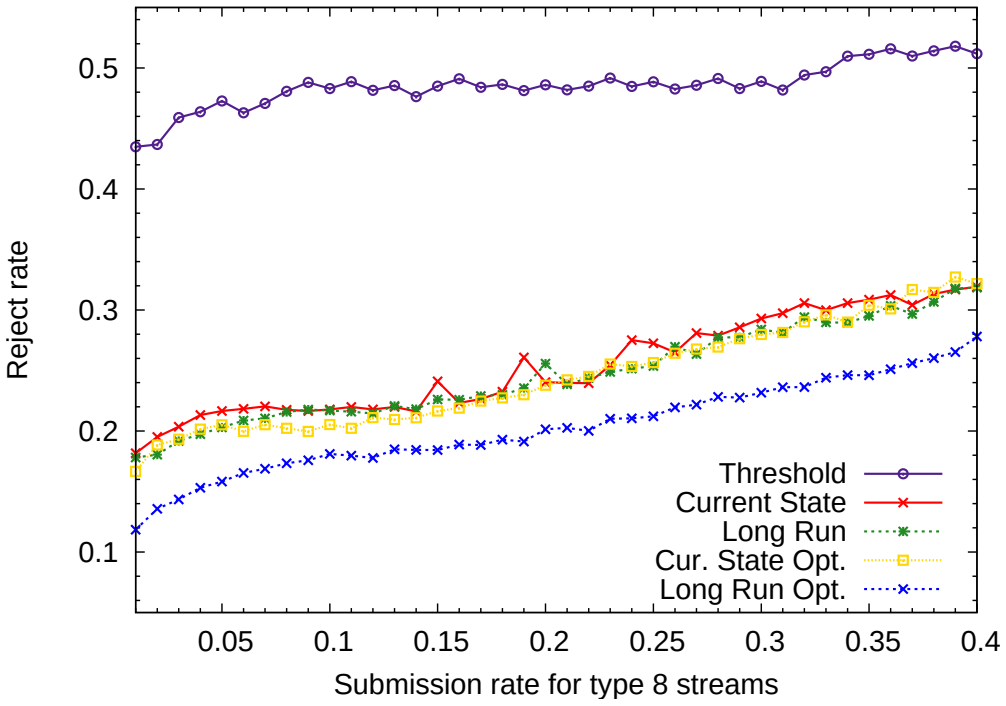


Figure 6.14: *Reject rate comparison of different policies.*

Chapter 7

API Extensions for Service Streams

This chapter presents the extensions made to SPIRE in order to deal with streams of requests. The idea remains the same, *i.e.*, there is still a (eventually replicated) front-end controlling the flow of messages; what follows discusses only the changes needed to implement the session based admission control algorithms that were introduced in Chapter 6.

7.1 Packet Double-Rewriting

The API is very similar to the one discussed in Chapter 4, with the exception that now the system must keep track of the active streams in order to take sensible admission control decisions. To do so, the APIs depicted in Figure 4.8 have been extended as shown in (Figure 7.1): the `StreamRoutingService` exposes a new operation, `addStream()`, request and response objects now store a cookie used to uniquely identify the stream they belong to, while the `Stream` represent a set of requests belonging to the same session, as defined in Section 6.1. Finally, the `RouterDispatcher` interceptor, whose `invoke()` method is shown in Figure 4.6, is now also able to redirect 'add stream' requests to the load balancer. It discriminates such type of requests by the local name of the payload (*addStream*, see Figure A.5).

StreamRoutingService As mentioned above, the mediation service now provides an operation for the purpose of adding new streams. The four methods discussed in Section 4.2.1 provide the same functionalities as the `RoutingService`, with the only subtlety that `forward()` and `result()` operations now create 'stream' requests and responses to carry a cookie, that is attached to every message exchanged by the parties:

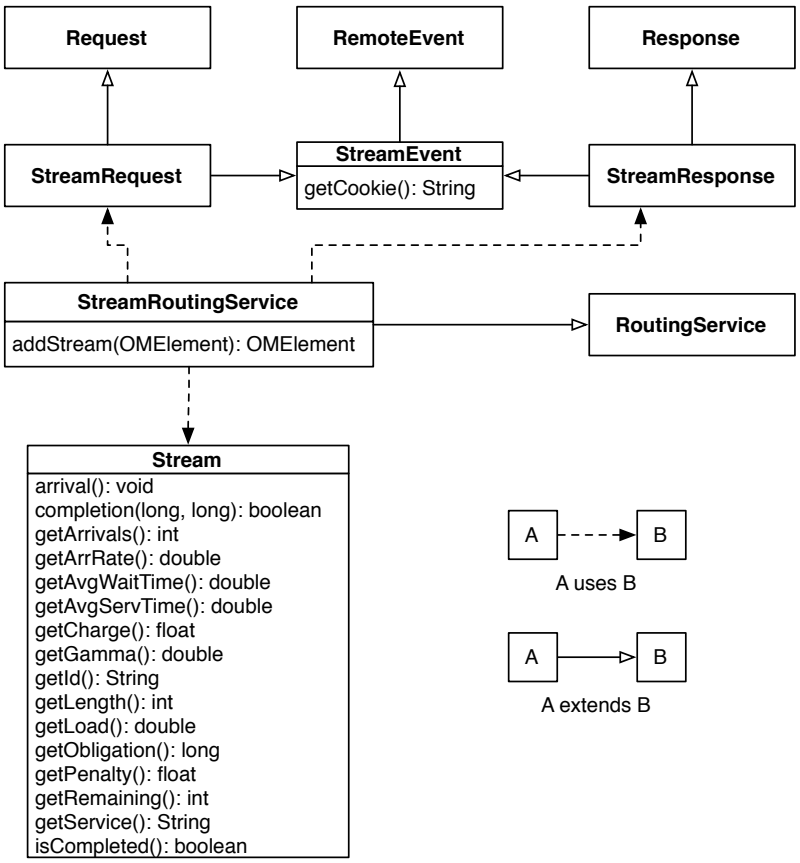


Figure 7.1: *APIs for streams. The* `RoutingService` *and the remote events objects are defined in Figure 4.8.*

- `addStream()` extracts the parameters from the payload (arrival rate, target service and number of jobs composing the stream) and creates a `Stream` instance. If the current-state or the long-run heuristic policies are in operation, it is possible to use values other than the default service time and demand parameters. Then, the `Stream` is sent to the `StreamDispatcher`, where the admission policy decides whether to admit it or not. If the new stream is accepted, SPIRE sends the cookie to the client. This step is needed because requests are accounted to the stream they belong to, while statistics information are collected on a stream basis. As a side note, both a synchronous and an asynchronous version of this operation have been implemented, but since their performance is very similar and the asynchronous implementation is by far more complicated (because the load balancer must store all the addressing information), only the synchronous version is used.

Stream The `Stream` interface represents an active stream. It provides methods to query the state of the stream as well as operations to modify its internal state.

Methods of the first category include `getArrRate()`, to get the monitored arrival rate of jobs belonging to the stream under consideration, as opposite to `getGamma()`, that returns the arrival rate specified by the *addStream* request message. Also in this class are the methods used to retrieve the economic parameters and `getId()`, that returns the cookie identifying the stream. The `getAvgWaitTime()` method returns the average waiting time for the completed requests, while `getAvgServTime()` gets the average service time experienced by completed jobs. Monitoring the arrival rate and the service time values allows SPIRE to easily compute the load offered by the current stream by using the traffic equation $\rho = \gamma b$ [78], whose value is returned by the `getLoad()` method. Finally, `getService()` can be used to get the service this stream belongs to (*i.e.*, where requests of this stream will be queued), while `getRemaining()`, `getArrivals()`, `getLength()` and `isCompleted()` can be used to check how many of the k jobs composing the stream have arrived and completed.

The second class of operations include only two methods, `arrival()` and `completion()`. The former simply increases the number of arrivals (which, in turns, updates the arrival rate for the stream under analysis). The latter, that requires the waiting and service times of the completed job as arguments, updates the average values, increments the number of completions, and returns *true* if the stream is completed, *i.e.*, if all the jobs have been run, *false* otherwise.

7.2 Requests Dispatching

As in the previous section, the dispatching API has also been modified in order to deal with streams. It is worth noting that due to the big differences between the various admission policies, two APIs are needed, one to implement the current-state heuristic and its extensions (improved and long-run policies), the other to implement the algorithm using the vector of thresholds.

7.2.1 Current State Heuristic Implementation

As before, the following focuses only on the changes needed to deal with streams. All the operations or steps that are the same as in the single job scenario, such as adding a new node or service, will not be covered. The structure of this section is the same as the one of Section 4.2.2. As shown in Figure 7.2, the main changes and additions in comparison with the single job scenario are:

- `StreamQos` extends the `Qos` interface. For queue i, it stores also the expected average service time, b_i, as defined by expression 6.2. If the average service time for a certain stream differs from the default value, it is included into the *addStream* message.

- `StreamDispatcher` extends the `Dispatcher` interface. It provides the `addStream()` method to evaluate the admission policy, while the server configuration is changed by using the `executeAllocationTask()` operation. This is invoked at stream completion events by the handler where the completion occurred. As in Chapter 4, such tasks are executed using a single-thread, dedicated thread pool in order to guarantee that only one of them is executed at any given time (*i.e.*, the new allocation vector is updated 'atomically'[1]).

- The `StreamEvaluatorTask` implements the admission policy; like allocation tasks, evaluator tasks run on a dedicated pipeline too. In order to prevent `addStream` messages from starving, the working queue is bounded to a small value (*e.g.*, 20) and, in case of 'overflow', the stream is automatically rejected. Going back to the semantics of the `addStream()` operation exposed by the load balancer (*i.e.*, synchronous versus asynchronous), this guarantees that admission decisions are taken promptly, and thus the decision to use a synchronous semantics proves to be correct because clients do not wait for a long time.

- Each `Allocation` instance stores a pair (n_i, n_i'), where the first value is the current number of servers that are allocated to queue i, while n_i' (the 'virtual' value) corresponds to the number of servers that should be allocated to queue i if the stream (not necessarily of type i) is accepted.

- The `State` interface represents the state of each queue. It holds the 4-tuple defined by expression 6.2 as well as methods to add new streams and query and modify their states.

The `addNode()` and `addService()` operations are exactly the same as in the single job scenario. The remainder of this chapter describes the information flow and the interaction between the API components when handling `addStream()`, `queue()` and `response()` events and analyzes their computational cost.

[1] In this context atomically does not mean that the operation is executed in a transactional manner, but simply that other threads do not interfere.

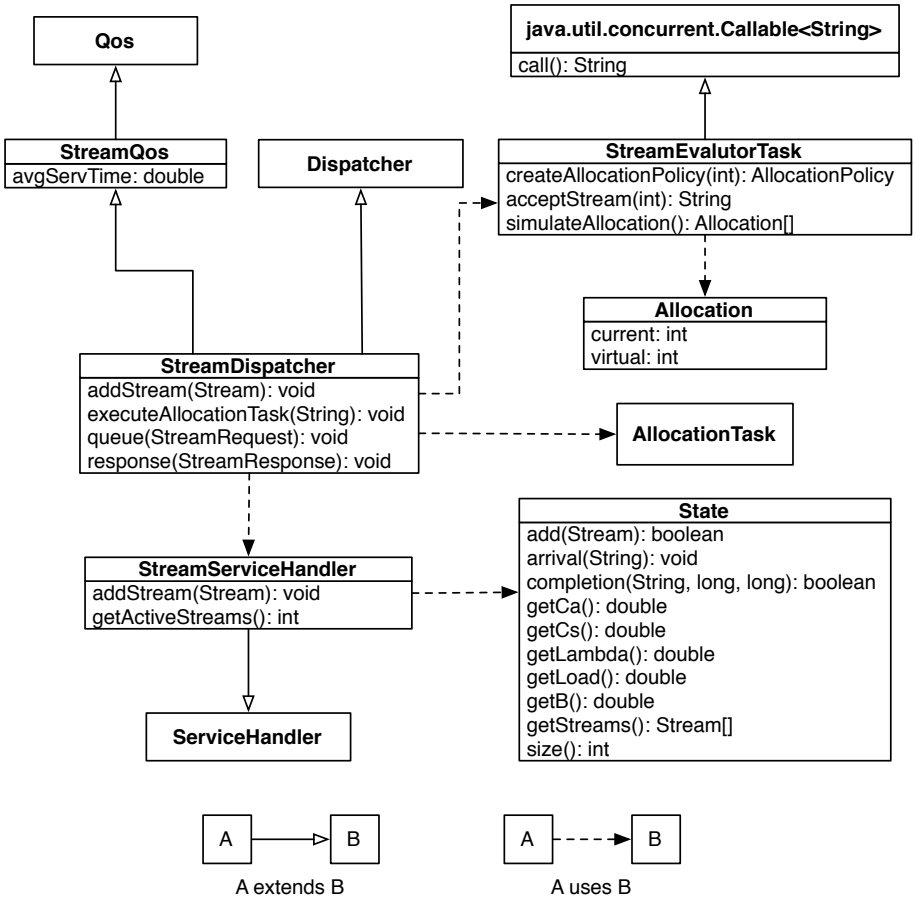


Figure 7.2: *Dispatching API for streams (current state, improved and long-run policies). The* `Dispatcher`*,* `ServiceHandler` *and* `AllocationTask` *and* `AllocationPolicy` *interfaces and the* `Qos` *class are the ones defined in Figure 4.11.*

Add stream This method requires a `Stream` object as argument. The dynamic of this operation is very simple: as depicted by the UML activity diagram shown in Figure 7.3, the `Dispatcher` creates and starts a `StreamEvaluatorTask` and waits for its decision. If the stream is accepted, the dispatcher returns the cookie to the client (the cookie is contained into the `Stream` object), otherwise it returns an error message. The way the admission decision is taken is obviously policy dependent, but the steps involved in reaching such decision are the same, no matter what algorithm is employed. The evaluator task first creates a new 'virtual' allocation vector, $(n'_1, \cdots, n'_m)$, by invoking the `simulateAllocation()` operation (the allocation policy to use is returned by the `createAllocationPolicy()` method with parameter the number of available nodes, N). Then it runs the admission algorithm and, if the taken decision is to accept a stream of type j, it

invokes the `acceptStream()` method. This (i) replaces all the current n_i values with the virtual vector values and (ii) invokes the `addStream()` method on handler j with the stream as argument. This, in turn, adds the new stream to the `State` by invoking the `add()` method. As in the case of service handlers, active streams can also be stored in several ways. For reasons similar to the one discussed in Section 4.2.2, each `State` instance saves the active streams into a hash table, where the key is a cookie and the value is a stream. Under these circumstances, adding a new stream requires constant time, plus the cost associated to the evaluation the admission policy.

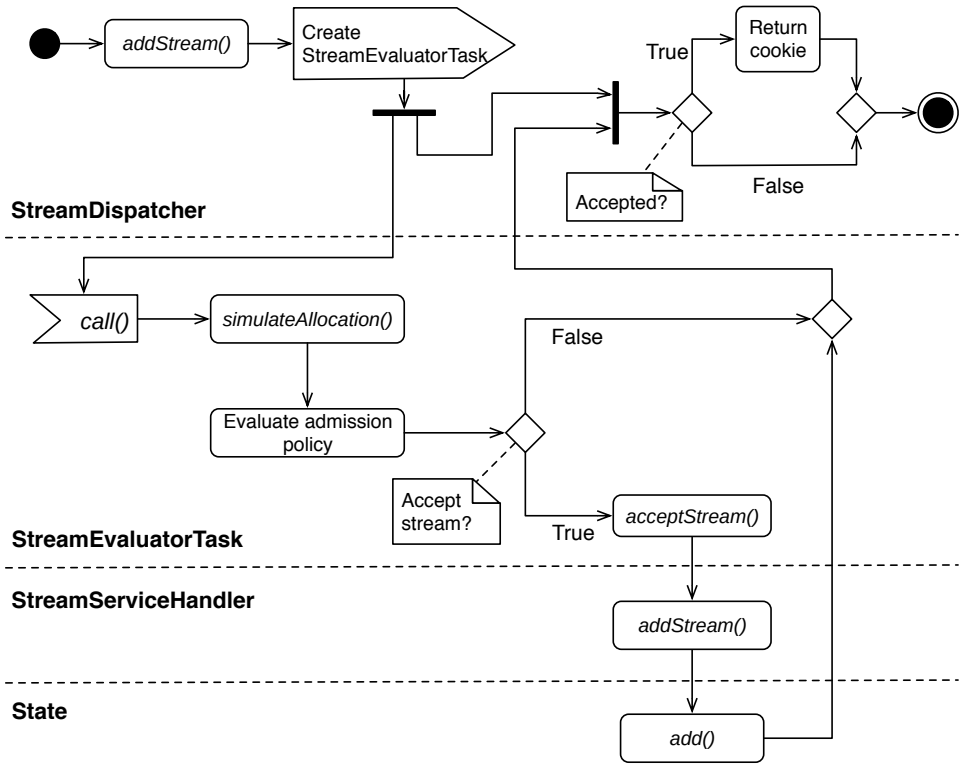


Figure 7.3: *Activity diagram for the* `addStream` *operation (current state, improved and long-run policies). The* `StreamEvaluatorTask.acceptStream()` *method puts the new allocation vector in operation. The* `StreamServiceHandler.addStream()` *method adds the stream to the set of 'active' streams.*

Queue Handling the arrival of a job belonging to a stream is simpler than the arrival of a single job because the admission decision has already been taken. The main consequence is that the employed algorithm does not depend on the admission policy in operation. Instead, the tasks to accomplish are

always the same: as Figure 7.4 shows, it is just a matter to find a spare node and to update the state of the stream the job belongs to. As for the single job scenario, the handler still provides two ways to handle queue events, `queue()` and `submit()`. When a request of type i belonging to stream j is received, the dispatcher decides which of the two operations to use (the former is invoked on queue i if there are no machines available in the idle nor the spare pools, or if queue i has at least one server in idle state; the latter if one of the spare servers is going to be used to execute the job). Handler i invokes `State.arrival()`, that in turns calls `Stream.arrival()` on the $j-th$ stream to increase the number of arrived jobs for that stream. After this stage, if the `submit()` method was used, the oldest job is executed on the specified node, while if the `queue()` operation was invoked, the specified request is forwarded to a (previously) idle machine. Finally, the cost of this operation is $O(1)$ if the jobs waiting are stored in a linked list or similar data structure.

Response Jobs are executed the same way as before. The only responsibility for the target machine is to save the cookie before running the job and to 'attach' it to the header of the response once the job has completed. When a response of type i belonging to stream j arrives (Figure 7.5), the dispatcher simply 'forwards' it to the $i-th$ `StreamServiceHandler`. The handler gets the cookie identifying stream j as well as the service and waiting time values from the response object and passes this 3-tuple to the state, via its `completion()` method. This increments the number of completed jobs for type i, decreases the number of pending jobs of type i, and updates the global average values for type i jobs, then invokes `completion()` on stream j with the service and waiting times as arguments.

The state of the stream is updated by increasing the number of completed jobs and by adding the new values to the averages. Finally, a boolean value indicating whether the stream has completed is returned. If the stream is completed, the `State` computes whether a penalty should be paid or not (*i.e.*, if the SLA was met or not), then it removes stream j from the hash table storing the active sessions. Finally, it returns the completion status to the handler. If the job completion event corresponds to the completion of stream j, handler i executes the allocation task by calling the `executeAllocationTask()` with parameter i (the argument is indeed used only by the improved allocation algorithm as queue i is used as a starting point by the hill-climbing algorithm described in Section 6.3.2), then it checks whether the server should be released or not by comparing n_i and the number of currently allocated servers. If that is the case, the server is returned to the dispatcher via the `releaseNode()` method (that will provide to allocate it elsewhere), otherwise

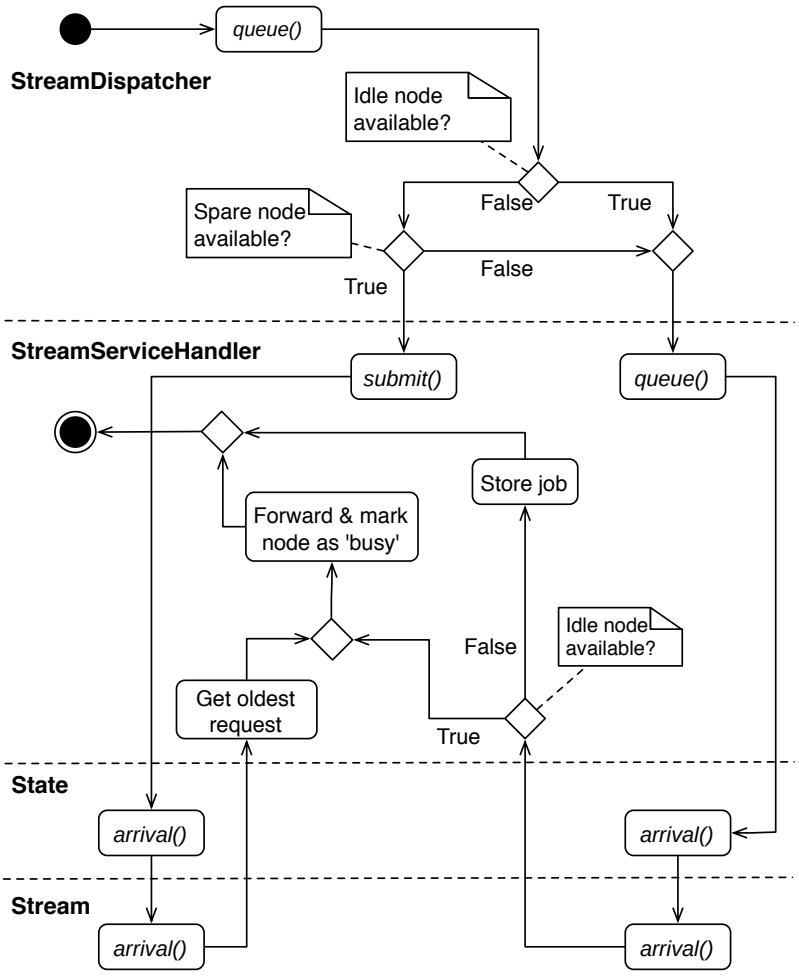


Figure 7.4: *Activity diagram for the* `queue` *operation. No admission control evaluation is involved at this stage. Each* `StreamRequest` *instance contains the cookie identifying the stream it belongs to as well as the target service.*

the handler forwards the next request, if any, as in the single job scenario.

The cost of response events depends whether a stream completion occurs as well and if the node is released or not: the time required by this operation is $O(1)$ if the stream is not completed and the node can be reused, $O(n)$ otherwise.

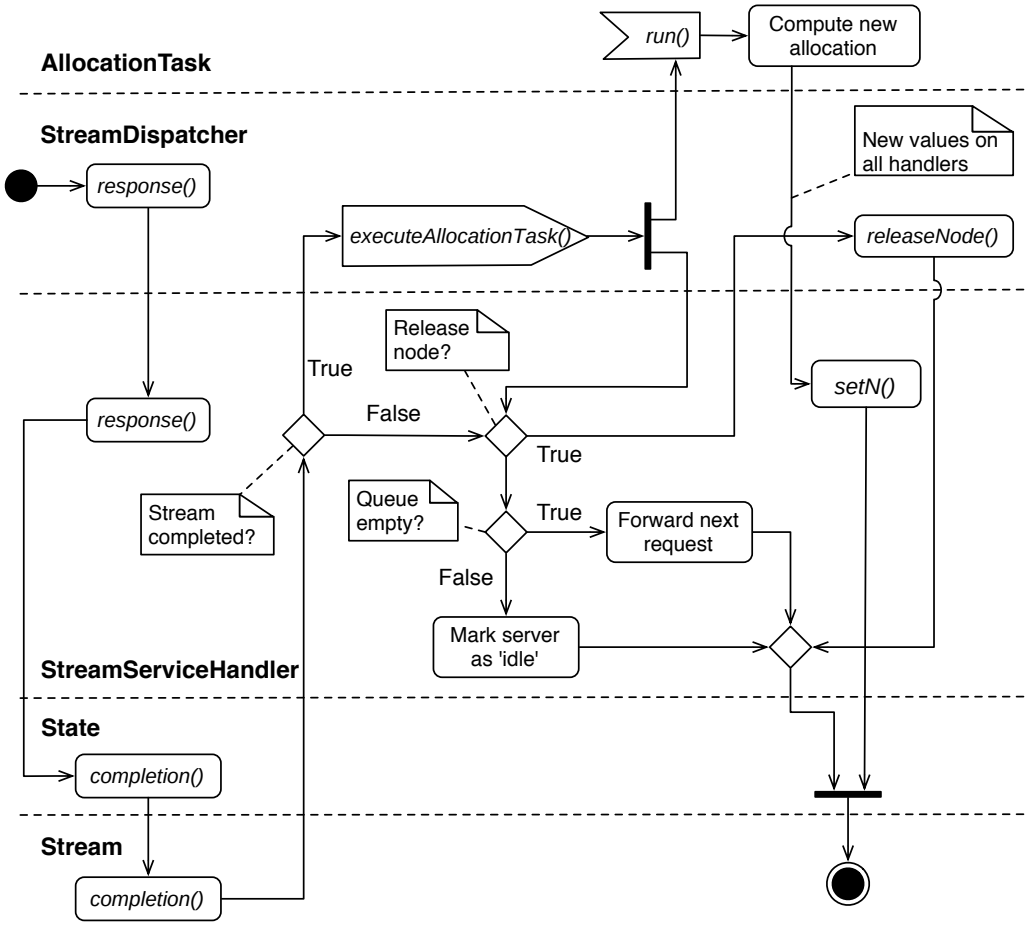


Figure 7.5: *Activity diagram for the* `response` *operation (current state, improved and long-run policies).*

7.2.2 Threshold Heuristic Implementation

The implementation of the threshold heuristic is much simpler than the one of the current-state. As depicted in Figure 7.6 the only changes between the current-state and the threshold API are the ones related to the `addStream()` operation.

To implement this admission policy, at stream arrivals of type i, the dispatcher does not need to know all the complex state information of all queues as before. Instead, it simply uses the threshold value of queue i, M_i: the new stream is accepted only if the number of active streams is less than M_i. This implies that the task previously used to take admission decisions at stream arrival events is not needed anymore. Moreover, the threshold algorithm has been implemented inside the

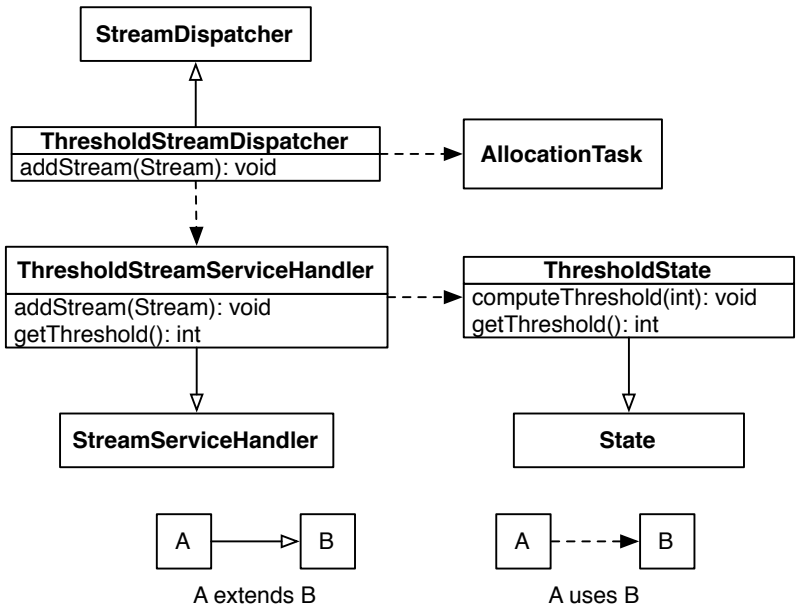


Figure 7.6: *Dispatching API for streams (threshold policy). The* `AllocationTask` *interface is defined in Figure 4.11 while the* `StreamDispatcher`, `StreamServiceHandler` *and* `State` *objects are defined in Figure 7.2.*

`ThresholdState` and it is invoked by handler i only when the number of servers allocated to it changes.

Add stream The signatures of the interface methods are the same as in the current-state case, but the behavior is different. Indeed, as depicted in Figure 7.7, it is quite similar to the `queue()` operation for single jobs.

The dispatcher reallocates servers in proportion to the offered loads every J stream arrival events. In other words, every J stream arrivals, it creates and starts an `AllocationTask` (using the usual dedicated, single-thread thread pool) in order to create a new allocation vector. Every time the `setN()` method is invoked with a new n_i value, a new threshold M_i is computed according to the algorithm described in Section 6.3.4. No matter whether a reallocation is occurring or not, the dispatcher passes the request to the handler, where the admission decision is taken: a stream of type i is admitted only if $M_i > L_i$. If the stream is accepted, the stream is added to the `ThresholdState` of type i, as before, and the cookie is sent back to the client. Thus, the cost associated to this operation is $O(1)$ if no reallocation is needed and $O(n)$ plus the evaluation of the

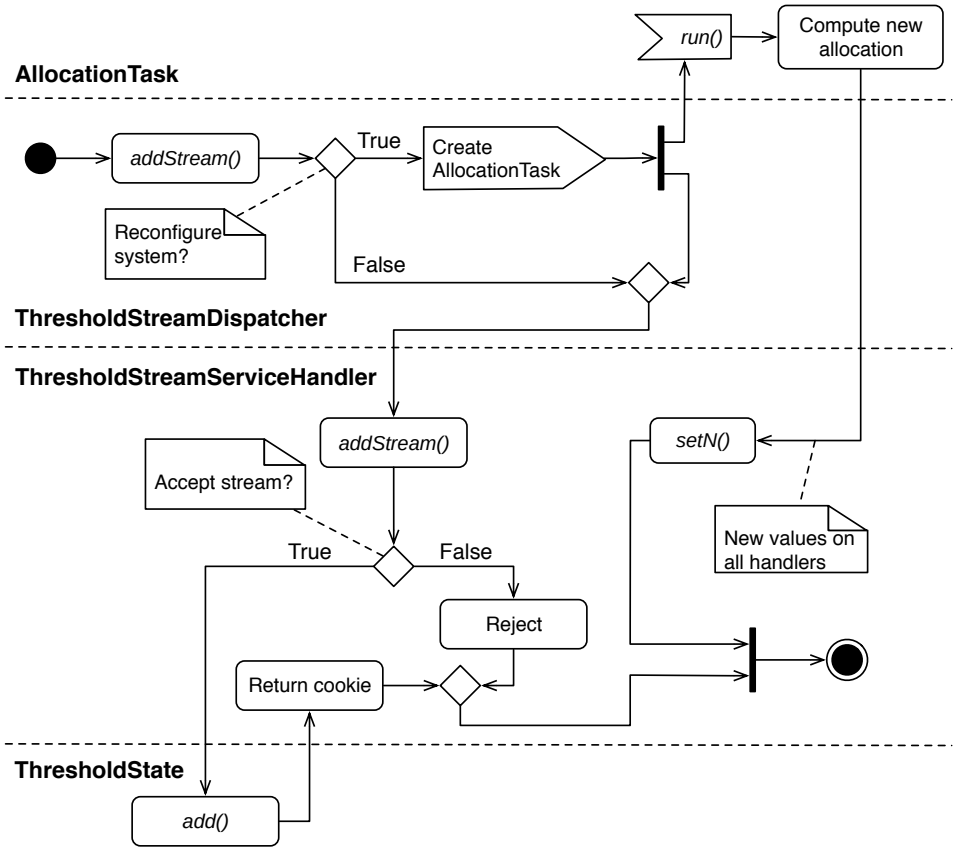


Figure 7.7: *Activity diagram for the* `addStream` *operation (threshold policy). Once computed the new allocation vector* $(n_1, \cdots, n_m)$*, the allocation task sets the new* n_i *values in operation by invoking the* `setN()` *method on each handler. This computes the new threshold values,* M_i*, too.*

threshold algorithm otherwise (this occurs every J stream arrivals), which resolves in evaluating the $g_i(\cdot)$ function defined in Section 6.3 for different values of M_i and for all queues.

Response Response events are very easy to handle if the threshold heuristic is in operation because no reallocation is necessary. As shown in Figure 7.8, the steps up to `Stream.completion()` are exactly the same as before (including the stripping of the stream from the hash table storing the active sessions, if completed). Then, no matter whether the stream has completed or not, there is a simple control to decide what to do with the server, like it occurs when the stream has not completed in Figure 7.5. Thus, the cost is exactly the same as before: $O(1)$ if the node is not released, $O(n)$ otherwise.

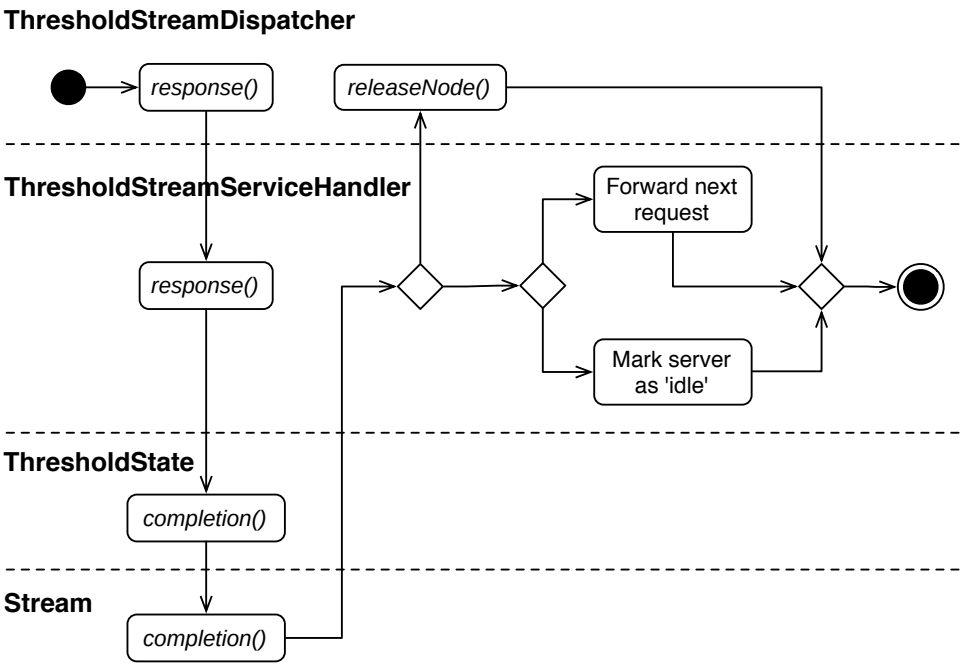


Figure 7.8: *Activity diagram for the* `response` *operation (threshold policy). Unlike the current state case shown in Figure 7.5, in this scenario no server reallocation is performed.*

7.3 Summary

This chapter has presented the extensions made to SPIRE to handle streams. Two different implementations have been discussed as the policies introduced in Chapter 6 require different approaches. The next chapter presents some experiments that were carried out using this version of SPIRE and shows that it is possible to optimize the performance of data centers by using the approach described in the previous two chapters.

Chapter 8

Experiments with Service Streams

In this chapter a 20-server cluster running the framework perviously described is employed to evaluate the effect of the admission policies for service streams introduced in Chapter 6. Unless otherwise stated, the following parameters are kept fixed:

- The QoS metric is the average waiting time, W;
- The number of jobs in each stream, k, is 50;
- The SLA states that the maximum average waiting time of jobs will be less than or equal to their average service time, *i.e.*, $q_i = b_i$;
- The hardware, network and software infrastructure is the same as used in Chapter 5. The only change was the upgrade in the Axis2 framework, from version 1.1 to version 1.3, in order to fix a few bugs affecting the previous version (mainly in the client API).

As for the analysis of the admission policies for single jobs, this chapter first assesses the performance of the framework when the exponential assumptions hold, then it shows what happens when such assumptions do not apply.

8.1 Performance when Exponentiality Assumptions are Satisfied

The first set of experiments were carried out with the aim to evaluate the effect of the various admission policies when the exponentiality assumptions for both service times and interarrival intervals apply, *i.e.*, when each subsystem can be modeled as an $M/M/n_i$ queue and thus the estimated average waiting time of a job can be estimated using equation (6.5).

Four services whose settings are shown in Table 8.1 were deployed on SPIRE. The offered load is increased by increasing the submission rate for streams of type 4. That parameter is varied in the range $\delta_4 \in (0.02, 0.2)$. At the lower end this represents a 60% loaded system, whereas at the higher end, if all streams were accepted (*i.e.*, if the 'Admit all' policy is employed), the system would be over-saturated, as the total load would be 105%.

Index	b_i	γ_i	k_i	δ_i	c_i	ρ_i
1	1.0	2.0	50	0.100	10.0	5.0
2	1.0	2.0	50	0.040	10.0	2.0
3	1.0	2.0	50	0.080	10.0	4.0
4	1.0	1.0	50	0.020 ... 0.200	10.0	1.0 ... 10

Table 8.1: *Experiment settings.* $m = 4, N = 20$.

The first experiment, shown in Figure 8.1, measures the average revenues obtained by the heuristic policies proposed in Chapter 6 when all charges and penalties are the same, *i.e.*, $c_i = r_i, \forall\, i$: if the average waiting time exceeds the obligation, users get their money back. For comparison, the effect of not having an admission policy is also displayed.

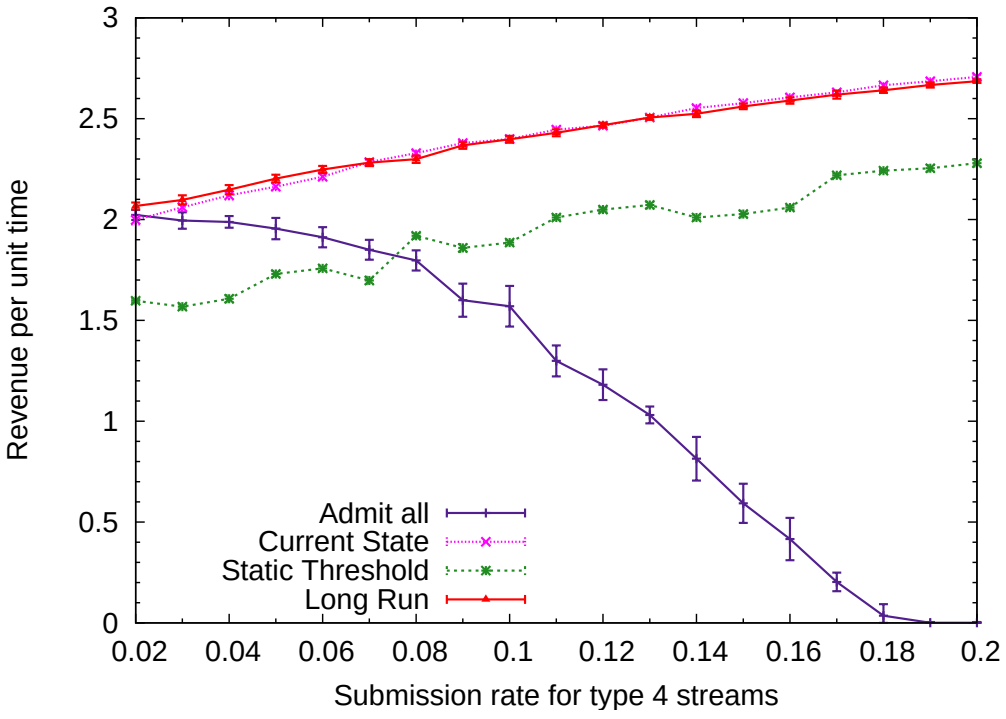


Figure 8.1: *Observed revenues for different policies (Markovian case).* $c_i = r_i, \forall\, i$.

Each point in the figure represents a SPIRE run lasting about 2 hours. In that time, between 1,400 (low load) and 1,700 (high load) streams of all types are accepted, which means that about 70,000 – 85,000 jobs go through the system. Samples of achieved revenues are collected every 10 minutes and are used at the end of the run to compute the corresponding 95% confidence interval.

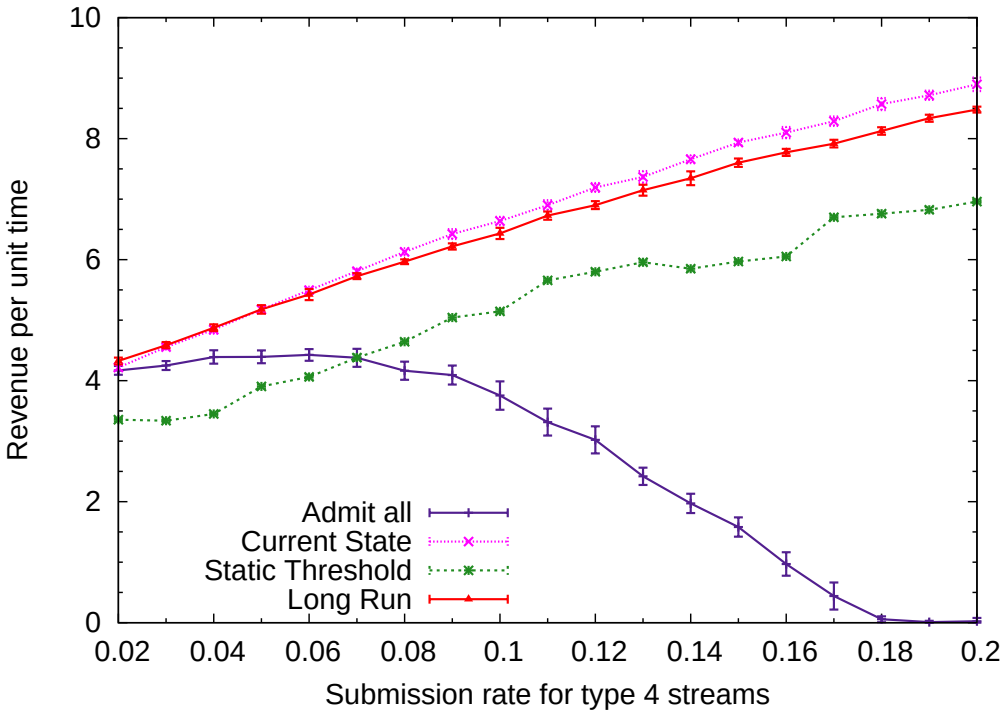


Figure 8.2: *Observed revenues for different policies (Markovian case).* $c_1 = 10, c_2 = 20, c_3 = 30, c_4 = 40, r_i = c_i$.

The most notable feature of this graph is that while the performance of the 'Admit all' policy becomes steadily worse as soon as the load increases and drops to 0 when it approaches the saturation point, the heuristic algorithms produce revenues that grow with the offered load. They achieve that growth not only by accepting more streams, but also by rejecting more streams at higher loads. Also, it is worth noting that the 'Long-Run' policy performs almost as well as the complex 'Current State' one, with the values between each other's confidence interval. Since, apart from the random traffic fluctuations, the average load does not change inside the same run, the settings for the 'Threshold' heuristic use an infinite configuration interval (*i.e.*, $J = \infty$) since, according to the experiment shown in Figure 6.11, that configuration is the one that guarantees the better performance when the load does not change. This means that, at time 0, SPIRE partitions the available machines

across the four queues and computes the threshold values. Then, the allocation and the threshold vector are not updated any more. When the load does not change over time, the revenues achieved by the static Threshold policy are within 20% of the ones obtained by the more complex algorithms. The last set of experiments of this chapter will show the effects of different window size values on the maximum achievable revenues when the load changes over time.

The second result concerns a similar experiment, except that now charges and related penalties differ between each queue: $c_1 = 10$, $c_2 = 20$, $c_3 = 30$ and $c_4 = 40$, $c_i = r_i$. The main difference compared to the previous experiment is that now it is more profitable to run, say, jobs of type 4 than jobs of type 3. Figure 8.2 shows that the maximum achievable revenues are now much higher than before in virtue of the higher charge values for type 2, 3 and 4 streams. Moreover, the Long Run heuristic still performs very well, while the difference between the Current State and the Threshold policies is about 25%.

In the third experiment the charges are the same as in Figure 8.2, however if the SLA is not met, users get back twice what they paid, *i.e.*, $r_i = 2c_i$.

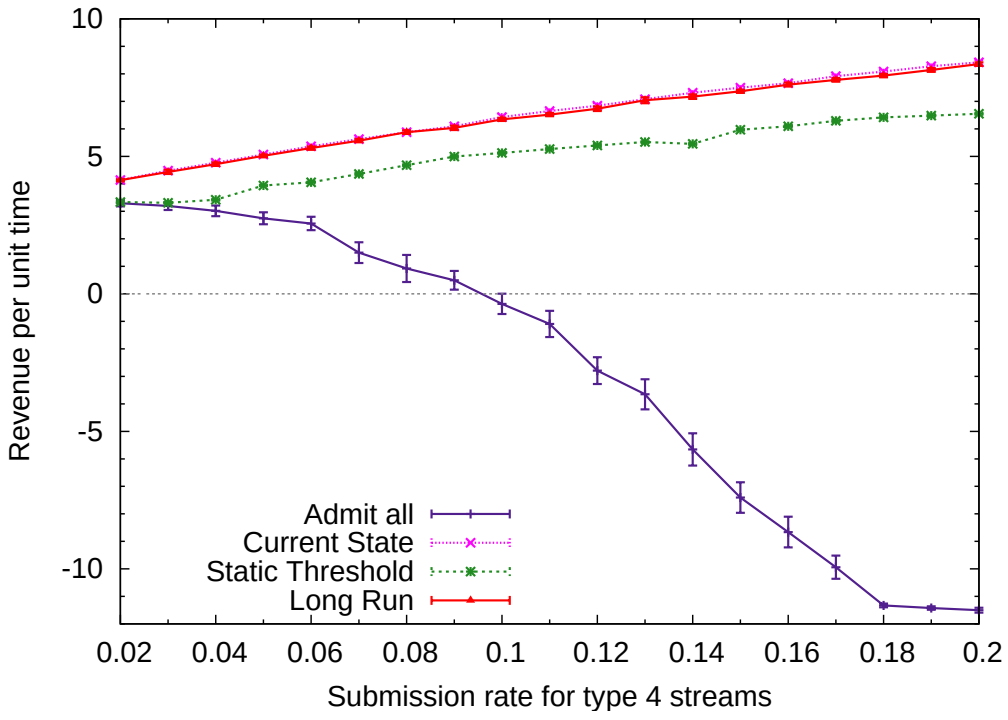


Figure 8.3: *Observed revenues for different policies (Markovian case, $r_i = 2c_i$).*

The most notable feature of the graph shown in Figure 8.3 is that now the revenues obtained by

the Admit all policy become negative as soon as the load starts increasing because penalties are very punitive. The Threshold policy performs very well if compared to the other heuristics: according to the results of the simulations carried out in Chapter 6 such policy is very conservative. This means that is much more likely that the Threshold policy rejects a stream than if it fails to meet the promised QoS.

8.2 Performance without Exponentiality Assumptions: Bursty Arrivals

In the second set of results, the Poisson job arrival processes are replaced with bursty arrivals. More precisely, if the overall arrival rate for jobs of a given type is γ, then 80% of the interarrival intervals are on the average $1/(5\gamma)$, and 20% are 4.2γ. This increases the squared coefficient of variation of interarrival times to 6.12. The aim of increasing variability is to make the system less predictable and decision making more difficult. It is worth stressing that it is not the stream submission rate, δ_i, which is the one subject to bursts, but the arrival rate, γ_i, at which jobs belonging to a stream arrive.

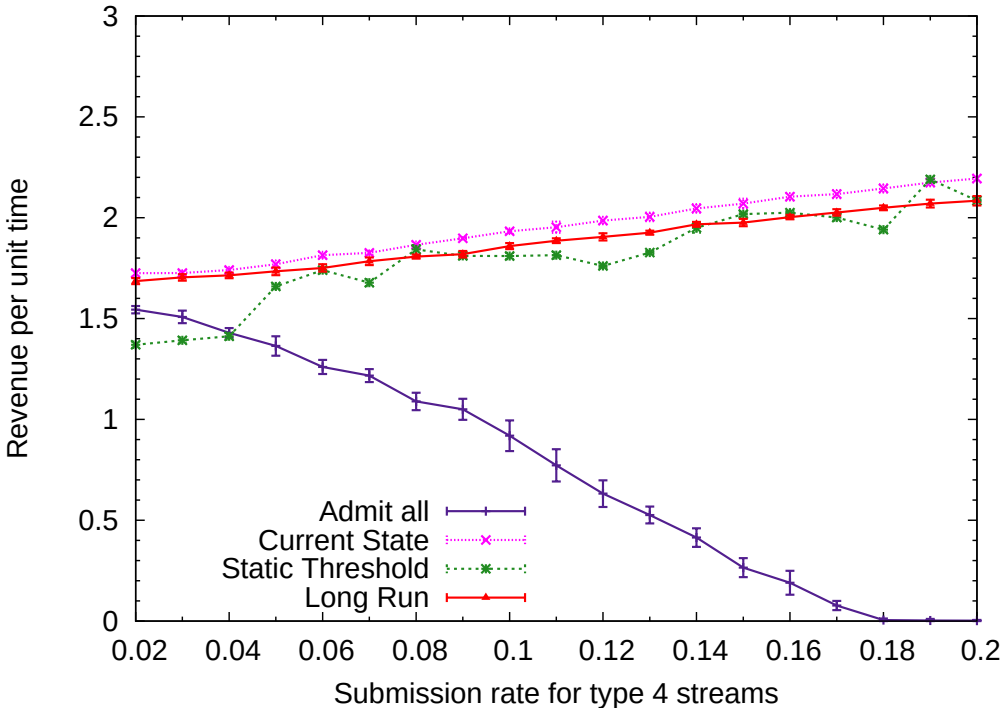


Figure 8.4: *Observed revenues for different policies ($ca^2 = 6.12$, $c_i = r_i$, $\forall i$).*

In Figure 8.4 the four job types have all the same economic parameters. The increased unpredictability caused by the bursty arrivals have some effect on the obtained revenues compared to the

Markovian case, with the Threshold heuristic performing almost as well as the Current State policy.

The same occurs also when some jobs are more expensive than others, like in Figure 8.5. The Current State heuristic performs about 20% worse than when the interarrival intervals are distributed exponentially. The other two policies, instead, perform almost the same, with the increased unpredictability that seems to have little effect on the Threshold algorithm.

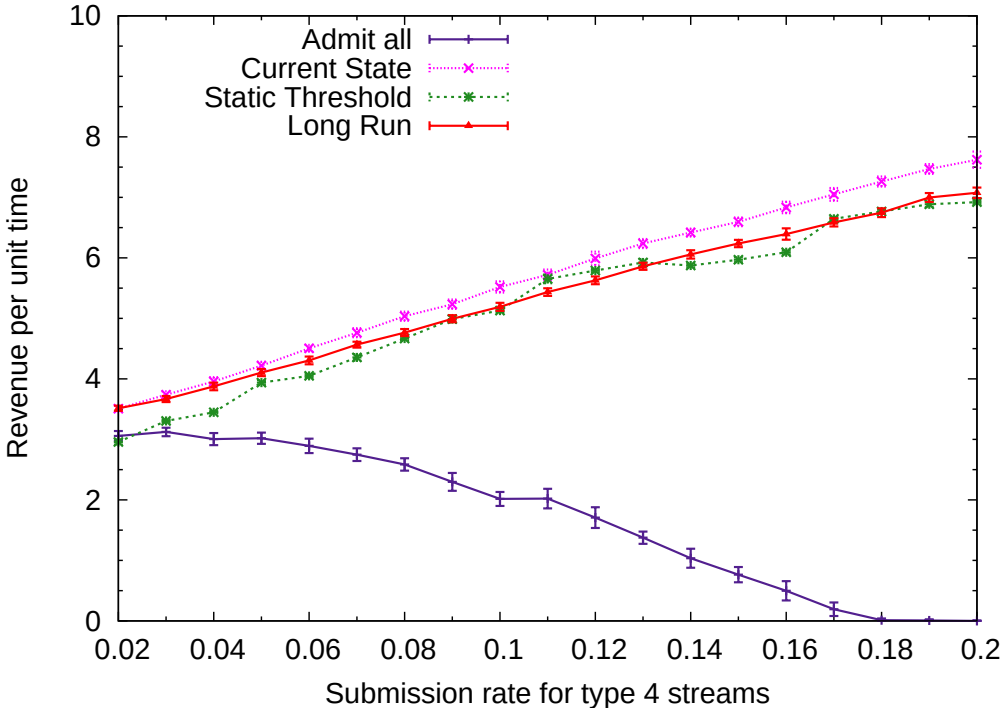


Figure 8.5: *Observed revenues for different policies ($ca^2 = 6.12$, $c_i \neq c_j$).*

Finally, when the penalties are higher than the related charges, like in Figure 8.6, the shape of the revenues is similar to the previous try (apart from the Admit all algorithm): the Current State and the Long Run heuristics are very close in term of performance, however the revenues they obtain are about 20% smaller than the Markovian case. Again, the increased variability has a little impact on the Threshold policy.

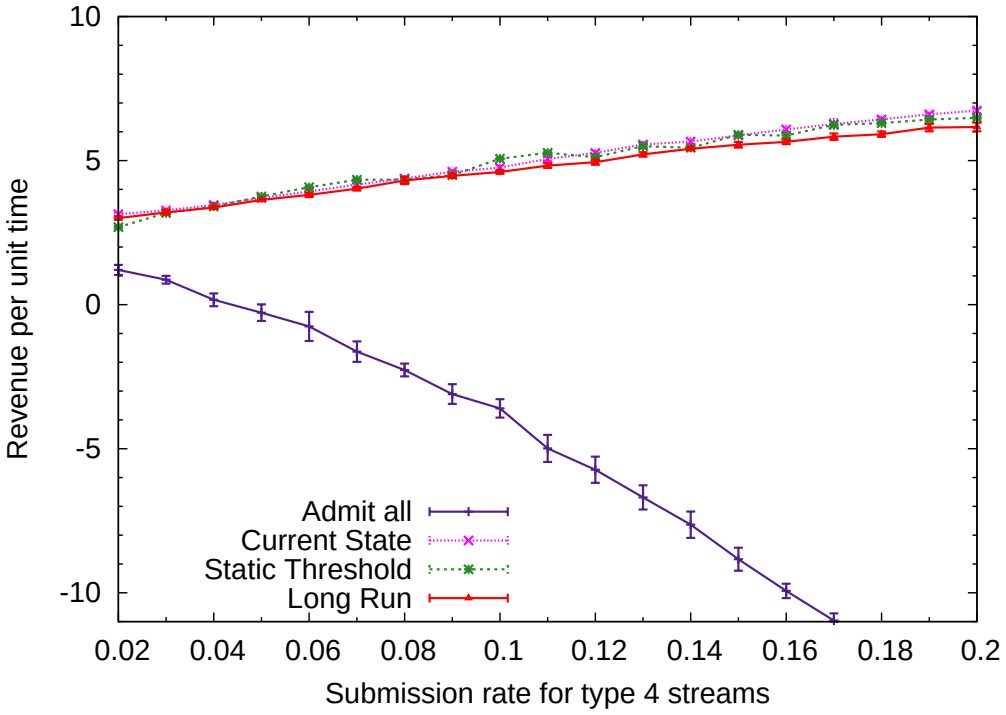


Figure 8.6: *Observed revenues for different policies ($ca^2 = 6.12$, $r_i = 2c_i$).*

8.3 Performance without Exponentiality Assumptions: Hyperexponentially Distributed Service Times

The thrid set of results concerns a scenario where service times are not distributed exponentially. A higher variability is introduced by generating jobs with hyperexponentially distributed service times: 80% of them are short, with mean service time 0.2 seconds, and 20% are much longer, with mean service time 4.2 seconds. The overall average service time is 1, as before, but the squared coefficient of variation of service times is now 6.15.

When all streams have the same charges and penalties, like in Figure 8.7, the Threshold heuristic performs almost as well as in the Markovian case (probably because of its very conservative behavior), and is very close to the performance of the other policies. The behavior of the Admit all policy is, as usual, very bad as soon as the load starts increasing.

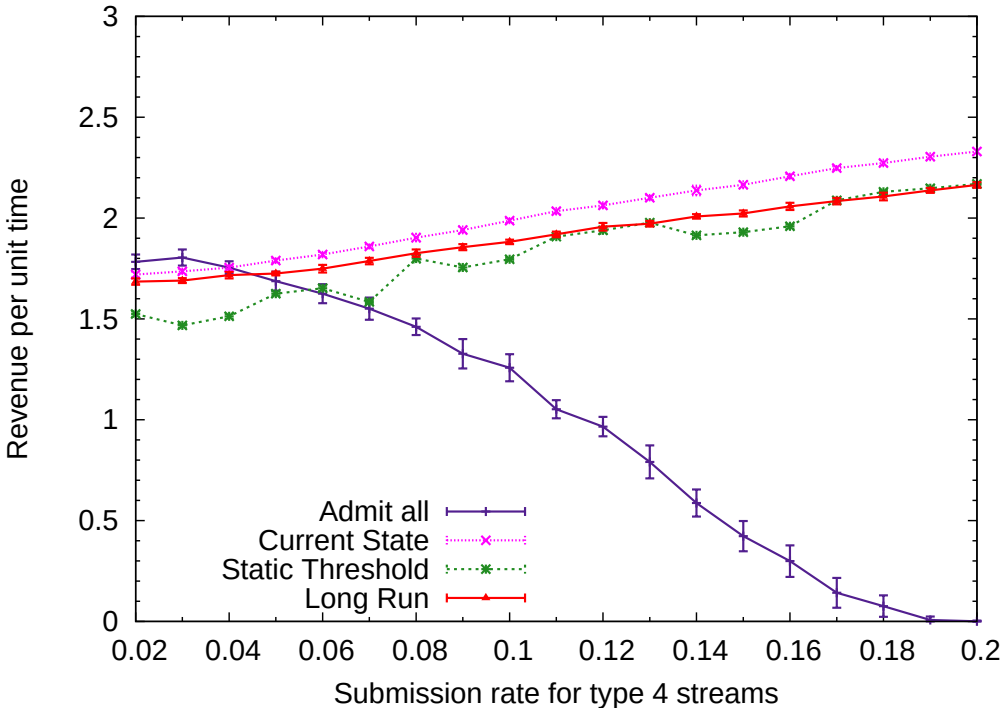


Figure 8.7: *Observed revenues for different policies ($cs^2 = 6.15$, $c_i = r_i$, $\forall i$).*

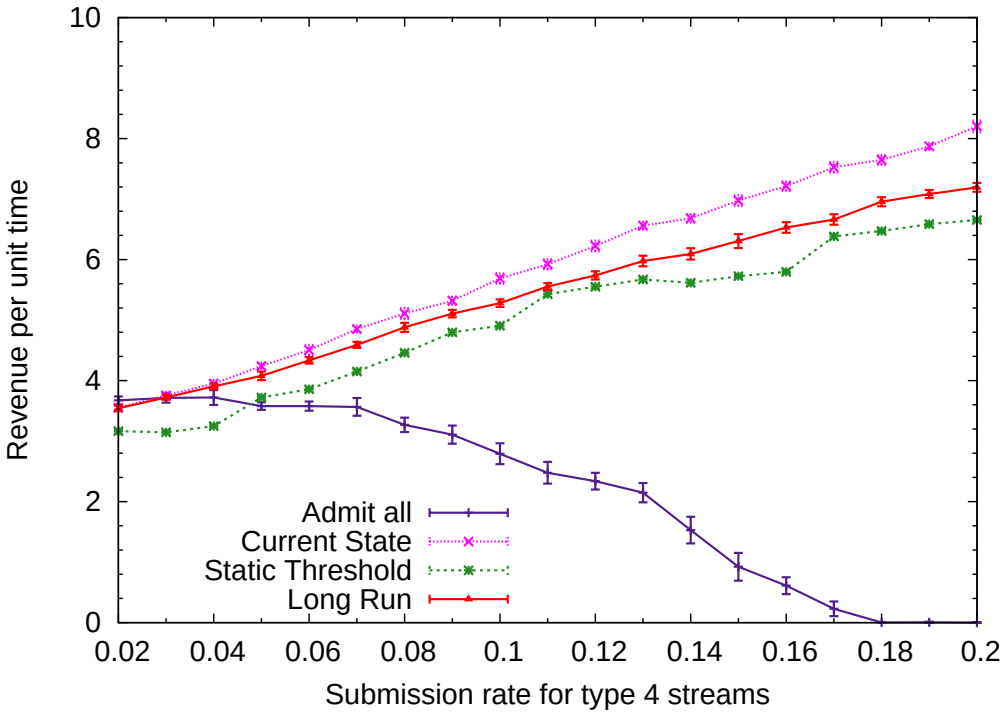


Figure 8.8: *Observed revenues for different policies ($cs^2 = 6.15$, $c_i \neq c_j$).*

When some jobs are more expensive than other (Figure 8.8) the Current State and the Threshold heuristics perform almost as in Figure 8.2, while the Long Run policy performs about 15% worse than the Current State.

Finally, Figure 8.9 shows that when the penalties are higher than the related charges, the behavior of the three policies is similar. The Current State and Long Run algorithms performs worse than in the Markovian case, while the wise Threshold heuristic performs almost the same way.

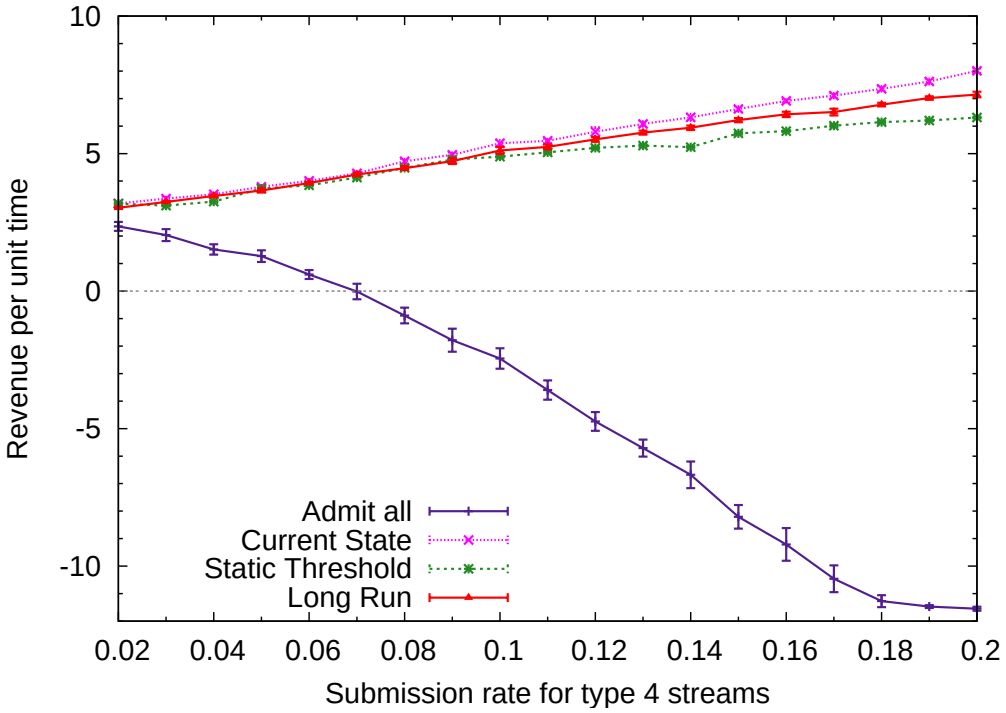


Figure 8.9: *Observed revenues for different policies ($cs^2 = 6.15$, $r_i = 2c_i$).*

8.4 Variable Load

So far, so good. The experiments discussed in the first part of this chapter tested the behavior of the heuristics under different conditions. However, the loading conditions did not change over the time, while it is well known that the volume of demand in production application environments fluctuates on several time scales (*i.e.*, daily and monthly cycles). The last set of experiments were carried out with the aim to test the robustness of SPIRE and the proposed algorithms when the load is non-stationary. The total load is the same as before, *i.e.*, ρ ranges between 60% and 105% by varying the

rate at which type 4 streams arrive to the system. However, every x seconds, δ_1 and δ_2 are swapped. In other words, during period 1 $\delta_1 = 0.1$ and $\delta_2 = 0.04$, during period 2 $\delta_1 = 0.04$ and $\delta_2 = 0.1$, and so on. As consequence, the loads for queues 1 and 2 fluctuate between 2 and 5. The other parameters are the same as in Figure 8.1: $c_i = r_i = 10$, $\forall i$, while the service time and interarrival interval distributions are exponentially distributed. For comparison reasons, the figures include an 'Oracle' Threshold policy, that knows exactly when the load variations occur and recomputes the allocation and admission threshold vectors.

In the first experiment, shown in Figure 8.10, the stream submission rates change every 300 seconds.

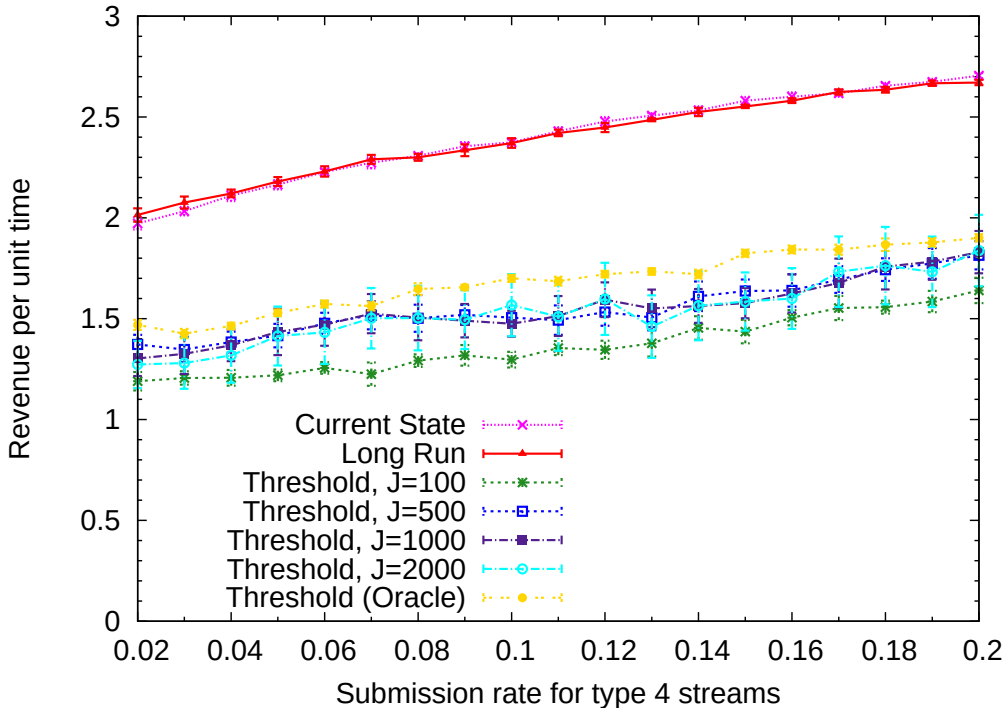


Figure 8.10: *Observed revenues for different policies with variable load (ρ_1 and ρ_2 change every 300 seconds).*

Because the allocation and admission decisions are taken every time a stream arrives or completes, the Current State algorithm is not affected by the changes in δ_1 and δ_2, and so it is capable to perform as well as when the load does not change (see Figure 8.10). The Long-Run heuristic, which is based on the assumption that queue i is subjected to a constant load of L_i streams, is robust enough to cope with load variations, and in fact it performs like in the constant load scenario. Finally, the revenues obtained by the Threshold policies are between 72% and 83% of the ones achieved under

constant load. The main reason is that the Threshold policy uses traffic estimates to compute the vector of admission thresholds: the Oracle does it by using the δ_i values, while the ones that periodically recompute the two vectors estimate the service times, arrival rates as well as the squared coefficient of variations of service times and interarrival intervals. As it has already been pointed out earlier in this thesis, this policy exhibits a very conservative behavior, which is further emphasized by the fact that the steady state is not reached. From a practical point of view, the Threshold algorithm does not seem very sensible with respect to the employed window size: using a bigger window size allows to system to achieve a slightly higher revenue (close to the oracle), but at the expense of a bigger confidence interval.

In the experiment shown in Figure 8.11 the two rates δ_1 and δ_2 are swapped every 60 seconds. As before, the Current State and Long-Run policies are not affected at all, while the Threshold seems to perform slightly better with medium sized windows.

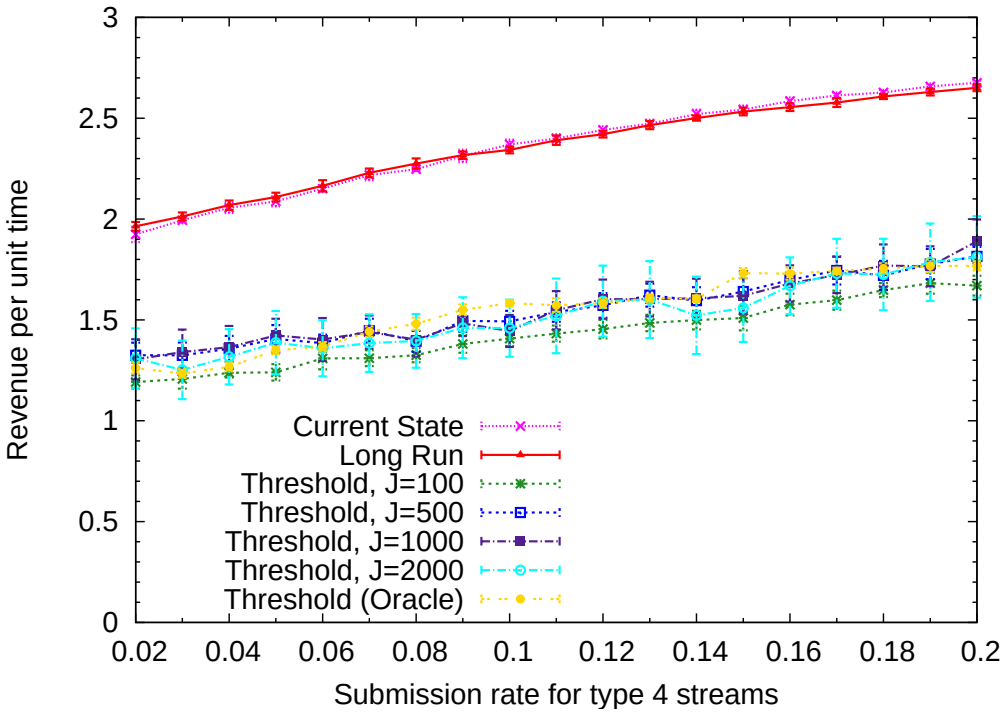


Figure 8.11: *Observed revenues for different policies with variable load (ρ_1 and ρ_2 change every 60 seconds).*

Finally, when the loads change every 10 minutes, no changes seem to occur compared to the first case. If a very short configuration interval is used the system cannot estimate the working

parameters thoroughly, and so the achieved revenue is a bit worse. Again, it is worth noting that using a larger window size improves the general performance, but the confidence intervals grow too.

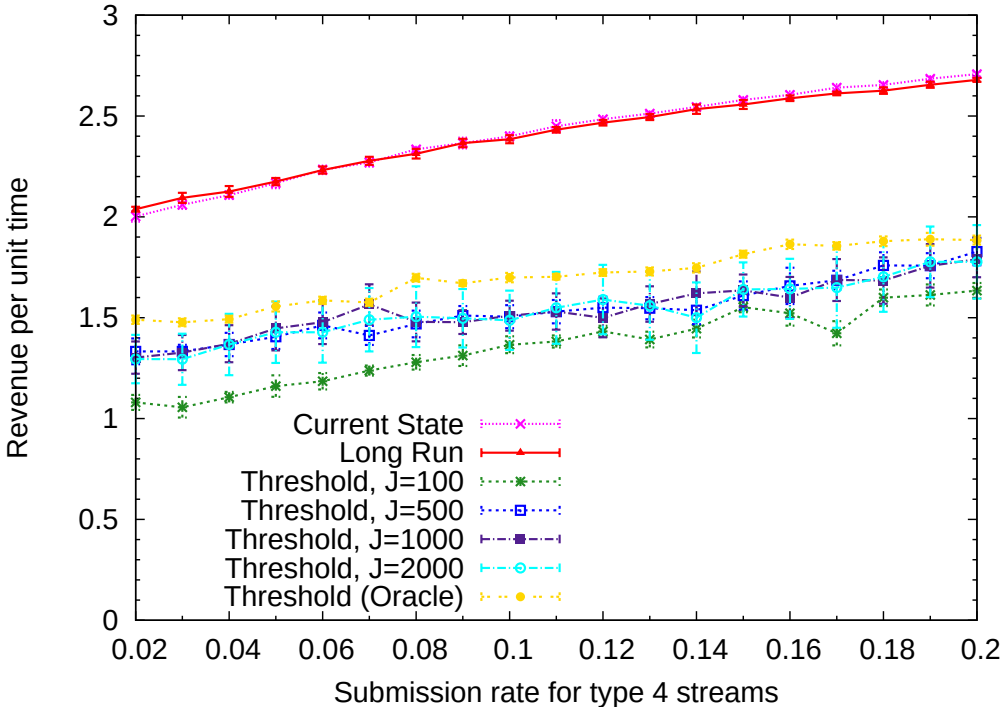


Figure 8.12: *Observed revenues for different policies with variable load (ρ_1 and ρ_2 change every 600 seconds).*

8.5 Summary

This chapter has presented several experiments aimed at evaluating the effects of allocation and session-based admission policies on the performance of the SPIRE system. The results have demonstrated that such algorithms have a significant impact on the earned revenue. From a practical point of view, the nonexistence of assumptions about the nature of service demand makes the system very robust.

Chapter 9

Conclusions

This chapter ends the dissertation with some concluding remarks, then it presents some insights on possible extensions and directions for future research.

9.1 Concluding Remarks

This thesis has proposed adaptive algorithms for maximizing the efficiency of service provisioning systems subject to QoS contracts. The dissertation has introduced two quantitative frameworks whose performance is measured in terms of the average revenue earned per unit time, and a middleware platform that provides an appropriate infrastructure. In order to approach the revenue maximization problem, a number of models have been formulated and analyzed in order to evaluate a variety of operating strategies for the efficient deployment of computing resources. The experiments that were carried out have demonstrated that policy decisions such as server allocations and admission choices can have a significant effect on the revenue. Moreover, those decisions are affected by the contractual obligations between clients and provider in relation to the service quality. The analysis outlined in this thesis has been broken down into two distinct areas.

Chapter 3 deals with the problem of how to structure and control a service provisioning system with contractual obligations involving single jobs: users pay for having their jobs run, but demand in turn a certain QoS. The model assumes that the performance is expressed in terms of either response time or waiting time, but other metrics could also be adopted. Also, the provider agrees to pay a penalty whenever the response time (or waiting time) of a job exceeds a certain bound. Since overload is an inevitable aspect of Internet services and over-provisioning is not always feasible (load spikes can be an order of magnitude greater than average), the provision of a service includes algorithms to (i) reconfigure the available resources to respond to changes in patterns of demand,

and (ii) to take admission decisions. Once the contractual obligations are known, it is the provider's responsibility to decide how to allocate the available resources and when to accept jobs, in order to make the system as profitable as possible. Chapter 3 describes dynamic heuristic policies for server allocation and job admission. The proposed algorithms are based on the notion of configuration intervals, or windows: the system collects traffic statistics and periodically updates its configuration in order to adapt to changes in user demand. As shown in Chapter 4, an important feature from a practical point of view is that the proposed policies are not very sensitive with respect to the window size that is used in their implementation. Moreover, even though the analysis assumes that jobs arrive according to an independent Poisson process, while service times are distributed exponentially, the algorithms are robust enough to be able to cope with bursty arrivals and non-exponential service times.

A natural generalization of the revenue maximization problem discussed in Chapter 3 is introduced in Chapter 6. The previous model applies charges, penalties, obligations and admission decisions to individual jobs. Unfortunately the admission control scheme that makes admission decisions on every job entering the system cannot be employed to deal with session-based traffic (some sessions might be broken or delayed at critical stages, such as checkout). The idea is the same as before, however now SLAs and admission decisions apply to groups of jobs, referred to as streams, rather than single requests. Users agree to pay a specified amount for each accepted and completed stream, and also to submit the jobs in it at a specified rate. The provider promises to run all jobs in the stream, and also to pay a penalty whenever the average performance for the stream falls below a certain limit. The described policies are still based on dynamic estimates of traffic patterns and models of system behaviour. However now the emphasis of the latter is on generality rather than analytical tractability. Thus, interarrival and service times are not required to be exponentially distributed. Instead, they are allowed to have general distributions with finite coefficient of variation. To handle the resulting models it is necessary to use approximations, however those approximations lead to policies that perform well and can be used in real systems.

The validity of the mathematical models described for dynamic server allocation and admission control have been demonstrated with the design and implementation of a prototype service provisioning system, called SPIRE. This platform was initialized as part of the research project QoSP and is described in Chapters 4 and 7. The prototype is a self-configurable and self-optimizing middleware platform that dynamically migrates servers among hosted services and enforces SLAs by monitoring traffic parameters, income and expenditure. It provides a test bed for experimentation

with different operating policies and different patterns of demand. The system has been deployed and tested extensively: the results show that the proposed allocation and admission policies perform well under several different traffic conditions.

9.2 Future Work

The scope for future work is rather large and includes possible directions:

- Instead of operating an admission policy, one might have a contract specifying the maximum rate at which users may submit jobs (or streams) of a given type, and then be committed to accepting all submissions. The question would then arise as to what that maximum rate should be.

- Instead of having a dedicated architecture, it may be possible to share a server among several types of services, eventually by using virtual machines to guarantee some form of isolation between the running applications. The resulting model would be completely different as, for example, there would be no need of private servers pool. However one would have to consider different job scheduling strategies, *e.g.*, preemptive and non-preemptive priorities, Round-Robin, etc.

- The models discussed in this dissertation assume that system reconfigurations are immediate or take a negligible amount of time. However, switching a server from one type of service to another may incur non-negligible costs in either money or time. Taking those costs into account would mean dealing with a much more complex dynamic optimization problem.

- The models proposed in Chapters 3 and 6 assume that servers execute at most one request at any given point. Experiments have shown that, thanks to the use of dynamic policies, the system performs well, even when heavily loaded. However this hypothesis might lead to sub-optimal performance. First of all, jobs may not be CPU bound, but require access to other, potentially remote, services, databases, etc. Second, recent advances in microprocessor architectures, and multi-cores in particular, make it possible to efficiently run several jobs in parallel, even if they are CPU bound. Thus, the idea is to further improve the efficiency of data centers by running concurrent jobs subject to SLAs. Of course, since there SLAs in operation, it is not possible to change the concurrency level at random: the idea is to maintain the same QoS level (*i.e.*, waiting or response time guarantees) as if jobs were run

alone by a performance management model that monitors the system's behaviour at runtime and modifies the degree of multiprogramming accordingly.

- So far the provisioning system was composed of one cluster only. When more than one cluster is involved, questions of routing of demands will also arise. Service provisioning systems are often made of several, geographically distributed, clusters (*i.e.*, content delivery networks). The reasons are manifold, but mainly because power consumption limits the size of data centers and because nodes deployed in multiple locations can optimize the service provision, for example by reducing bandwidth costs, improving end-user performance (latency is lower if the content or service is closer to the user), or increasing availability (a major breakdown at one location would not affect the others). The different clusters may wish to cooperate with each other in servicing user requests or they may be competitors trying to attract as large a fraction of the market as possible. This adds another dimension to both the SLA formulation and operating policies as well as the structure of the provisioning system.

- Power consumption is a major problem for data centers of all sizes and it impacts the density of servers and the total cost of ownership. This is causing changes in data center configuration and management. To stress out how important the problem is, just think that data centers are limited by power: it is estimated that in 2010, for each dollar spent on servers, $0.71 will be spent for power/cooling, while for each Watt consumed for power, data centers consume another Watt for cooling. Finally, it has been reported that Facebook, the company running the popular social networking web site, is spending well over $1 million a month on electricity alone [12].

  The problem can be stated in the following terms: when a server is in power up, it can service incoming requests. When a server is in power down, it cannot process any job, but will consume less (or no) energy. Thus it would be interesting to extend the revenue maximization problems introduced in the previous chapters by taking into account the cost for running the servers and assuming that the machines can be switched on and off dynamically: admission and allocation polices would still prevent the provisioning system from being overloaded, while power management policies would help reducing the operating costs.

Appendix A

Exchanged Messages

This appendix shows the SOAP messages exchanged between the parties to implement the framework described in Chapter 4.

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:node="http://org.ncl.ac.uk/qosp"
  xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  <soapenv:Header>
    <node:qosp>
      <node:messageType>
        <node:value>New node</node:value>
      </node:messageType>
    </node:qosp>
  </soapenv:Header>
  <soapenv:Body>
    <node:addNode>
      <node:name>127.0.0.1:8080</node:name>
    </node:addNode>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure A.1: *Example of a* `addNode` *message. The endpoint for the Service*$_1$ *will be* `http://127.0.0.1:8080/Service1`*, the one for Service*$_2$ *will be* `http://127.0.0.1:8080/Service2`*, etc. All the extra information (i.e. addressing metadata) are omitted.*

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:node="http://org.ncl.ac.uk/qosp"
    xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
   <soapenv:Header>
          <codestore:qosp>
          <codestore:MessageType>
              <codestore:value>New service</codestore:value>
          </codestore:MessageType>
      </codestore:qosp>
  </soapenv:Header>
  <soapenv:Body>
      <codestore:addService>
        <service name="Service1">
           <!-- the obligation is specified in ms precision -->
          <qos charge="200" penalty="1000" obligation="10000" />
        </service>
        <service name="Service2">
          <qos charge="100" penalty="500" obligation="2000" />
        </service>
        <!-- eventually other services here -->
      </codestore:addService>
      <!-- eventually other stuff here -->
  </soapenv:Body>
</soapenv:Envelope>
```

Figure A.2: *Example of a* `addService` *message. The economic parameters for Service*$_1$ *are* (c_1, q_1, r_1) *= (200, 10.0, 1000) while the ones for Service*$_2$ *are* (c_2, q_2, r_2) *= (100, 2.0, 500) All the extra information (i.e. addressing metadata) are omitted.*

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:wsa="http://www.w3.org/2005/08/addressing" xmlns:soapenv
    ="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Header>
    <node:qosp xmlns:node="http://ncl.ac.uk/qosp">
      <node:messageType xmlns:node="http://ncl.ac.uk/qosp">Result</
          node:messageType>
      <node:serviceInfo xmlns:node="http://ncl.ac.uk/qosp">
        <node:node xmlns:node="http://ncl.ac.uk/qosp">128.240.149.146:18080</
            node:node>
        <node:service xmlns:node="http://ncl.ac.uk/qosp" node:deployed="false">
            FunctionalTestService1</node:service>
        <node:messageID xmlns:node="http://ncl.ac.uk/qosp">
            urn:uuid:4BB08A85FDC224361C11570174738421</node:messageID>
        <node:actionMapping xmlns:node="http://ncl.ac.uk/qosp">urn:test1</
            node:actionMapping>
      </node:serviceInfo>
      <node:timeInfo xmlns:node="http://ncl.ac.uk/qosp">
        <node:serviceTime xmlns:node="http://ncl.ac.uk/qosp">12815</
            node:serviceTime>
        <node:arrivalTime xmlns:node="http://ncl.ac.uk/qosp">1157017475111</
            node:arrivalTime>
      </node:timeInfo>
    </node:qosp>
    <wsa:To>http://giga18.ncl.ac.uk:8081/axis2/services/RoutingService</wsa:To>
    <wsa:ReplyTo>
      <wsa:Address>http://10.8.150.50:6060/axis2/services/annonService13526262/
          annonOutInOp</wsa:Address>
    </wsa:ReplyTo>
    <wsa:FaultTo>
      <wsa:Address>http://128.240.149.146:18080/axis2/services/
          FunctionalTestService1</wsa:Address>
    </wsa:FaultTo>
    <wsa:MessageID>urn:uuid:A02723EDF24620304D11570174758073</wsa:MessageID>
    <wsa:Action>urn:result</wsa:Action>
    <wsa:RelatesTo wsa:RelationshipType="http://www.w3.org/2005/08/addressing/
        reply">urn:uuid:7DD37A66519B8626AB11570174751591</wsa:RelatesTo>
  </soapenv:Header>

  <soapenv:Body>
    <!--
        The result is stored here. It is not used by SPIRE; it is
        forwarded "as-it-is" to the client
        -->
  </soapenv:Body>
</soapenv:Envelope>
```

Figure A.3: *Example of* `result` *message. The final destination (i.e. the client endpoint) is stored into the* `ReplyTo` *block. The* `qosp` *block contains the servers that executed the request, the target service, the service time and other values used to update the state of the controller.*

```
<manager:qosp>
  <manager:messageType>
    <manager:value>Forward</manager:value>
  </manager:messageType>

  <!-- This element specifies wether the handler has to record the -->
  <!-- service time or not. -->
  <manager:waitingTime manager:needed="true|false" manager:sentAt="10552544875">
    <!-- This value is present only if needed is set to 'true' -->
    <manager:arrivalTime>11552944875</manager:arrivalTime>
  </manager:waitingTime>

  <manager:clientEPR>
    <!-- The client endpoint, i.e., where the response has to be sent -->
    <manager:value>
      http://10.8.149.156:6060/axis2/services/__ANONYMOUS_SERVICE__/
          __OPERATION_OUT_IN__
    </manager:value>
  </manager:clientEPR>
  <!-- Request ID, used by the client to correlate request and response -->
  <!-- (set by the Axis2 engine) -->
  <manager:messageID>urn:uuid:4BB08A85FDC224361C11570174738421</
      manager:messageID>
  <!-- Request identifier (set by SPIRE) -->
  <manager:requestID>45356L</manager:requestID>
  <!-- This values is included only if the request is part of a Stream -->
  <manager:cookie>456L</manager:cookie>
</manager:qosp>
```

Figure A.4: *Structure of the portion of SOAP header containing the state of the request. The client endpoint, that is the endpoint where the final result will be sent, is contained into the* `clientEPR` *block, while the* `messageID` *block contains the message identifier of the request. This will be used by the controller to correlate the request and the response.*

```
<?xml version='1.0' encoding='utf-8'?>
<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
  <soapenv:Header>
    <client:something xmlns:client="http://ncl.ac.uk/qosp" />
  </soapenv:Header>
  <soapenv:Body>
    <client:addStream xmlns:client="http://ncl.ac.uk/qosp">
      <client:gamma>0.2f</client:gamma>
      <client:length>10000</client:length>
      <client:service>Service1</client:service>
    </client:addStream>
  </soapenv:Body>
</soapenv:Envelope>
```

Figure A.5: *Example of* `addStream` *request (addressing information not shown). It is also possible to evaluate requests with custom economic parameters and service times (not if the threshold policy is employed).*

Bibliography

[1] Amazon EC2 Service Level Agreement, October 2008. URL http://aws.amazon.com/ec2-sla/.

[2] Amazon Elastic Computing Cloud (EC2). URL http://aws.amazon.com/ec2/.

[3] IBM Launches New Cloud Computing Consulting and Implementation Services, November 2008. URL http://www-03.ibm.com/press/us/en/pressrelease/26168.wss.

[4] What is managed hosting?, 2008. URL http://www.rackspace.co.uk/whatismanagedhosting/.

[5] Tarek F. Abdelzaher and Nina Bhatti. Web Content Adaptation to Improve Server Overload Behavior. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 31 (11-16):1563–1577, 1999. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/S1389-1286(99)00031-6.

[6] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions*. National Bureau of Standards, Washington D.C., 10th printing edition, 1972.

[7] Amazon. Amazon Web Services, 2008. URL http://aws.amazon.com/.

[8] Yair Amir, Baruch Awerbuch, and R. Sean Borgstrom. A cost-benefit framework for online management of a metacomputing system. In *Proceedings of the first international conference on Information and computation economies (ICE '98)*, pages 140–147, New York, NY, USA, 1998. ACM. ISBN 1-58113-076-7. doi: http://doi.acm.org/10.1145/288994.289026.

[9] Mikael Andersson, Martin Höst, Jianhua Cao, Christian Nyberg, and Maria Kihl Palm. Design and Evaluation of an Overload Control System for Crisis-Related Web Server Systems. In *International Conference on Internet Security and Protection (ICISP 2006)*, August 2006.

[10] Artur Andrzejak, Martin Arlitt, and Jerry Rolia. Bounding the Resource Savings of Utility Computing Models. Technical Report HPL-2002-339, Internet Systems and Storage Laboratory, HP Laboratories, December 2002. URL http://www.hpl.hp.com/techreports/2002/HPL-2002-339.pdf.

[11] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D.P. Pazel, J. Pershing, and B. Rochwerger. Océano-SLA based management of a computing utility. In *Proceedings of the IEEE/IFIP International Symposium on Integrated Network Management*, pages 855–868, May 2001. doi: 10.1109/INM.2001.918085.

[12] Michael Arrington. Facebook May Be Growing Too Fast. And Hitting The Capital Markets Again. TechCrunch, October 2008. URL `http://www.techcrunch.com/2008/10/31/facebooks-growing-problem/`.

[13] Xin Bai, Dan C. Marinescu, Ladislau Bölöni, Howard Jay Siegel, Rose A. Daley, and I-Jeng Wang Wang. A macroeconomic model for resource allocation in large-scale distributed systems. *Journal of Parallel and Distributed Computing*, 68(2):182–199, February 2008. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/j.jpdc.2007.07.001. Academic Press, Inc.

[14] N. Bartolini, G. Bongiovanni, and S. Silvestri. Self-* Overload Control for Distributed Web Systems. In *Proocedings of the 16th International Workshop on Quality of Service (IWQoS 2008)*, pages 50–59. IEEE Computer Society, June 2008. doi: 10.1109/IWQOS.2008.11.

[15] BBC News. Sun offers processing by the hour, February 2005. URL `http://news.bbc.co.uk/1/hi/technology/4229021.stm`.

[16] Mohammed N. Bennani and Daniel Menascé. Resource Allocation for Autonomic Data Centers Using Analytic Performance Models. In *Proceedings of the Second International Conference on Autonomic Computing (ICAC'05)*, pages 229–240, June 2005.

[17] Jeff Bezos. Amazon.com CEO Jeff Bezos on Animoto. Startup School 2008, April 2008. URL `http://blog.animoto.com/2008/04/21/amazon-ceo-jeff-bezos-on-animoto/`. `http://blog.animoto.com/2008/04/21/amazon-ceo-jeff-bezos-on-animoto/`.

[18] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, Sep/Oct 1999. ISSN 0890-8044. doi: 10.1109/65.793694.

[19] D. Breitgand, E.A. Henis, O. Shehory, and J.M. Lake. Derivation of response time service level objectives for business services. In *2nd IEEE/IFIP International Workshop on Business-Driven IT Management (BDIM '07)*, pages 29–38, May 2007. doi: 10.1109/BDIM.2007.375009.

[20] Eric A. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4):46–55, July-August 2001. ISSN 1089-7801. doi: 10.1109/4236.939450.

[21] Rajkumar Buyya, Toni Cortes, and Hai Jin. Single System Image. *International Journal of High Performance Computing Applications (IJHPCA)*, 15(2):124–135, Summer 2001.

[22] Rajkumar Buyya, Chee S. Yeo, and Srikumar Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications (HPCC '08)*, pages 5–13. IEEE Computer Society, 2008. ISBN 978-0-7695-3352-0. doi: http://dx.doi.org/10.1109/HPCC.2008.172. Keynote paper.

[23] Bogdan Caprita, Wong Chun Chan, Jason Nieh, Clifford Stein, and Haoqiang Zheng. Group ratio round-robin: O(1) proportional share scheduling for uniprocessor and multiprocessor systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05)*, pages 36–36, Berkeley, CA, USA, 2005. USENIX Association.

[24] Valeria Cardellini, Michele Colajanni, and Philip S. Yu. Dynamic Load balancing on Web-Server Systems. *IEEE Internet Computing*, 3(3):28–39, May/June 1999. ISSN 1089-7801. doi: 10.1109/4236.769420.

[25] Valeria Cardellini, Emiliano Casalicchio, Michele Colajanni, and Philip S. Yu. The state of the art in locally distributed web-server systems. *ACM Computing Surveys*, 34(2):263–311, June 2002. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/508352.508355.

[26] Jakob Carlström and Raphael Rom. Application-aware admission control and scheduling in web servers. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 2, pages 506–515, 2002. doi: 10.1109/INFCOM.2002.1019295.

[27] Thomas L. Casavant and Jon G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988. ISSN 0098-5589. doi: 10.1109/32.4634.

[28] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurements. In *Proceedings of the Eleventh IEEE/ACM International Workshop on Quality of Service (IWQoS 2003)*, pages 381–400, June 2003.

[29] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. *ACM SIGOPS Operating Systems Review*, 35(5):103–116, 2001. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/502059.502045.

[30] Huamin Chen and P. Mohapatra. Session-based overload control in qos-aware web servers. In *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 2, pages 516–524. IEEE Computer Society, June 2002. doi: 10.1109/INFCOM.2002.1019296.

[31] Yiyu Chen, Amitayu Das, Natarajan Gautam, Qian Wang, and Anand Sivasubramaniam. Pricing-based strategies for autonomic control of web servers for time-varying request arrivals. *Engineering Applications of Artificial Intelligence*, 17(7):841 – 854, 2004. ISSN 0952-1976. doi: DOI:10.1016/j.engappai.2004.09.001. Special issue on Autonomic Computing Systems.

[32] Ludmila Cherkasova and Peter Phaal. Session-Based Admission Control: A Mechanism for Peak Load Management of Commercial Web Sites. *IEEE Transactions on Computers*, 51(6):669–685, 2002. ISSN 0018-9340. doi: http://dx.doi.org/10.1109/TC.2002.1009151.

[33] Adrian Cockcroft and Bill Walker. *Capacity Planning for Internet Services*. Prentice Hall Professional Technical Reference, 2001. ISBN 0130894028.

[34] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[35] Andrea D'Ambrogio. A Model-driven WSDL Extension for Describing the QoS of Web Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS '06)*, pages 789–796, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2669-1. doi: http://dx.doi.org/10.1109/ICWS.2006.10.

[36] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin 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 (USITS'03)*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[37] Ronald P. Doyle, Jeffrey S. Chase, Omer M. Asad, Wei Jin, and Amin M. Vahdat. Model-based resource provisioning in a web service utility. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[38] T. Eilam, K. Appleby, J. Breh, G. Breiter, H. Daur, S. A. Fakhouri, G. D. H. Hunt, T. Lu, S. D. Miller, L. B. Mummert, J. A. Pershing, and H. Wagner. Using a utility computing framework to develop utility systems. *IBM System Journal*, 43(1):97–120, 2004. ISSN 0018-8670.

[39] Sameh Elnikety, Erich Nahum, John Tracey, and Willy Zwaenepoel. A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings of the 13th international conference on World Wide Web (WWW '04)*, pages 276–286, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X. doi: http://doi.acm.org/10.1145/988672.988710.

[40] Daniel Friedman. The double auction market institutions: a survey. In *Proceedings of the workshop*

on double auction markets, pages 3–25. Santa Fe Institute Studies in the Science of Complexity, New Mexico, Perseus Publishing, 1991.

[41] S. Ghosh, R. Rajkumar, J. Hansen, and J. Lehoczky. Scalable resource allocation for multi-processor qos optimization. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 174–183, May 2003. doi: 10.1109/ICDCS.2003.1203464.

[42] Google. Google Search Statistics from 9/11/01, 2004. URL `http://www.google.com/press/zeitgeist/9-11-search.html`. `http://www.google.com/press/zeitgeist/9-11-search.html`.

[43] Google. Google SOAP Search API, 2006. URL `http://code.google.com/apis/soapsearch/index.html`.

[44] GPUGRID.net, 2009. URL `http://www.gpugrid.net/`.

[45] I. S. Gradshteyn and I. M. Ryzhik. *Table of Integrals, Series, and Products*. Academic Press, 1980.

[46] S. Graupner, V. Kotov, and H. Trinks. Resource-sharing and service deployment in virtual data centers. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW'02)*, pages 666–671. IEEE Computer Society, 2002. doi: 10.1109/ICDCSW.2002.1030845.

[47] GridWorks. PBS GridWorks – Enabling On-Demand Computing, 2009. URL `http://www.pbsgridworks.com`.

[48] Jordi Guitart, David Carrera, Vicenç Beltran, Jordi Torres, and Eduard Ayguadé. Designing an overload control strategy for secure e-commerce applications. *Computer Networks*, 51(15):4492 – 4510, 2007. ISSN 1389-1286. doi: DOI:10.1016/j.comnet.2007.05.010.

[49] J.P. Hansen, Sourav Ghosh, Ragunathan Rajkumar, and J. Lehoczky. Resource management of highly configurable tasks. In *Proceedings. 18th International Parallel and Distributed Processing Symposium (IPDPS'04S)*, pages 116–, April 2004. doi: 10.1109/IPDPS.2004.1303070.

[50] Mor Harchol-Balter, Mark Crovella, and Cristina D. Murta. On Choosing a Task Assignment Policy for a Distributed Server System. In *Proceedings of the 10th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools (TOOLS '98)*, pages 231–242, London, UK, 1998. Springer-Verlag. ISBN 3-540-64949-2.

[51] *HP WebQoS Administration Guide*. Hewlett-Packard, 2001. URL `http://docs.hp.com/en/1563/WebQoSAdmin.pdf`.

[52] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[53] Bernardo A. Huberman, Fang Wu, and Li Zhang. Ensuring trust in one time exchanges: solving the qos problem. *Netnomics*, 7(1):27–37, 2005. ISSN 1385-9587. doi: http://dx.doi.org/10.1007/s11066-006-9002-2.

[54] IBM. Workload Management with LoadLeveler. IBM Redbooks, November 2001. URL `http://www.redbooks.ibm.com/redbooks/pdfs/sg246038.pdf`.

[55] IBM. Provisioning On Demand Introducing IBM Tivoli Intelligent ThinkDynamic Orchestrator. IBM Redbooks, December 2003. URL `http://www.redbooks.ibm.com/abstracts/sg248888.html`.

[56] IETF. Hypertext Transfer Protocol – HTTP/1.1, June 1999. URL `http://www.ietf.org/rfc/rfc2616.txt`.

[57] Deepal Jayasinghe. SOA Development with Axis2, Part 1: Understanding Axis2 Basis. IBM developerWorks, August 2006. `http://www-128.ibm.com/developerworks/webservices/library/ws-soa-axis2-1/`.

[58] Deepal Jayasinghe. *Quickstart Apache Axis2 - A practical guide to creating quality web services*, chapter Handler and Phase in Apache Axis2. Packt Publishing, May 2008. `http://www.packtpub.com/article/handler-and-phase-in-apache-axis`.

[59] Laxmikant V. Kalé, Sameer Kumar, Mani Potnuru, Jayant DeSouza, and Sindhura Bandhakavi. Faucets: Efficient resource allocation on the computational grid. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP '04)*, pages 396–405, Washington, DC, USA, August 2004. IEEE Computer Society. ISBN 0-7695-2197-5. doi: http://dx.doi.org/10.1109/ICPP.2004.35.

[60] Kyoung-Don Kang, Sang H. Son, and John A. Stankovic. Differentiated Real-Time Data Services for E-Commerce Applications. *Electronic Commerce Research*, 3(1–2):113–142, 2003.

[61] Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory System Behavior of Java-Based Middleware. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA '03)*, page 217, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1871-0.

[62] K. Keahey, I. Foster, T. Freeman, and X. Zhang. Virtual Workspaces: Achieving Quality of Service and Quality of Life in the Grid. *Scientific Programming*, 13(4):265–275, 2005. ISSN 1058-9244.

[63] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36 (1):41–50, Jan 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1160055.

[64] Jeffrey O. Kephart and Rajarshi Das. Achieving Self-Management via Utility Functions. *IEEE Internet Computing*, 11(1):40–48, Jan.-Feb. 2007. ISSN 1089-7801. doi: 10.1109/MIC.2007.2.

[65] William LeFebvre. Cnn.com: Facing a world crisis. Invited talk at USENIX LISA'01, December 2001. URL `http://www.tcsa.org/lisa2001/cnn.txt`. `http://www.tcsa.org/lisa2001/cnn.txt`.

[66] R. Levy, J. Nagarajarao, G. Pacifici, A. Spreitzer, A. Tantawi, and A. Youssef. Performance Management for Cluster Based Web Services. In *Prooceedings IFIP/IEEE Eighth International Symposium on Integrated Network Management*, pages 247–261, March 2003.

[67] Ying Li, Kewei Sun, Jie Qiu, and Ying Chen. Self-reconfiguration of service-based systems: a case study for service level agreements and resource optimization. In *Proceedings of the 2005 IEEE International Conference on Web Services (ICWS 2005)*, volume 1, pages 266–273, July 2005. doi: 10.1109/ICWS.2005.103.

[68] Greg Linden. *Make Your Data Useful*. Amazon, November 2006. URL `http://home.blarg.net/~glinden/StanfordDataMining.2006-11-29.ppt`.

[69] Zhen Liu, Mark S. Squillante, and Joel L. Wolf. On Maximizing Service-Level-Agreement Profits. In *Proceedings of the 3rd ACM conference on Electronic Commerce (EC'01)*, pages 213–223, New York, NY, USA, 2001. ACM. ISBN 1-58113-387-1. doi: http://doi.acm.org/10.1145/501158.501185.

[70] Chenyang Lu, T.F. Abdelzaber, J.A. Stankovic, and S.H. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. In *Proceedings of the Seventh IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, pages 51–62, 2001. doi: 10.1109/RTTAS.2001.929865.

[71] Sai Rajesh Mahabhashyam and Natarajan Gautam. Dynamic resource allocation of shared data centers supporting multiclass requests. In *Proceedings of the First International Conference on Autonomic Computing (ICAC '04)*, pages 222–229, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2114-2.

[72] Michele Mazzucco, Isi Mitrani, Jennie Palmer, Mike Fisher, and Paul McKee. Web Service Hosting and Revenue Maximization. In *Proocedings of the Fifth European Conference on Web Services (ECOWS'07)*, pages 45–54, November 2007.

[73] David T. McWherter, Bianca Schroeder, Anastassia Ailamaki, and Mor Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*, page 535, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2065-0.

[74] MEMSET. Memset hosting: Dedicated servers, 2006. URL http://www.memset.com/dedicated_servers.php. http://www.memset.com/dedicated_servers.php.

[75] Daniel A. Menascé. Mapping Service-Level Agreements in Distributed Applications. *IEEE Internet Computing*, 8(5):100–102, September-October 2004. ISSN 1089-7801. doi: 10.1109/MIC.2004.47.

[76] Daniel A. Menascé, Virgilio A. F. Almeida, Rodrigo Fonseca, and Marco A. Mendes. Business-oriented resource management policies for e-commerce servers. *Performance Evaluation*, 42(2-3): 223–239, October 2000. ISSN 0166-5316. doi: http://dx.doi.org/10.1016/S0166-5316(00)00034-1.

[77] Daniel A. Menascé, Daniel Barbará, and Ronald Dodge. Preserving QoS of E-Commerce Sites Through Self-Tuning: A Performance Model Approach. In *Proceedings of the 3rd ACM conference on Electronic Commerce (EC '01)*, pages 224–234, New York, NY, USA, 2001. ACM. ISBN 1-58113-387-1. doi: http://doi.acm.org/10.1145/501158.501186.

[78] Isi Mitrani. *Probabilistic Modelling*. Cambridge University Press, 1998.

[79] Justin Moore, David Irwin, Laura Grit, Sara Sprenkle, and Jeff Chase. Managing Mixed-Use Clusters with Cluster-on-Demand. Technical report, Duke University, November 2002.

[80] OpenNebula.org. URL http://www.opennebula.org/.

[81] Jennie Palmer and Isi Mitrani. Optimal Server Allocation in Reconfigurable Clusters with Multiple Job Types. In *International Conference on Computational Science and its Applications (ICCSA 2004)*, Lecture Notes in Computer Science, 3044, pages 76–86. Springer, 2004.

[82] M. P. Papazoglou and D. Georgakopoulos. Introduction. *Communications of the ACM*, 46(10):24–28, 2003. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/944217.944233.

[83] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. Morgan Kaufmann, 3rd edition, 2003.

[84] Platform – The Power of Sharing. Platform LSF, 2009. URL http://www.platform.com/Products/platform-lsf.

[85] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 298–307, Dec 1997. doi: 10.1109/REAL.1997.641291.

[86] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. QoS-driven server migration for Internet data centers. In *Tenth IEEE International Workshop on Quality of Service (IWQoS 2002)*, pages 3–12. IEEE Computer Society, May 2002. doi: 10.1109/IWQoS.2002.1006569.

[87] Supranamaya Ranjan and Edward Knightly. High-Performance Resource Allocation and Request Redirection Algorithms for Web Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 19 (9):1186–1200, 2008. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2007.70810.

[88] Ori Regev and Noam Nisan. The POPCORN Market—An Online Market for Computational Resources. In *Proceedings of the first international conference on Information and computation economies (ICE '98)*, pages 148–157, New York, NY, USA, 1998. ACM. ISBN 1-58113-076-7. doi: http://doi.acm.org/10.1145/288994.289027.

[89] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, sixth edition, 1997.

[90] Mathias Sallé and Claudio Bartolini. Management by contract. In *Proocedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, volume 1, pages 787–800 Vol.1, April 2004. doi: 10.1109/NOMS.2004.1317766.

[91] SlashDot.org. What is the "Slashdot Effect"?, 2000. URL `http://slashdot.org/faq/slashmeta.shtml`. `http://slashdot.org/faq/slashmeta.shtml`.

[92] Dennis Sosnoski. Java programming dynamics, Part 2: Introducing reflection. IBM developerWorks, June 2003. `http://www.ibm.com/developerworks/library/j-dyn0603/`.

[93] Christopher Stewart and Kai Shen. Performance modeling and system management for multi-component online services. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 71–84, Berkeley, CA, USA, 2005. USENIX Association.

[94] Christopher Stewart, Terence Kelly, and Alex Zhang. Exploiting nonstationarity for performance prediction. *ACM SIGOPS Operating Systems Review*, 41(3):31–44, 2007. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1272998.1273002.

[95] Sun Microsystems. Java Management Extensions (JMX), 2009. URL `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/`. `http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/`.

[96] Sun Microsystems. Sun Grid Engine v. 6.2u1, December 2008. URL `http://www.sun.com/software/sge/`.

[97] Sun Microsystems, Inc. N1 Grid Technology – Just In Time Computing. White Paper, 2003. URL `www.sun.com/software/solutions/n1/wp-n1.pdf`.

[98] *Java Message Service Specification - Version 1.1*. Sun Microystems, March 2002. `http://java.sun.com/products/jms/docs.html`.

[99] The Apache Software Foundation. Apache Axis2 – Web Services Engine, August 2008. URL http://ws.apache.org/axis2/. http://ws.apache.org/axis2/.

[100] The Apache Software Foundation. Apache Axiom – The XML Object Model, September 2008. URL http://ws.apache.org/commons/axiom/. http://ws.apache.org/commons/axiom/.

[101] The Apache Software Foundation. Apache Tomcat, 2006. URL http://tomcat.apache.org/. http://tomcat.apache.org/.

[102] University of California. SETI@home v. 6.06, 2009. URL http://setiathome.ssl.berkeley.edu/.

[103] University of Winsconsin-Madison. Condor – High Throughput Computing, December 2008. URL http://www.cs.wisc.edu/condor/.

[104] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. *SIGOPS Operating Systems Review*, 36(Special Issue: Cluster Resource Management):239–254, 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844151.

[105] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An Analytical Model for Multitier Internet Services and Its Applications. *ACM SIGMETRICS Performance Evaluation Review*, 33(1):291–302, 2005. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1071690.1064252.

[106] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2009. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/1496091.1496100.

[107] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37, 1961. URL http://www.jstor.org/stable/2977633.

[108] N. Ya. Vilenkin. *Combinatorics*. Academic Press, 1971.

[109] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. In *Twelfth IEEE International Workshop on Quality of Service (IWQOS 2004)*, pages 57–66, June 2004. doi: 10.1109/IWQOS.2004.1309357.

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

[111] W3C. Web Services Description Language (WSDL) 1.1, March 2001. http://www.w3.org/TR/wsdl.

[112] W3C. Web Services Addressing (WS-Addressing), August 2004. http://www.w3.org/Submission/ws-addressing/.

[113] W3C. SOAP Message Transmission Optimization Mechanism, January 2005. URL http://www.w3.org/TR/soap12-mtom/. http://www.w3.org/TR/soap12-mtom/.

[114] W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition), April 2007. URL http://www.w3.org/TR/soap12-part1/.

[115] W3C. Web Services Description Language (WSDL) Version 2.0: Additional MEPs, June 2007. http://www.w3.org/TR/wsdl20-additional-meps/.

[116] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility Functions in Autonomic Systems. In *Proceedings of the International Conference on Autonomic Computing (ICAC'04)*, pages 70–77, May 2004. doi: 10.1109/ICAC.2004.1301349.

[117] Xia Wang, Tomas Vitvar, Mick Kerrigan, and Ioan Toma. A QoS-Aware Selection Model for Semantic Web Services. *Service-Oriented Computing ,Äì ICSOC 2006*, pages 390–401, 2006. doi: http://dx.doi.org/10.1007/11948148_32.

[118] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *SIGOPS Operating Systems Review*, 35(5):230–243, December 2001. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/502059.502057.

[119] Ward Whitt. Approximations for the GI/G/m Queue. *Production and Operations Management*, 2(2): 114–161, 1993.

[120] Edward Wustenhoff. *Service Level Agreement in the Data Center*. Sun Microsystems, April 2002. URL http://www.sun.com/blueprints/0402/sla.pdf. http://www.sun.com/blueprints/0402/sla.pdf.

[121] Cheng-Zhong Xu, Bojin Liu, and Jianbin Wei. Model Predictive Feedback Control for QoS Assurance in Webservers. *IEEE Computer*, 41(3):66–72, March 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.93.

[122] Chee Shin Yeo and R. Buyya. Integrated risk analysis for a commercial computing service. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2007)*, pages 1–10, March 2007. doi: 10.1109/IPDPS.2007.370241.

[123] Chee Shin Yeo and Rajkumar Buyya. Service Level Agreement based Allocation of Cluster Resources: Handling Penalty to Enhance Utility. In *Proceedings of the 7th IEEE International Conference on Cluster Computing (Cluster 2005)*, Boston, MA, USA, 2005. IEEE Computer Society: Los Alamitos, CA, USA. ISBN 0-7803-9486-0. doi: 10.1109/CLUSTR.2005.347075.

[124] Li Zhang and Danilo Ardagna. SLA Based Profit Optimization in Autonomic Computing Systems. In *Proceedings of the 2nd international conference on Service oriented computing (ICSOC '04)*, pages 173–182, New York, NY, USA, 2004. ACM. ISBN 1-58113-871-7. doi: http://doi.acm.org/10.1145/1035167.1035193.

[125] Jingyu Zhou and Tao Yang. Selective Early Request Termination for Busy Internet Services. In *Proceedings of the 15th international conference on World Wide Web (WWW '06)*, pages 605–614, New York, NY, USA, 2006. ACM. ISBN 1-59593-323-9. doi: http://doi.acm.org/10.1145/1135777.1135866.