

RICE UNIVERSITY

**Nested QoS: Providing Flexible SLAs in Shared
Storage Systems**

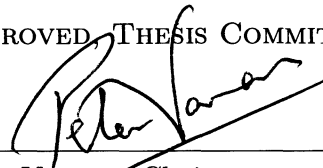
by

Hui Wang

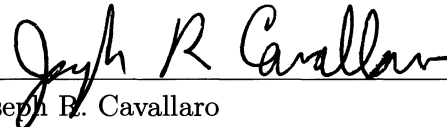
A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

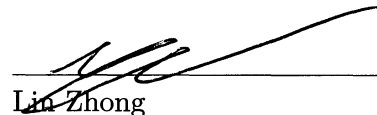
APPROVED THESIS COMMITTEE:



Peter Varman, Chair
Professor of Electrical and Computer
Engineering and Computer Science



Joseph B. Cavallaro
Professor of Electrical and Computer
Engineering and Computer Science



Lin Zhong
Associate Professor of Electrical and
Computer Engineering and Computer
Science

Houston, Texas

May, 2012

ABSTRACT

Nested QoS: Providing Flexible SLAs in Shared Storage Systems

by

Hui Wang

The increasing popularity of storage and server consolidation introduces new challenges for resource management, capacity provisioning, and application performance guaranteeing. In addition, the bursty nature of storage workloads results in a large gap between the peak and the average capacity required to meet response time bounds, leading to low overall server utilization and high cost. This situation is driving the development of elastic QoS models that allow clients greater flexibility in adopting SLAs tailored to their workload characteristics and performance requirements, while allowing the service provider opportunities to optimize provisioning and scheduling decisions.

This thesis presents a novel service model, called the Nested QoS model, for multiplexing concurrent bursty workloads in shared storage systems. The solution employs two strategies together: systematically classifying requests with a graduated QoS and flexibly scheduling the classified portions. The results show that the Nested QoS model provides (i) performance isolation and strong performance guarantees for both well-behaved and misbehaving workloads; (2) a flexible and auditable elastic SLA definition; and (3) improved server utilization.

Acknowledgements

First and foremost, I would like to thank my advisor Professor Peter Varman for his great help during my research and thesis work. He provided valuable feedback and insightful suggestions to my research topics. He was always there to listen and to discuss all the research problems with me. I really appreciate all the time and effort he spent on revising my papers and thesis. The thesis and all my research papers couldn't have been finished without his help.

I would like to thank Professor Joseph Cavallaro and Professor Lin Zhong for serving on my thesis committee and giving me valuable feedbacks. I would also like to thank all my friends at Rice.

Last but not least, I would like to thank my parents for their unconditional love and support, and thank Guohui for all the help and encouragement during my study at Rice.

Research support from the National Science Foundation (NSF) is gratefully acknowledged.

Contents

Abstract	ii
List of Illustrations	iii
List of Tables	v
1 Introduction	1
1.1 Overview	1
1.2 Problems Addressed in The Thesis	4
1.3 Contributions	9
1.4 Thesis Organization	9
2 Background and Related Work	11
2.1 Storage System models	11
2.1.1 Direct Attached Storage (DAS)	11
2.1.2 Network Attached Storage (NAS)	13
2.1.3 Storage Area Network (SAN)	14
2.1.4 Summary	15
2.2 Related Work	16
2.2.1 Proportional Sharing Model	16
2.2.2 Service Curves Model	17
2.2.3 Workload Shaping Model	18
2.3 Summary	20
3 Nested QoS Model	21
3.1 Nested QoS Model	21
3.1.1 Token Bucket Traffic Envelope	22

3.1.2	Nested QoS Model and SLAs Specification	24
3.1.3	Model Analysis	27
3.2	Scheduling Framework Based on the Nested QoS Model	32
3.2.1	Scheduling Algorithms	34
3.2.2	Capacity Planning	39
3.2.3	Workload Parameters	43
4	Performance Evaluation	44
4.1	Experimental Setup	44
4.2	Workload Isolation	50
4.3	Bad-Region Isolation	52
4.4	Reduced Capacity Provisioning	55
4.5	Linux Implementations	59
4.5.1	Single Workload	59
4.5.2	Multiple Workloads	60
4.6	Summary	65
5	Conclusions and Future Work	66
5.1	Conclusions	66
5.2	Future Work	67
	Bibliography	68

Illustrations

1.1	A General Sharing Environment	3
2.1	Direct Attached Storage Architecture	12
2.2	Network Attached Storage Architecture	13
2.3	Storage Area Network Architecture	14
3.1	Upper Bound on Arrivals imposed by the Token Bucket	24
3.2	Nested QoS Model	25
3.3	An Example for Workload Classification	26
3.4	Effect of Bad Requests on Later Requests. (a) Three bad requests arrive at p. (b) Bad requests delayed to conform to SLAs. (c) Later requests delayed by bad requests at p. (d) Later requests meet deadline if requests at p are removed	28
3.5	Scheduler Framework	33
3.6	Request Classifier with Token Bucket	33
4.1	Arrival pattern for (a) Both W1 and W2 are well-behaved. (b) W1 violates SLAs and sends more requests during time 600s-700s	45
4.2	W1 and W2 are well-behaved workloads. Response time distribution and CDF of W1 and W2 with three scheduling methods: Nested QoS, pClock, WF2Q.	46

4.3	The bandwidth allocation for well-behaved W1 and W2 by three methods: Nested QoS, <i>pClock</i> , WF2Q.	47
4.4	Response time distribution and CDF of W1(violation) and W2 with three scheduling methods: Nested QoS, <i>pClock</i> , WF2Q.	48
4.5	The bandwidth allocation for W1 (violation) and W2 by three methods: Nested QoS, <i>pClock</i> , WF2Q.	49
4.6	Response time distribution for W1 and W1 (violation) with three scheduling methods: Nested QoS, <i>pClock</i> , WF2Q.	53
4.7	W1 violates its SLA and sends more requests from 600s to 700s. Nested QoS isolates the bad region and still guarantee the well-behaved part. However <i>pClock</i> delays all of W1's requests from 600s all the way up to 790s.	54
4.8	Reduced capacity requirements for different deadlines using Nested QoS	56
4.9	Reduced capacity requirements using Nested QoS	57
4.10	Arrival rate of <i>prxy</i> workload	60
4.11	Request response time distribution and CDF for <i>prxy</i> workload using Nested QoS and <i>pClock</i>	61
4.12	Arrival rate of <i>proj</i> workload	62
4.13	Response time distribution for multiple workloads	63
4.14	Response time CDF for multiple workloads	64

Tables

1.1	Comparison of Scheduling Algorithms	8
3.1	Symbols	35

Chapter 1

Introduction

1.1 Overview

Server and storage consolidation are becoming increasingly common in data centers, due to the economies of shared infrastructure, the benefits of centralized management, high reliability, and lower operating cost. However, the increased resource consolidation also introduces new challenges for performance isolation, resource provisioning, capacity planning, and meeting client's performance Quality of Service (QoS) expectations. This situation is even more serious for storage systems, because of the dependence of service time on access characteristics, and the burstiness of many storage workloads.

There are several large data centers established by large companies like IBM [1], Google [2] and Amazon [3]. These data centers provide compute and storage services for individual and enterprise applications. In this thesis we only focus on storage resource and services. In these data centers the clients purchase storage space to store and retrieve their data, while the service providers provision and manage the underlying physical resources in order to meet the client's accessibility and reliability requirements. These requirements are typically defined by Service-level Agreements

(SLAs) between the client and the service provider.

An SLA is a negotiated agreement between a service provider and a customer. It is usually defined in measurable terms so that services received by the customers can be monitored and compared against the QoS specified by the SLA [4]. SLAs define a number of Service Level Objectives (SLOs) like bandwidth, latency, uptime, etc. In storage and server consolidation environments, well-defined SLAs and QoS models should specify the following items: first, the SLAs should define the workload properties. It is not possible for the service provider to guarantee specific performance without a description of the workload; second, it should be possible to verify the charges for services, and to demonstrate SLA violations on the part of either the client or the provider. Typical performance SLAs are usually described in terms of minimum throughput guarantees [5, 6] (IOPS) or response time bounds [7, 8] for rate-controlled clients. Figure 1.1 shows a general framework for sharing a storage server among multiple clients. The requests of multiple clients are routed to the shared storage server which must schedule the competing requests to meet performance QoS defined by the SLAs.

There are several challenges to provide QoS in a shared storage server. First, since the storage system is shared by multiple concurrent workloads, the performance of well-behaved clients should not be hurt by the behavior of other clients that may try to monopolize the server. In other words, the service provider should provide performance isolation for different clients based on their workload and QoS requirements.

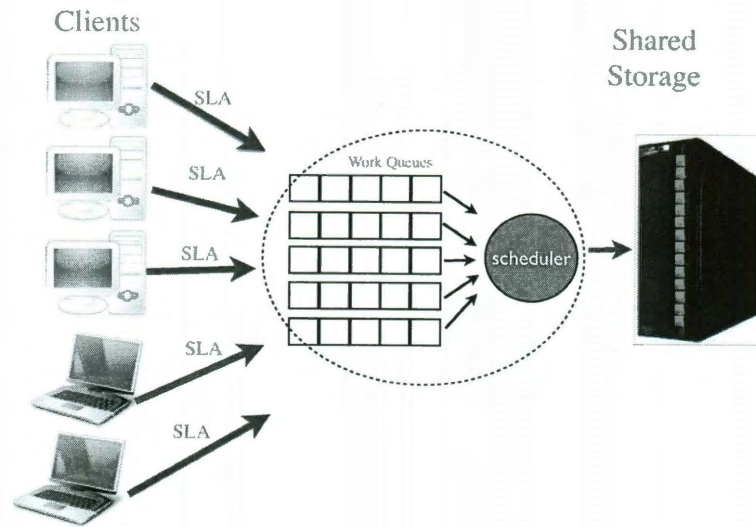


Figure 1.1 : A General Sharing Environment

Second, storage workloads tend to be very bursty [9, 10, 11]: *i.e.*, the instantaneous arrival rate during some time intervals is significantly higher than the long-term request arrival rate. The peak capacity required to handle short-duration bursts of requests, may be an order of magnitude or more than the long-term average requirement. Providing capacity based on the peak rate can result in significant over-provisioning, increasing capital expenditures and power and cooling infrastructure for servers. The server is significantly underutilized during the non-bursty periods, resulting in low server efficiencies. This has been confirmed by measurements in actual data centers [12]. Furthermore, fine-grained run-time capacity management is only partially successful since the workload is unpredictable and can change drastically over small time intervals. For storage devices which incur appreciable latency in transitioning

between states and where idle power consumption is significant, the operational and energy costs of over provisioning can be very high. Accurate provisioning is complicated by the bursty nature of storage workloads [9, 13] and sharing by multiple clients. Third, mechanisms to handle excess traffic from clients that exceed their SLAs limits need to be handled carefully by the storage server. Unlike the case of communication protocols in networking systems, dropping requests when the system is oversubscribed is not a viable option for storage systems, since storage IO protocols do not generally support automatic retransmission mechanisms.

The situation described above is driving the development of elastic QoS models that allow clients greater flexibility in choosing SLAs tailored to their workload characteristics and performance requirements, while allowing the service provider to optimize provisioning and scheduling decisions.

1.2 Problems Addressed in The Thesis

We use three examples to illustrate the problems we are trying to address in this thesis.

Example I Consider in a shared system with two clients C_A and C_B . The SLAs of both clients require a throughput of 100 IOPS (IOs Per Second) and worst-case latency bound of 100 ms. C_A sends requests at a uniform rate of 100 IOPS, while C_B sends 50 requests in a burst, every 500 ms. If each client had its own server with 100 IOPS capacity, then all the requests of C_A will have a 10 ms response time, while

the worst-case latency of request of C_B is 500 ms. If both workloads are consolidated on a single server of capacity 200 IOPS, the execution profile of both workloads can change drastically. If requests are served in the order of arrival, then the requests of C_A will be delayed by the requests of C_B , and can face latencies of up to 250 ms. A good QoS model should *isolate the performance of contending workloads*, and prevent them from affecting each other's performance in a negative way.

Example II Consider client C_A of Example I which sends requests at a uniform rate of 100 IOPS and has a latency requirement of 10ms. The workload can expect a 10ms response time with a server of capacity 100 IOPS. Now suppose C_A slightly alters its pattern by bursting in the first 100 ms of each 1 second interval. Specifically, suppose it sends 6 requests at the start of the interval followed by 4 requests every 20ms apart, before reverting to its stipulated arrival rate of 1 request every 10 ms for the remainder of the interval. This local violation (violation at the beginning of each second) will delay not only the first 5 of the 6 requests in the burst at the start of the interval but also the following 4 well-behaved requests in the 100 ms interval. That is, the number of requests that miss the deadline is almost double the number that violated the arrival SLAs. A good QoS model should *localize the temporary violation and prevent it from affecting the well-behaved part of the workload*.

Example III How much capacity is required to make sure that all the good requests of the workload in Example II, meet their 10 ms deadline? The 5 requests at the start of the interval and the requests arriving 20 ms later should all finish within

30 ms, which implies a capacity of $7 \text{ IOs} / 30 \text{ ms} = 233 \text{ IOPS}$. Thus the required capacity to meet the response time has doubled because of the presence of a few bad requests. A good QoS model should *provide guarantees for the well-behaved part of a workload with minimal capacity*.

The three examples just described motivate us to define three principles (P1 through P3) that QoS schedulers in storage consolidation environments should satisfy.

- **P1: Inter-client Isolation** (Do not harm others) – This is the fundamental requirement of *performance isolation* which requires that clients be insulated from the behavior of other concurrently executing clients sharing the resource. Specifically, the system should encapsulate a client so that any bad behavior on its part is not allowed to adversely affect other well-behaved clients. If a client exceeds its stipulated SLA and sends more requests in a time interval than allowed by its contractual agreement, it should not be allowed to garner additional service at the expense of other well-behaved clients.
- **P2: Intra-client Isolation** (Do not harm oneself) – This is simply the performance isolation requirement applied to a single client. It asserts that the well-behaved and ill-behaved portions of a workload should also be isolated from each other, so that the effects of the bad behavior are temporally localized. In other words, if the client exceeds its SLAs during some time interval, it may be penalized during this period but once its behavior becomes compliant

it should start receiving its SLAs guarantees again.

- **P3: Capacity Requirements** – The service provider should provide a rich set of SLAs specifications with different cost and performance QoS guarantees to satisfy a diverse set of client needs. Scheduling algorithms should minimize the capacity required to meet the set of SLAs and support accurate estimation of capacity requirements. Finally, it should be possible to audit the workload and observed performance, to verify if both parties had met their SLAs requirements in case of dispute.

In this thesis, we present a novel service model, called Nested QoS, which can fulfil all the three properties P1 to P3. The model permits flexible SLAs for the clients sharing a server, with significantly smaller capacity provisioning requirements than previous approaches [14, 7]. The Nested QoS model formalizes the observation that a disproportionate fraction of server capacity is used to handle the small tail of highly bursty requests. It classifies the workload with several traffic envelopes defined by Token Buckets [15], which are described in detail in Chapter 3. By choosing different traffic envelopes with different performance guarantees and capacity requirements, clients can tailor the SLAs to their workloads in accordance with their expectations and budgets. We also propose a scheduling framework for efficiently and flexibly sharing a server among multiple concurrent clients. Our approach combines two orthogonal techniques to significantly reduce the capacity requirements of the server: (i) use of the Nested QoS model to classify workloads and (ii) scheduling algorithms

Algorithms	Proportional Share	pClock	This thesis
Proportional bandwidth allocation	Yes	Yes	Yes
Fairness	Yes (Fine grained)	Yes (Coarse grained)	Yes (Coarse grained)
Latency Control	No	Yes	Yes
Workload Isolation (P1)	Yes	Yes	Yes
Bad-Region Isolation (P2)	No	No	Yes
Capacity Saving (P3)	NA	No	Yes

Table 1.1 : Comparison of Scheduling Algorithms

that multiplex server capacity between the fragments of the classified workloads to meet the SLAs.

The results in this thesis are compared against two well-known schedulers, *Proportional Share* (PS) and *pClock* as summarized in Table 1.1. PS allocates clients weighted bandwidth at a fine granularity so that the difference between the alloca-

tions of any two backlogged clients is always bounded by 1. In contrast, *pClock* and our scheduler are designed to handle short-term bursts of requests by allowing temporary unfairness in the bandwidth allocations. These two schedulers therefore provide latency control for workloads independent of their throughput requirements, unlike PS where response time and throughput are coupled. The issues related to workload isolation, bad-region isolation, and capacity requirements are discussed in Chapter 3.

1.3 Contributions

This thesis proposes a model called Nested QoS for providing performance SLAs in a shared server environment. The advantages of Nested QoS are:

- (i) Allows defining flexible SLAs based on workload characteristics and client's willingness to pay.
- (ii) Allows service provider to increase degree of consolidation and lower costs.
- (ii) Intuitive model and easy to implement. Permit audit and arbitration in case of dispute.
- (iii) Provides simple capacity estimation based on the Nested QoS parameters.

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 introduces storage systems models and current QoS models in storage systems. The limitations of those models are also discussed. In Chapter 3 we start by introducing the token bucket model,

with which we characterize the workload. Section 3.1.2 describes our Nested QoS model based on the token bucket model. In Section 3.1.3 we analyze how the Nested QoS model meets the three properties defined above. The scheduling framework and algorithm are also discussed in Chapter 3.2. Chapter 4 presents the performance evaluation. Finally, we conclude the thesis in Chapter 5 .

Chapter 2

Background and Related Work

This chapter describes the background and the related work. In Section 2.1 we first introduce three popular storage systems models. The challenges arising in shared storage environments are also discussed. In section 2.2 we present three QoS models that have been used in storage systems, and discuss the strengths and limitations of those models.

2.1 Storage System models

The rapid growth of IO-related applications like web search engines, web services, and online video, lead to the growing requirements of high reliability, usability, efficiency, and simpler management. In the past decades, several storage models have been developed to meet the requirements of data accessing for different applications.

2.1.1 Direct Attached Storage (DAS)

One of the most commonly used storage models is *Direct Attached Disk* (DAS), which is widely used in personal laptops, desktops and small organization servers. In a DAS system, one or more disk drives are directly connected to the server through SCSI or IDE interfaces [16]. Figure 2.1 show the architecture of a DAS system.

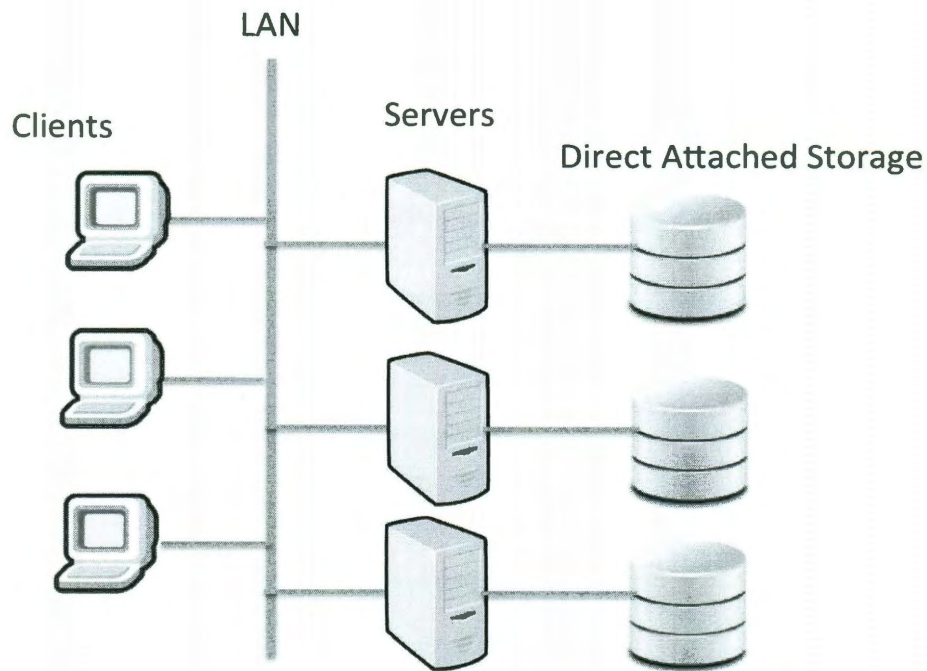


Figure 2.1 : Direct Attached Storage Architecture

DAS is suitable for accessing small amounts of local data from a server and has the benefit of cost-efficiency. However, DAS also has many limitations. One obvious limitation is reliability. As shown in Figure 2.1, a client can access the disk storage only through the connected server. A failure or crash of the server could cause the non-availability of the stored data. Another limitation is the difficulty of sharing free space. The free space on one disk cannot be shared by other clients, without additional middle-ware solutions or a clustered file system. Because of those limitations, other storage models have been developed. We will discuss those models in the following sections.

2.1.2 Network Attached Storage (NAS)

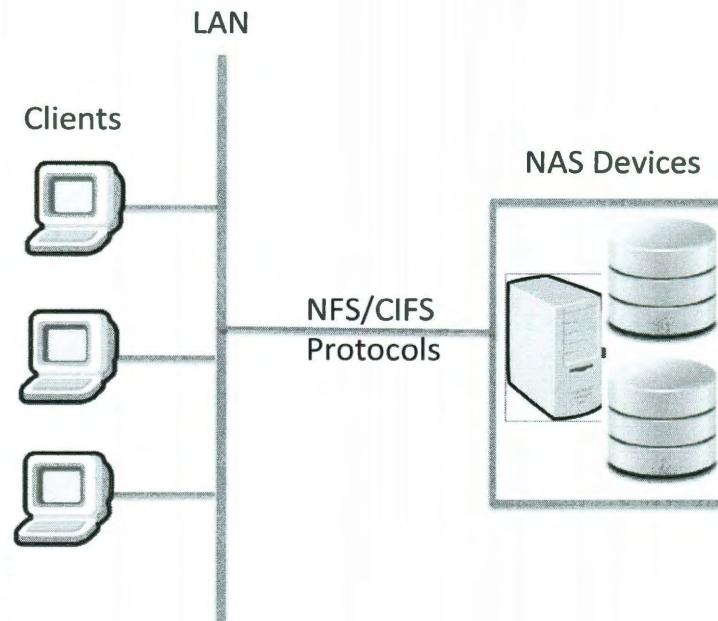


Figure 2.2 : Network Attached Storage Architecture

Network Attached Storage (NAS) is a file-level storage device that is set up with its own network address rather than being directly attached to a computer server (like DAS). It exports storage data at a file level and can be accessed over a computer network by heterogeneous clients. The clients access the NAS devices via a network file system protocols like NFS and CIFS [16, 17].

A number of factors are making NAS a very popular solution. First, NAS is a convenient method of sharing files among multiple computers and makes efficient use of data centre space. Second, it offers a convenient way for simple installation and management. As the volume grows, NAS systems can provide scalable solutions that

can be upgraded more easily and more cost-effective than DAS.

2.1.3 Storage Area Network (SAN)

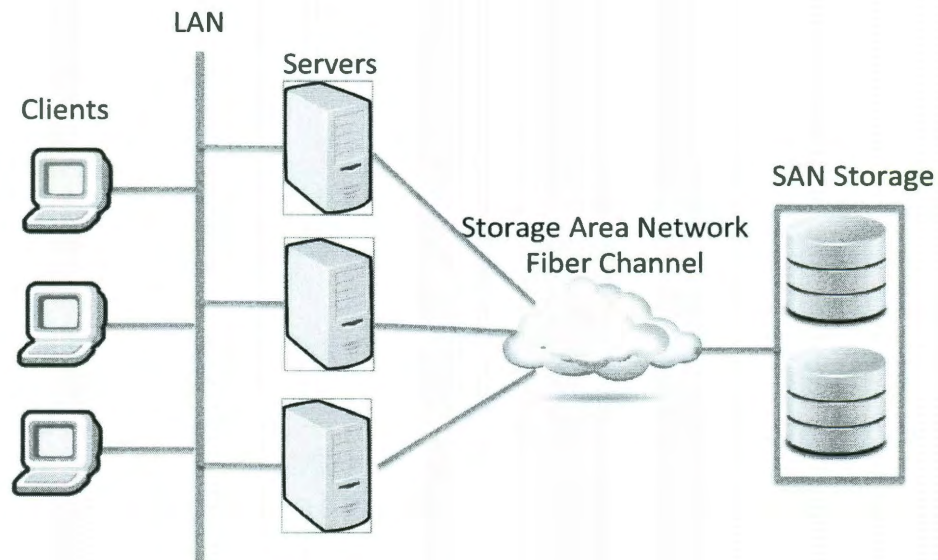


Figure 2.3 : Storage Area Network Architecture

A Storage Area Network (SAN) is a dedicated, high performance storage network that transfers data between front-end servers and storage devices, as shown in Figure 2.3. SAN is connected to the server with high speed Fibre Channel, which allows for simultaneous gigabyte communication between the different components in the network [16, 17]. Because of this high speed connection, it is ideal for moving large chunks of data across distances. Unlike NAS which exports file level data, SAN exports block level devices in a distributed environment. The distributed architecture of SANs offers higher levels of performance and availability than other storage mod-

els. Dynamic load balancing across the network enables SANs to provide fast data transfer while reducing I/O latency and server workload. In effect large numbers of users can be served by the SAN without creating bottlenecks on the LAN and the server network. Because of those properties, SANs is good for bandwidth intensive storage such as data-bases, transaction processing and video. However, the cost for high performance storage arrays connected to the SAN and Fiber Channel switches can be high.

2.1.4 Summary

Shared storage systems like NAS, SAN or clustered storage consisting of individual storage devices stitched together by middle-ware, face the following challenges:

- **Performance isolation**

The storage server should provide performance isolation for the clients sharing the system. Well-behaved clients should not encounter performance degradation because of the behavior of other clients. That is, the performance should be guaranteed independent of the behavior of other clients.

- **Capacity estimation**

The storage server should provide enough resources for the clients to meet their performance SLAs, while avoiding over-provisioning. Accurate capacity estimation to guarantee the performance of all the clients is a big challenge.

Several QoS service models have been proposed to address those problems. We will introduce these models in Section 2.2.

2.2 Related Work

2.2.1 Proportional Sharing Model

A large body of QoS-based resource allocation and scheduling work deals with the issue of proportional sharing (in terms of bandwidth allocation). The general idea is to emulate the behavior of an ideal (continuous) Generalized Processor Sharing (GPS) [18] scheduler in a discrete system, and divide the resource at a fine granularity in proportion to client weights. In the simplest proportional sharing model, each client i has a weight w_i , and the server allocates capacity in proportion to the weight w_i . In particular, suppose $A(t)$ is the set of active clients at time t , then client $i \in A(t)$ is allocated a bandwidth of $Cw_i / \sum_{j \in A(t)} w_j$, where C is the capacity of the server. A large number of algorithms have been proposed for proportional resource sharing *e.g.* Fair Queuing [19], WFQ [8, 20], WF2Q [14], Start Time Fair Queuing [5], SelfClocking [21], Leap Forward [22] etc.

In [23], Gulati *et al.* considered an enriched resource model that includes reservations and limits in addition to weights. Reservations define a lower bound on the allocation made to a client, while limits define upper bounds on their allocation. They also described a scheduler, mClock, that provides fine-grained proportional allocation subject to reservation and limit constraints.

To sum up, proportional sharing model focuses on the bandwidth allocation in proportion to specified weights, possibly modulated by lower and upper bounds. In these models, it is not possible to specify an independent response time requirement that is independent of its throughput. That is, those works did not explicitly address the problems of latency control. In addition, these models do not directly address the issue of capacity planning.

2.2.2 Service Curves Model

A second QoS model focuses on providing latency controls along with bandwidth allocation [8, 7] based on service curves [24, 25]. In addition to providing minimum bandwidth guarantees, individual requests are guaranteed a maximum response time provided the client traffic satisfies stipulated constraints on burst size and arrival rate. Cruz *et al.* [8, 20] utilize the service curves concept to regulate workload patterns and arrival rates. They provide the SCED [8] algorithm to schedule workloads specified by service curves. However, a major problem of the SCED algorithm is that it may result in starvation of a client which uses spare system capacity. Gulati *et al.* propose an algorithm *pClock* [7], which uses a token bucket to control the arrival burst size and flow rate, and provide a synchronization scheme to avoid starvation. In contrast to setting "earliest possible value" as a deadline, *pClock* sets the deadline of a request to be as late as possible. This allows *pClock* greater flexibility in scheduling spare capacity.

An important issue not addressed by these methods is the impact that a badly-behaved workload (one which violates its arrival constraints) has on its own performance and QoS guarantees. Since the existing methods do not isolate the non-compliant part of the workload from its well-behaved portions, even small violations can lead to loss of QoS guarantees over extended (unbounded) portions of the workload. In addition, only a single response time guarantee is supported by this model, so the flexibility is limited and the provisioned capacity requirements are high. We will discuss this problem in detail later in Chapter 3. Furthermore, the capacity required to guarantee the performance is based on worst-case behavior of all the workloads. If the total capacity is less than the worst-case requirement, the performance is not predictable. That is, how much a client's performance is degraded depends on local workload patterns rather than the QoS parameters.

2.2.3 Workload Shaping Model

The work more closely related to the Nested QoS are network QoS models where traffic shaping is used to decompose the workloads, and provide performance guarantees in terms of bandwidth and latency. There is a lot of work related to network QoS, like Network calculus [26], QoS in Packet Networks [27], Early Detection [28], and D-BIND [29] etc. Typically, arriving network traffic is made to conform to a token-bucket model by regulating the arrivals, and dropping requests that do not conform to the bucket parameters. With this drop-and-retransmission mechanism, the work-

load performance is guaranteed for the admitted portion of the workload, and the server utilization is maximized. However, drop-and-retransmission is not generally acceptable in storage systems, whose protocols do not support automatic retry.

The Nested QoS model classifies different portions of the workload into different classes and schedules them with different response time bounds. Empirical study of storage workloads to show the benefits of exempting a fraction of the workload from response time bounds was presented in [30], and used in the design of a slack-based two-level scheduler for a single client workload in [31]. However, there was no formal QoS model underlying the approach, that precluded specifying a well-defined SLA. The Nested QoS model provides a formal (but intuitive and enforceable) way to specify the notion of QoS. Furthermore, the model is easy to enforce, and mutually verifiable in case of dispute.

To conclude, the current QoS-based scheduling methods in storage systems do not support flexible latency controls. Their underlying SLAs are based on a single latency bound for the entire workload. As a consequence the server capacity requirements are high, and the server are highly under utilized. By using SLAs based on the Nested QoS, capacity requirements of a client are reduced significantly, since the extreme portions of the workload have relaxed latency requirements. Furthermore, the performance of a badly-behaving workload is still precisely defined by the SLAs. In the previous models, even temporary SLAs violations by the client could result in non-predictable performance for the offending workload. The details will be discussed

in the next chapter.

2.3 Summary

In this chapter, we described various storage models commonly used in both desktop and enterprise-class systems. Strengths and limitations are also discussed for each model. We summarized the challenges arising in shared storage systems in Section 2.1. In Section 2.2 we presented three existing QoS models and discussed their limitations.

In this thesis, we address these challenges arising in shared storage systems, by providing a novel Nested QoS service model.

Chapter 3

Nested QoS Model

In this chapter we will introduce the Nested QoS service model and describe its operation in detail. Section 3.1.1 starts by presenting the idea of token buckets (TBs) [15], which we use to regulate the request traffic. Section 3.1.2 introduces the Nested QoS model. Section 3.1.3 discusses how the token bucket and the Nested QoS model classify the workloads and satisfy the Properties (P1-P3) defined in Chapter 1. We also introduce how to implement the Nested QoS model in a storage system and how to apply the Nested QoS to serve multiple clients in Section 3.2. A scheduling framework and algorithm based on the Nested QoS service model are presented in Section 3.2.1. Finally, capacity estimation analysis is provided in Section 3.2.2.

3.1 Nested QoS Model

In Chapter 1 we have explained that each client in a shared storage environment has SLAs that specify the performance it will receive provided its input traffic satisfies stipulated restrictions. It requires that the workload should be properly defined by certain parameters, in order to provide the performance guarantee for each client. However, defining the restriction of a workload is not easy. Describing a workload with only throughput or burst size is not enough. For example, suppose there are

two clients that have the same throughput of 1000 requests/second and worst-case response time requirement of 5 milliseconds. Client *A* sends requests in a uniform way, which is one request per millisecond, while client *B* sends a burst of 10 requests every 10 milliseconds. The capacity required for the client *A* to meet its SLA is 1000 IOPS, compared with 2000 IOPS for Client *B*.

In [32] we analysed the factors that affect the resource requirements and workload performance, and concluded that it is the burst size, burst frequency, and throughput that determines the capacity requirement and workload performance. Inspired by this, we use the token bucket to characterize the workload in the Nested QoS model. In [32] we also found that the capacity required is not linear with the response time bound. By relaxing the response time guarantee for a small fraction of the workload, there is a very sharp reduction in the capacity needed. This is also the foundation of our original motivation of the Nested QoS. We will introduce the token bucket and the Nested QoS model in the following sections.

3.1.1 Token Bucket Traffic Envelope

In order to characterize the burst size, burst frequency and throughput, we use a TB to describe the arrival pattern of a workload. A TB has two parameters (σ, ρ) , in which σ is the bucket size and ρ is the rate at which tokens are generated. The tokens will keep increasing at the rate of ρ until reaching the bucket size of σ (overflow situation). In any time interval of length T , the total number of tokens generated is

limited by $\sigma + \rho T$. This property allows one to regulate a workload's maximum burst size (σ), throughput (ρ), and frequency of maximum-sized bursts (σ / ρ).

When using TB as the traffic envelope for a workload, the traffic specification is provided by a token bucket, which asserts that for any time interval of length T the number of requests sent by the client should be no more than $\sigma + \rho T$. A usual implementation of the traffic envelope is to assume a reservoir (the bucket) initially filled with σ tokens that is fed fresh tokens at a constant rate of ρ ; however, the maximum number of tokens in the reservoir is capped at σ . Whenever a new request arrives it removes a token from the bucket if it is not empty. As long as the request finds a token when it arrives, the traffic meets its SLAs constraint and is considered *well behaved*. A request that arrives when there is no token in the bucket is considered to be a *bad request*, and the client is considered to be *ill behaved*.

Figure 3.1 shows an example of an upper bound (heavy line) on the arrival traffic (dashed line) induced by a (σ, ρ) token bucket. The client is well behaved in the interval $[0, a)$ since all arrivals lie below the upper bound restriction, but ill-behaved between times a and b , where the arrivals exceed the upper bound constraint. The SLAs guarantee the requests of client i a response time limit of δ_i provided its arrival traffic is within the stipulated (σ_i, ρ_i) token bucket arrival constraint.

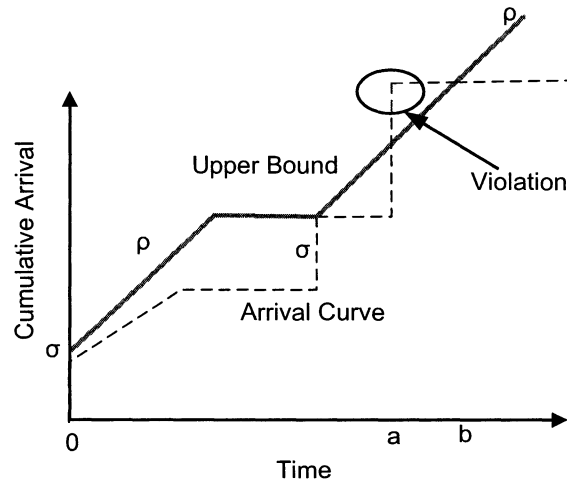


Figure 3.1 : Upper Bound on Arrivals imposed by the Token Bucket

3.1.2 Nested QoS Model and SLAs Specification

As mentioned at the beginning of this Chapter, storage workloads are very bursty, and guaranteeing a small response time to all requests requires very high capacity. However, by relaxing the tight response time bound for a small fraction of the workload there is a very sharp reduction in the capacity needed [32, 30, 31]. We formalize those properties and propose a novel Nested QoS service model to provide elastic SLA in storage systems.

Figure 3.2 shows the abstract architecture of the Nested QoS service model. The workload W of a client consists of a sequence of requests. The performance SLAs in terms of latencies are determined by multiple nested classes $C_1, C_2 \dots C_n$. Class C_i is specified by three parameters: $(\sigma_i, \rho_i, \delta_i)$, where (σ_i, ρ_i) are token bucket parameters

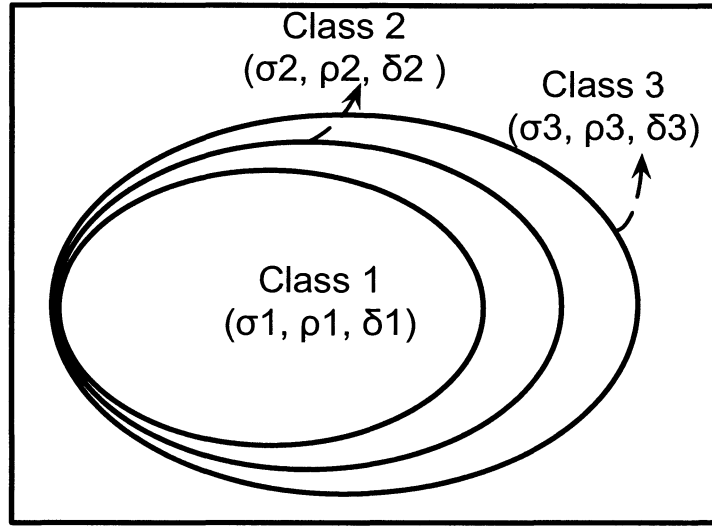


Figure 3.2 : Nested QoS Model

and δ_i is the response time bound. C_i consists of the maximally-sized subsequence of requests of W that is compliant with a (σ_i, ρ_i) token bucket: that is, the number of requests in any interval of length t is upper bounded by $\sigma_i + \rho_i t$, and no other request of W can be added to the sequence without violating the constraint. The token bucket provides an envelope on the traffic admitted to each class by limiting its burst size (σ_i) and arrival rate (ρ_i). All requests in C_i have a response time limit of δ_i . Nesting requires that $\sigma_i \leq \sigma_{i+1}$, $\rho_i \leq \rho_{i+1}$ and $\delta_i \leq \delta_{i+1}$.

For example, a 3-class Nested QoS model (30, 120 IOPS, 500ms), (20, 110 IOPS, 50ms), (10, 100 IOPS, 5ms) indicates that: all the requests in the workload that lie within the (10, 100 IOPS) envelope have a response time bound of 5ms; the requests within the less restrictive (20, 110 IOPS) arrival constraint have a latency bound of

50ms, while those conforming to the (30, 120 IOPS) arrival bound have a latency limit of 500ms.

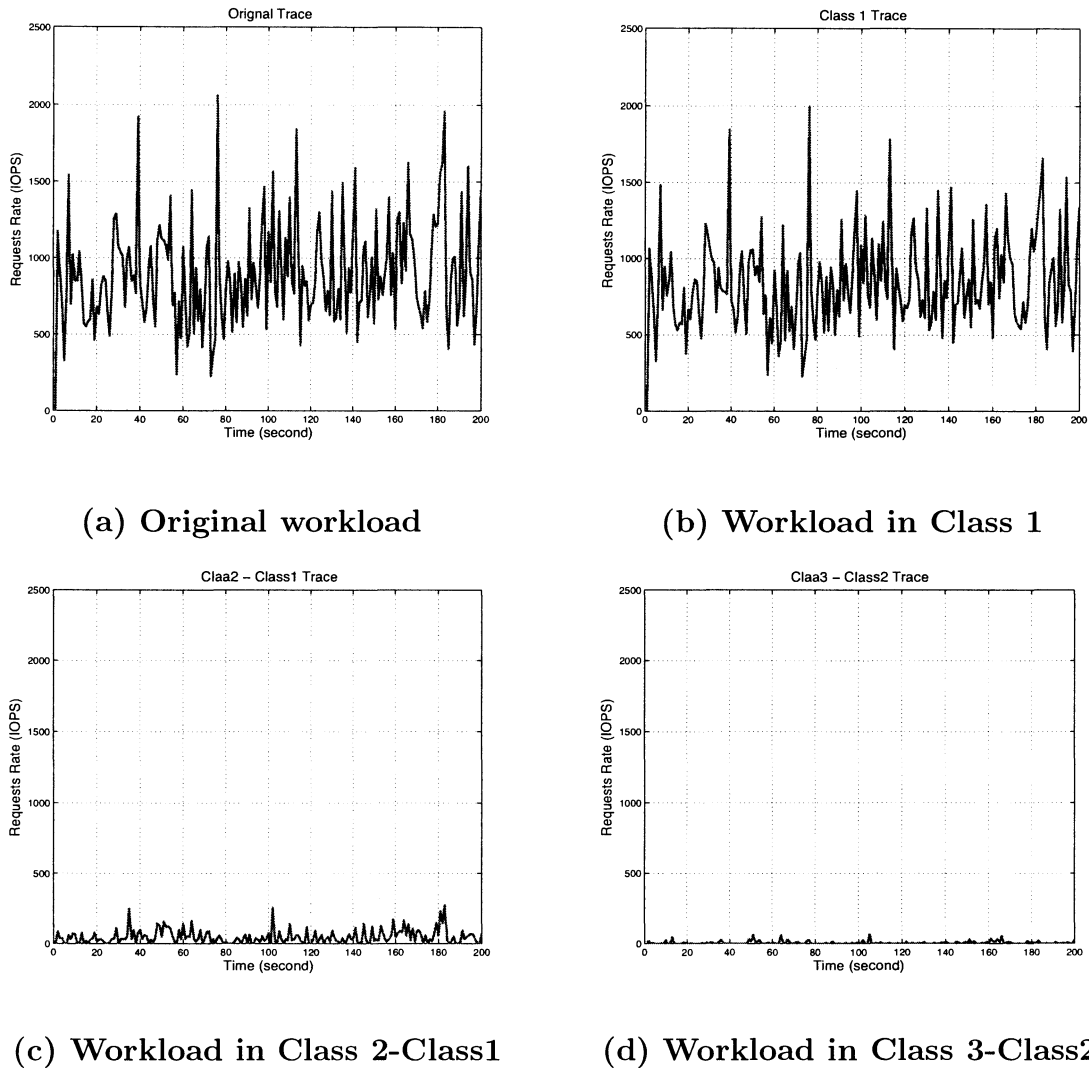


Figure 3.3 : An Example for Workload Classification

To give an example of request classification, Figure 3.3 shows the filtering of the Exchange workload from Microsoft as it goes through the token buckets.

The Nested QoS model formalizes the observation that a disproportionate fraction of server capacity is used to handle the small tail of highly bursty requests. It offers a spectrum of response time guarantees based on the burstiness of the workload and allows clients define the classes and performances flexible. In Section 3.2.3 we will discuss how to choose the TB parameters to classify the workload.

3.1.3 Model Analysis

Next we will discuss how the Nested QoS and token bucket satisfy the three properties (P1-P3) defined in Chapter 1.

When a client sends more requests than agreed to by the SLAs, we call this a *violation*. A simple approach to isolation the violation part is to police the traffic of each client and then simply *drop* the requests that exceed the arrival upper bound. These discarded (bad) request will need to be submitted again later. Such an approach may be suitable in some environments like in computer networks where the protocols automatically provide mechanisms for retransmission and recovery from dropped packet transmissions. However, storage protocols are not designed to handle lost requests; dropping requests from oversubscribed clients in this situation will result in a cascading series of undesirable events, possibly culminating in the eventual failure of the application.

The *pClock* algorithm [7] provides a second approach to this problem based on *delaying* the requests of the ill-behaved client. A bad request arriving at time t is

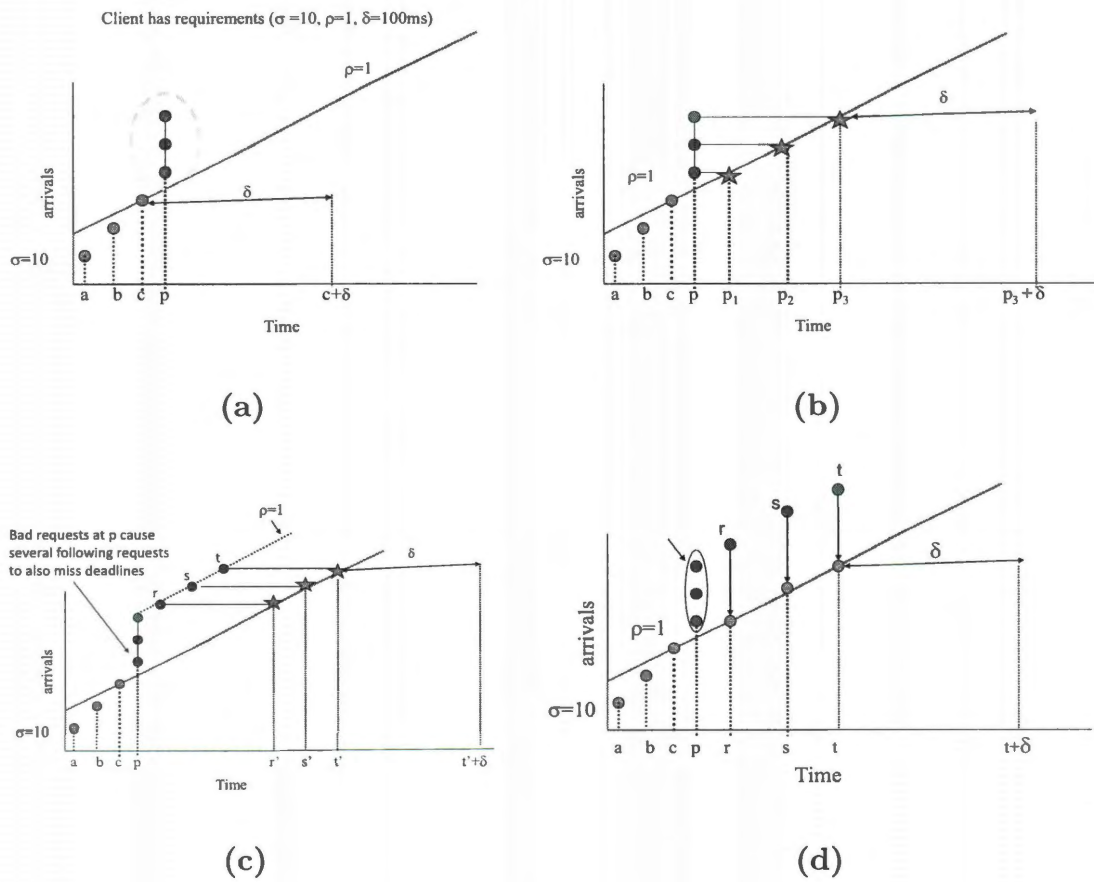


Figure 3.4 : Effect of Bad Requests on Later Requests. (a) Three bad requests arrive at p . (b) Bad requests delayed to conform to SLAs. (c) Later requests delayed by bad requests at p . (d) Later requests meet deadline if requests at p are removed

treated as if it had actually arrived at a later time $t' > t$, such that at t' the arrival would meet the upper bound. Figure 3.4 shows the arrival traffic of a hypothetical workload. In Figure 3.4(a) there are three bad requests arriving at time p . These will be treated as if they had actually arrived at the later times p_1 , p_2 , and p_3 respectively as shown in Figure 3.4(b). Instead of scheduling them to finish by their true deadlines $p + \delta$, they will instead be treated as if their deadlines were delayed to $p_1 + \delta$, $p_2 + \delta$ and $p_3 + \delta$ respectively. The delay makes the requests appear to satisfy the arrival specification and thereby protects other clients from being affected. However the assigned deadlines for these bad requests are later than their true deadlines, and hence are not guaranteed to meet the response time SLA.

A drawback of delaying bad requests as described above is the cascading effect that can have on subsequent requests. Continuing with the previous example, suppose that the subsequent requests of the client after time p are at times r , s , t and so on, at a rate $\rho = 1$ as stipulated by the SLAs (see Figure 3.4(c)). Because the three bad requests at p were delayed, the requests following them will also need to be delayed to remain within their SLAs constraint. Hence, these requests will be treated as arriving at times r' , s' and t' with correspondingly delayed deadlines $r' + \delta$, $s' + \delta$ and $t' + \delta$ respectively.

Potentially all future requests of this client could miss their deadlines because of the small overburst that occurred in the past. Hence, while the technique of delaying bad requests can insulate other clients from the ill-behaved one, it violates

our second property P2: it does not isolate the good portions of a client’s workload from the effects of its own badly behaved parts. If we remove the three bad requests arriving at time p from the input stream, the requests arriving at r, s, t will be good and will be finished by true deadlines $r + \delta, s + \delta$ and $t + \delta$ as shown in Figure 3.4(d). In the Nested QoS, the removed requests will not be discarded, but will instead be classified as a higher class (not shown in the figure) requests with a later deadline.

To sum up, the Nested QoS model addresses **P1** by classifying the bad requests into a higher level class, which has low priority and won’t compete with the requests from other well-behaved clients. Also, the Nested QoS model addresses **P2** by *decomposing* the workload into different classes and independently scheduling them with differing QoS requirements. We refer to this as providing *graduated QoS* guarantees. The server capacity required to meet QoS guarantees depends upon the capacity requirements of the individual clients and the scheduling policy. A typical performance SLAs guarantees a client’s requests a maximum response time of δ provided it is ”well-behaved”, *i.e.* it conforms to the arrival upper bound implied by a specified token bucket. Each client i estimates its capacity μ_i based on its maximum burst size σ_i , response time δ_i and average rate ρ_i for each class.

In addition to using decomposition to limit individual capacity requirements, capacity can be reduced by using scheduling to exploit the heterogeneity in the QoS requirements of concurrent clients. We discuss two scheduling policies Fair Queuing (FQ) and Earliest Deadline First (EDF) below.

Fair Queuing (FQ) An FQ scheduler divides the available capacity among the n clients in a fine-grained manner in proportion to their weights. The system capacity is $C_{FQ} = \sum_i \mu_i$. Client i is assigned a weight $w_i = \mu_i/C_{FQ}$ so that it receives at least μ_i capacity at a fine-grained intervals during operation.

Earliest Deadline First (EDF) In contrast to FQ, the EDF scheduling policy minimizes the capacity requirements needed to meet a set of deadlines, by exploiting the differences in the response times of different clients. The *p*Clock scheduler [7] uses EDF scheduling and always selects the request with the smallest (earliest) deadline. A simple example to illustrate the potential benefit follows. Consider two clients that each send a burst of 50 requests every 100ms. The first client requires a response time of 50ms for its requests and the second requires 100ms. The capacity needed for the first client is 50 requests/50 ms = 1000 IOPS, while that for the second client is 50 requests/100 ms = 500 IOPS. A fair scheduler would use a server of 1500 IOPS and multiplex the two workloads evenly in a 2 : 1 ratio; in the first 50ms it would complete 50 requests of client 1 and 25 requests of client 2, while in the next 50ms it would do the remaining 25 requests of client 2. Both clients meet the deadlines for all their requests.

An EDF scheduler would change the order of service so that it does all 50 of client 1's requests first (since they have a smaller deadline), followed by the 50 requests of client 2. This requires a capacity of only 1000 IOPS to finish all requests by their deadlines. Due to this potential for reduced capacity we will use an EDF-based

scheduler in this thesis. However, simple direct use of EDF will not work, in the sense that isolation properties P1 and P2 can be violated if not done correctly. Intuitively this is because clients are much more closely coupled under EDF scheduling so the requirements for flexibility clash with the need for strict regulation. For instance, suppose client 1 misbehaved and sent 100 requests instead of 50. Since all 100 requests will have a shorter deadline than client 2's requests, they will all be served first in an EDF schedule, completing after 100ms. All the requests of client 2 will have missed their deadline. In contrast, a fair scheduler will not delay any of client 2's requests past their deadline, and delay only the requests of the offending client, client 1.

The capacity estimation for Nested QoS is presented in Section 3.2.2.

3.2 Scheduling Framework Based on the Nested QoS Model

In this section, we present the scheduling framework and algorithm based on the Nested QoS model, and apply the model to multiple clients. Figure 3.5 shows a overall architecture of our system for a 3-level Nested QoS model for n client. The framework consists of two components: *request classification* and *request scheduling*.

Request Classification The workload from each client is first classified by their own *Request Classifier*. The *Request Classifier* classifies the requests from clients i into several classes $C_{i,j}$, each of which provides a different response time guarantee $\delta_{i,j}$. Figure 3.6 shows the detailed information of *Request Classifier*.

The Request Classifier is implemented using a cascade of token buckets, $B_1, B_2 \dots B_n$

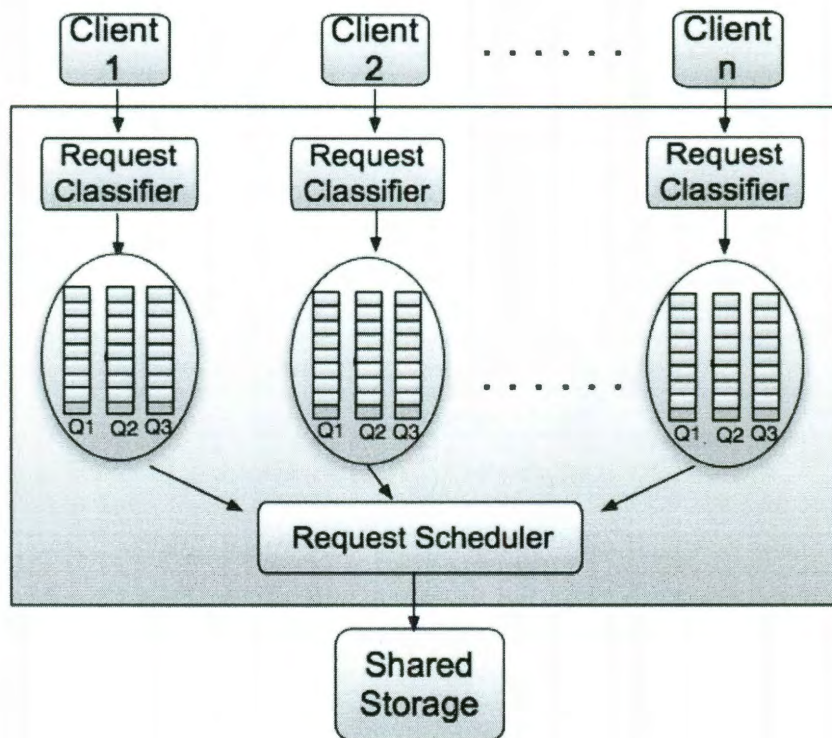


Figure 3.5 : Scheduler Framework

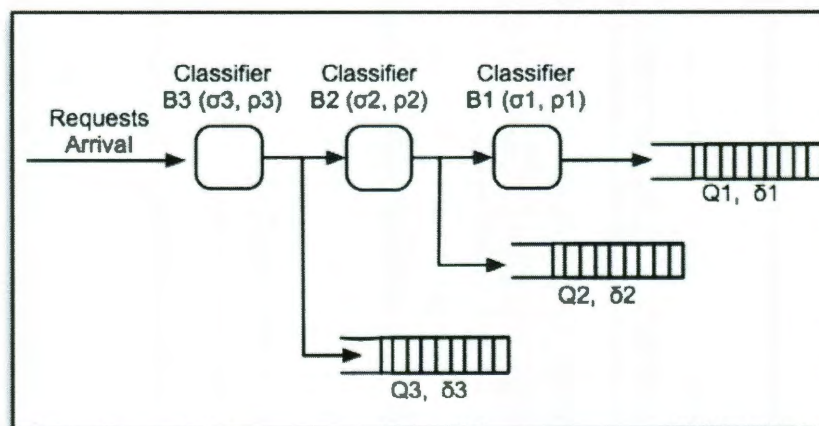


Figure 3.6 : Request Classifier with Token Bucket

(innermost is B_1). B_i has parameters (σ_i, ρ_i) that regulates the number of requests that pass through it in any interval. Initially B_i has σ_i tokens; an arriving request removes a token from the bucket (if there is one) and passes thorough to B_{i-1} (or Q_1 if i is 1); if there are no tokens in B_i the request goes into the queue Q_{i+1} instead. B_i is filled with tokens at a constant rate ρ_i , but the maximum number of tokens is capped at σ_i . The buckets filter the arriving workload so that queue Q_1 receives all requests of class C_1 , Q_2 receives requests of $C_2 - C_1$, and Q_3 receives requests of $C_3 - C_2$.

Request Scheduling The queues of each client are exposed to the scheduler, which is responsible for multiplexing the server among the different clients so that response times of individual requests in the clients can be guaranteed. By ensuring that requests in queue $Q_{i,j}$ meet a response time of $\delta_{i,j}$, the SLAs of the Nested QoS model can be met. The scheduler services requests across the queues within a client based on their deadlines using an Earliest Deadline First (EDF) policy. In Section 3.2.2 we provide a method to compute the capacity required to meet the SLAs specified by the Nested QoS model parameters.

3.2.1 Scheduling Algorithms

Request Arrival: When the r^{th} request from client i arrives at time t it will go through the *Request Classifier* as shown in Figure 3.6. First the *Request Classifier* invokes **TokenUpdate()**, which updates the number of tokens in each class of TB

Symbols	Meaning
$C_{i,j}$	Class j of client i
$B_{i,j}$	Token Bucket at level j of client i
$Q_{i,j}$	Queue at level j of client i
$\sigma_{i,j}, \rho_{i,j}$	Token Bucket parameters of $C_{i,j}$
$s_{i,j}^r$	Start tag of request r in $Q_{i,j}$
$f_{i,j}^r$	Finish tag of request r in $Q_{i,j}$
$MinF_{i,j}$	Minimum Finish tag of pending requests in $Q_{i,j}$
Φ	Set of requests waiting for server scheduling

Table 3.1 : Symbols

1. Request Arrival (request r , client i , time t):**begin**TokenUpdate($B_{i,j}$, t) for all levels j of client i ;RequestClassification(r , i , t);If r is classified as $C_{i,k}$; RequestTagging(r , $C_{i,k}$, t); Insert r into $Q_{i,k}$;**end****2. Scheduler:****begin**If (Φ is empty)

return;

Let t be current time; Select r in Φ with the smallest finish tag $MinF_{i,j}$; Dispatch request r to server;**end****Algorithm 1: Algorithm Structure**

TokenUpdate($B_{i,j}, t$):

begin

/*The biggest burst allowed is bounded by $\sigma_{i,j}$ */

Let Δ be the time difference between current time

and the previous time that $B_{i,j}$ was updated;

$tokens(i, j) += \Delta * \rho_{i,j}$;

If ($tokens(i, j) > \sigma_{i,j}$)

$tokens(i, j) = \sigma_{i,j}$;

end

RequestClassification(r , client i , t):

begin

Find the token bucket with the smallest index k that satisfies:

$(tokens(i, k + 1) \geq 1) \ \& \ (tokens(i, k) < 1)$

$\forall j \geq (k + 1), tokens(i, j) -= 1$;

Classify $r \in C_{i,k+1}$;

end

RequestTagging($r, C_{i,k}, t$):

begin

$s_{i,j}^r = t$;

$f_{i,j}^r = s_{i,j}^r + \delta_{i,k}$;

If ($Q_{i,j}$ is empty)

$MinF_{i,j} = f_{i,j}^r$;

end

Algorithm 2: Algorithm Components

at the current time. Then it invokes **RequestClassification()** which classifies the request into a specific class. If there are no tokens in $B_{i,j-1}$ but there is at least one token in $B_{i,j}$, it means the workload arrival satisfies the traffic limitation defined by $B_{i,j}$ but does not satisfy $B_{i,j-1}$. The r^{th} request is then classified into $C_{i,j}$ and placed into $Q_{i,j}$. The start tag, s_i^r , is set to the current time t and f_i^r is set to $s_i^r + \delta_{i,j}$. If all TBs of client i have at least one token at t , it means the r^{th} request satisfies the limitation of the most stringent class $C_{i,1}$, the request is placed in $Q_{i,1}$ and assigned a response time of $\delta_{i,1}$. In the process of **RequestClassification()**, the tokens of each bucket are also updated.

Scheduler: The system scheduler selects the request to dispatch to the server based on an Earliest Deadline First (EDF) policy among the requests waiting in the queues. Because the finish tag in each queue of each client is in an ascending order, the scheduler only needs to compare the finish tag of the first request from each queue, which reduces the compute complexity. A complete description is provided in Algorithms 1 and 2.

Summary We summarize the salient features of our method. First, the use of graduated QoS allow the system to provide very good QoS guarantees at a fraction of the capacity required to provide 100% guarantees. The use of an EDF scheduler allows the capacity requirements to be further reduced by exploiting the heterogeneity in

the client QoS requirements. The scheduler thereby addresses property P3. Next, the use of *classification* rather than simple delay allows our scheduler to uphold property P2 of isolating the good and bad portions of an individual workload. The TB based traffic envelopes can regulate workloads automatically and allow us to guarantee the isolation property P1.

In Chapter 4 we provide experimental validation of our method using several block-level real storage traces to validate specific features of the Nested QoS, as well as the benefits possible in practice as well.

3.2.2 Capacity Planning

In the Nested QoS model, the workload W consists of a sequence of requests arriving at times $1, 2, 3, \dots$. The classification splits W into classes C_1, C_2, \dots, C_n . C_i consists of the requests of W that are output by the token bucket B_i . All requests in C_i have a response time no more than δ_i . From the nested definition, we require that $\sigma_i \leq \sigma_{i+1}$, $\rho_i \leq \rho_{i+1}$ and $\delta_i \leq \delta_{i+1}$. The problem is to estimate the server capacity required to meet the SLAs. We define a *busy period* to be an interval in which there are one or more requests in the system.

Lemma 1 With EDF scheduling, the capacity of a single workload required for all requests to meet their deadlines in the n-level Nested QoS model satisfies:

$$\forall j, C \leq \max\{\sigma_j/\delta_j + \sum_{k=1}^{j-1}(1+\rho_k \times (\delta_{k+1}-\delta_k))/\delta_j, \rho_j\}. \quad 1 \leq j \leq n \quad (1)$$

Proof: We bound the maximum number of requests that need to finish by time t , where $t = 0$ is the start of a system busy period. Define $\eta_i(t)$ to be the number of tokens in bucket B_i at time t . By definition, $\eta_i(0) = \sigma_i$ for all $i = 1, \dots, n$. Define $N_t(a, b)$ to be the maximum number of requests with deadline less than t , which enter any of the queues Q_1, Q_2, \dots, Q_n in the interval (a, b) . Let $j, 1 \leq j \leq n$, be the largest index for which $t \geq \delta_j$. Define $\tau_i = t - \delta_i, 1 \leq i \leq j$, and for notational convenience let $\tau_{j+1} = 0$. Then

$$N_t(0, t) = \sum_{i=1}^j N_t(\tau_{i+1}, \tau_i) \quad (2)$$

Now $N_t(\tau_{i+1}, \tau_i)$ consists exactly of the requests that have been admitted by bucket B_i in $[\tau_{i+1}, \tau_i)$. Hence,

$$N_t(\tau_{i+1}, \tau_i) \leq \eta_i(\tau_{i+1}) + \rho_i \times (\tau_i - \tau_{i+1}) - \eta_i(\tau_i) \quad (3)$$

Summing both sides of formula (3) for all $i = 1, \dots, j$

$$\sum_{i=1}^j N_t(\tau_{i+1}, \tau_i) \leq \sum_{i=1}^j \rho_i \times (\tau_i - \tau_{i+1}) + \sum_{i=1}^j (\eta_i(\tau_{i+1}) - \eta_i(\tau_i)) \quad (4)$$

Rewriting the last summation of the right hand side of the equation (4):

$$\sum_{i=1}^j (\eta_i(\tau_{i+1}) - \eta_i(\tau_i)) = \sum_{i=1}^{j-1} (\eta_i(\tau_{i+1}) - \eta_{i+1}(\tau_{i+1})) + \eta_j(\tau_{j+1}) - \eta_1(\tau_1) \quad (5)$$

we also have

$$\eta_i(t) \leq \eta_{i+1}(t) + 1 \quad (6)$$

(This can be proved by induction over the arrival instants of requests. The number of tokens in bucket B_i at any time is no more than the number of tokens in any bucket $B_j, j > i$. For the base case, equation (6) holds since $\sigma_i \leq \sigma_{i+1}$, for all $i = 1, \dots, n-1$. The details of the proof are omitted.)

By substituting and dropping all negative terms of equation (5):

$$\sum_{i=1}^j N_t(\tau_{i+1}, \tau_i) \leq (j-1) + \sum_{i=1}^j (\rho_i \times (\tau_i - \tau_{i+1})) + \eta_j(\tau_{j+1}) \quad (7)$$

Now,

$$\eta_j(\tau_{j+1}) = \eta_j(0) = \sigma_j \quad (8)$$

$$\tau_i - \tau_{i+1} = \delta_{i+1} - \delta_i, \quad i = 1, \dots, j-1 \quad (9)$$

$$\tau_j - \tau_{j+1} = t - \delta_j \quad (10)$$

Hence,

$$\sum_{i=1}^j N_t(\tau_{i+1}, \tau_i) \leq \sigma_j + \sum_{i=1}^{j-1} (1 + \rho_i \times (\delta_{i+1} - \delta_i)) + \rho_j \times (t - \delta_j) \quad (11)$$

The capacity (C) required to finish these $N_t(0, t)$ requests by time t is upper bounded by $N_t(0, t)/t$. Hence:

$$C \leq \sigma_j/t + \sum_{i=1}^{j-1} (1 + \rho_i \times (\delta_{i+1} - \delta_i))/t - \rho_j \times \delta_j/t + \rho_j \quad (12)$$

Now, if $(\sigma_j + \sum_{i=1}^{j-1} (1 + \rho_i (\delta_{i+1} - \delta_i))) < (\rho_j \times \delta_j)$ the inequality reduces to:

$$C \leq \rho_j \quad (13)$$

Otherwise, the RHS is maximized when t takes on its smallest value, which is δ_j . In this case, the inequality reduces to:

$$C \leq \sigma_j/\delta_j + \sum_{i=1}^{j-1} (1 + \rho_i \times (\delta_{i+1} - \delta_i))/\delta_j \quad (14)$$

The above two inequalities must hold for all values of t , and hence for all possible values of j , $1 \leq j \leq n$.

Putting it all altogether we get:

$$C \leq \max\{\sigma_j/\delta_j + \sum_{i=1}^{j-1} (1 + \rho_i \times (\delta_{i+1} - \delta_i))/\delta_j, \rho_j\}, \quad \forall j, 1 \leq j \leq n$$

Lemma 2 With EDF scheduling, the capacity of a single workload required for all requests to meet their deadlines in the Nested QoS model, when all ρ_i are equal to ρ , is given by:

$$C \leq \max\{\sigma_j/\delta_j + (j-1)/\delta_j + \rho(1 - \delta_1/\delta_j), \rho\}. \quad \forall j, 1 \leq j \leq n \quad (15)$$

For the case when all ρ_i are equal to ρ , and the class parameters are multiples of the base value:

Lemma 2.1: Let $\alpha = \delta_{i+1}/\delta_i$, $\beta = \sigma_{i+1}/\sigma_i$ and $\lambda = \beta/\alpha$ be constants. The server capacity required to meet SLAs is no more than: $\max_{1 \leq j \leq n} \{\rho, \lambda^j(\sigma_1/\delta_1) + (j-1)/\alpha^j\delta_1 + \rho(1 - 1/\lambda^j)\}$. For $\lambda < 1$, the server capacity is bounded by $\sigma_1/\delta_1 + \rho$, which is less than twice the capacity required for servicing C_1 .

Lemma 3: Consider a shared server environment where there are m workloads each with n classes. Let $\sigma_{i,j}$, $\rho_{i,j}$, $\delta_{i,j}$ denote the parameters for class j of workload i . Suppose the $\delta_{i,j}$ are arranged in non-decreasing order, and denoted by Δ_k ($1 \leq k \leq m * n$). Define $N_t^i(a, b)$ to be the maximum number of requests in workload i with deadline less than t , which enter any of the queues Q_1, Q_2, \dots, Q_n in the interval (a, b) . Then the maximum number of requests that need to finish by time t from all workloads is $N_t^{all}(0, t) = \sum_{i=1}^m N_t^i(0, t)$. $N_t^i(0, t)$ is given by (2) and (11). Then the total capacity C_{all} for m workloads are given:

$$\forall k, C_{all} \leq \max\{\sum_{i=1}^m N_{\Delta_k}^i(0, \Delta_k)/\Delta_k, \sum_{i=1}^m \rho_{i,n}\} \quad (17)$$

and $N_{\Delta_k}^i(0, \Delta_k)$ is given by (2) and (11).

3.2.3 Workload Parameters

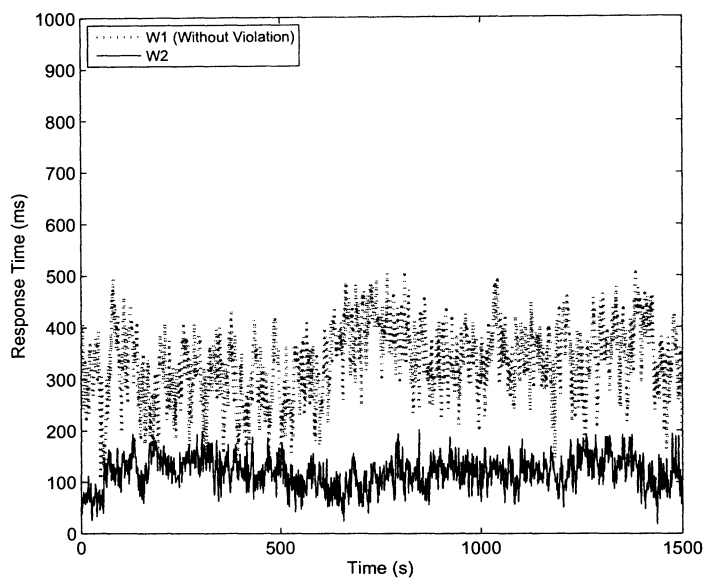
In this section, we describe how the Nested QoS parameters of a workload will typically be determined. The client first decides the number of classes and their sizes (as a fraction of workload size) by empirically profiling the workload to achieve a satisfactory tradeoff between capacity required (cost) and performance. (Usually three classes appear to be sufficient over a variety of workloads.) Using a decomposition algorithm (see [31]) one can determine the minimum capacity κ_1 required for a fraction f_1 of the workload to meet the deadline δ_1 . We choose $\rho_1 = \kappa_1$ and $\sigma_1 = \rho_1 \delta_1$. We similarly profile each of the classes, and find a pair of (σ_2, ρ_2) which satisfy that a fraction of f_2 requests fall into class 2 defined by (σ_2, ρ_2) ; and then find a pair of (σ_3, ρ_3) which satisfy that a fraction of f_3 requests fall into class 3 defined by (σ_3, ρ_3) . According to the Nested QoS definition, the parameters should satisfy: $\sigma_1 \leq \sigma_2 \leq \sigma_3$, and $\rho_1 \leq \rho_2 \leq \rho_3$. The method of how to choose the parameters is described in [33].

Chapter 4

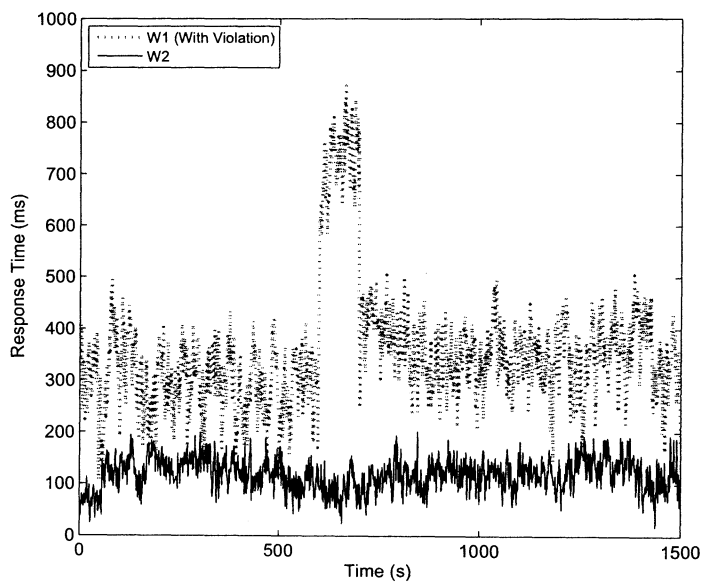
Performance Evaluation

4.1 Experimental Setup

In this section, we describe the results of an empirical evaluation of our scheduling method (also referred to as Graduated QoS) using both a process-driven system simulator Yacsim [34] and a prototype in Linux kernel. In Section 4.2 to Section 4.4 we use Yacsim simulation to evaluate the Nested QoS model. In Section 4.5 we use results from prototype in Linux to show the performance of the Nested QoS model. In the experiments, we used three types of real block-level storage application traces from the UMass Storage Repository [35]. We conducted experiments focused on illustrating the properties P1 to P3 detailed in Chapter 1: *(i)* Can we isolate badly-behaved workloads from good ones so that they do not affect the performance of the latter? *(ii)* Can we localize regions of bad behavior of a single workload so as to avoid affecting its well-behaved regions? *(iii)* Can we provide high quality of service with low provisioned capacity?

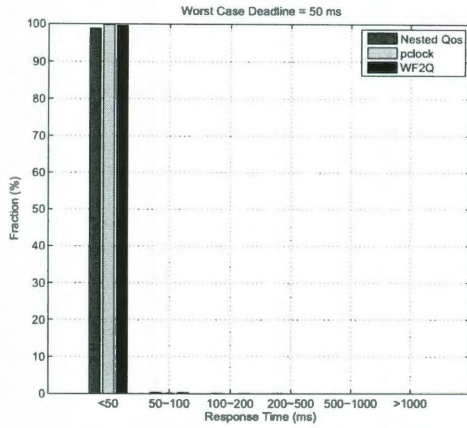


(a) Arrival pattern for W1 and W2

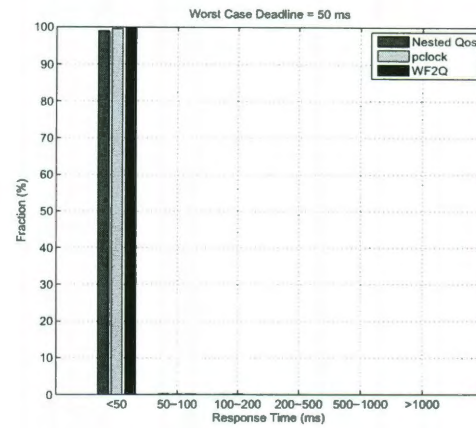


(b) Arrival pattern for W1(violation) and W2

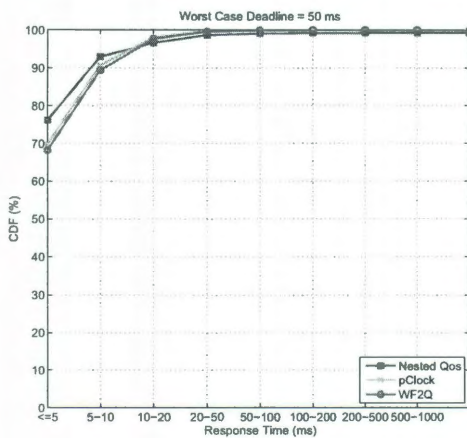
Figure 4.1 : Arrival pattern for (a) Both W1 and W2 are well-behaved. (b) W1 violates SLAs and sends more requests during time 600s-700s



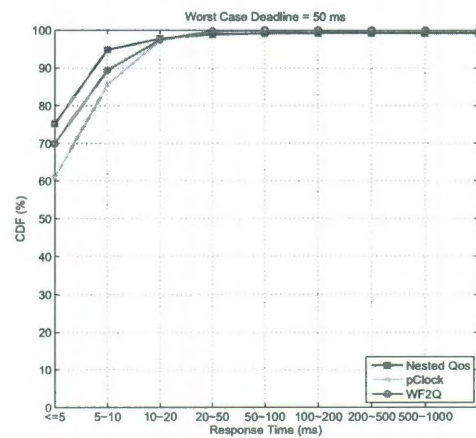
(a) Response time distribution of W1



(b) Response time distribution of W2

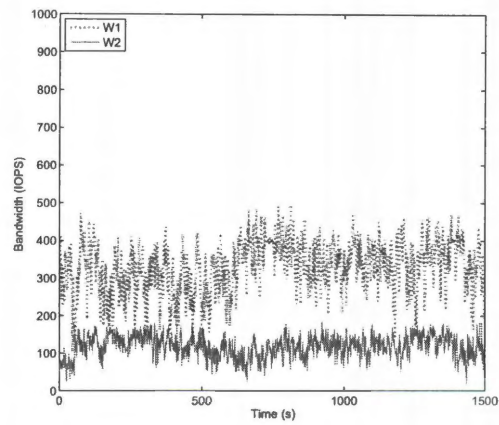


(c) Response time CDF of W1

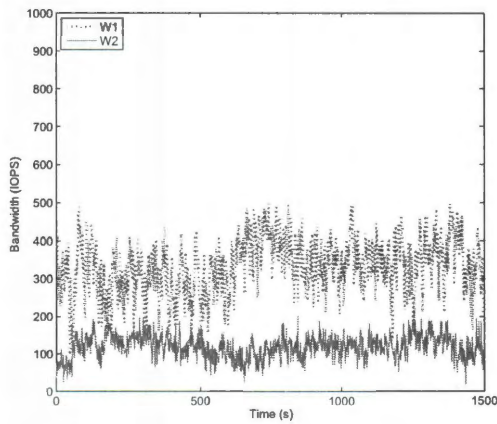
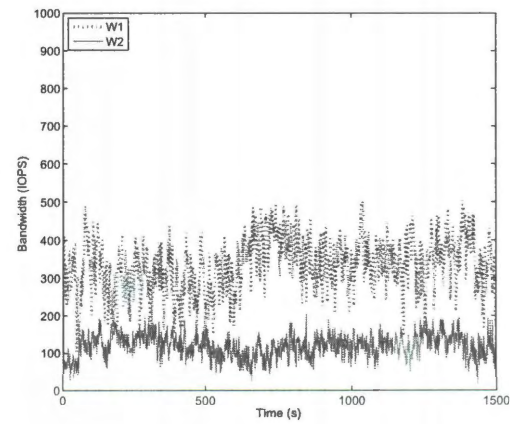


(d) Response time CDF of W2

Figure 4.2 : W1 and W2 are well-behaved workloads. Response time distribution and CDF of W1 and W2 with three scheduling methods: Nested QoS, pClock, WF2Q.



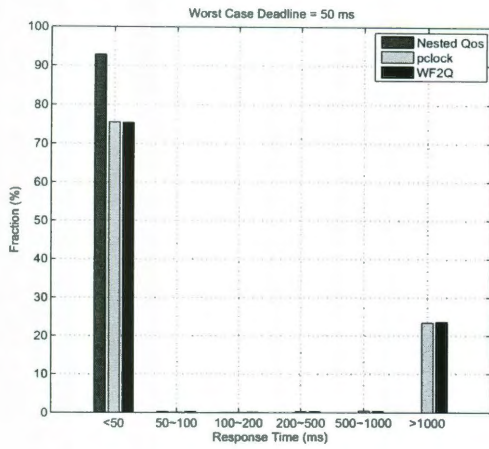
(a) Bandwidth allocation by Nested QoS

(b) Bandwidth allocation by *p*Clock

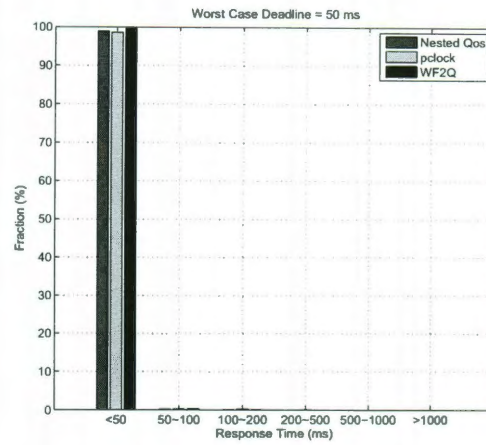
(c) Bandwidth allocation by WF2Q

Figure 4.3 : The bandwidth allocation for well-behaved W1 and W2 by three methods:

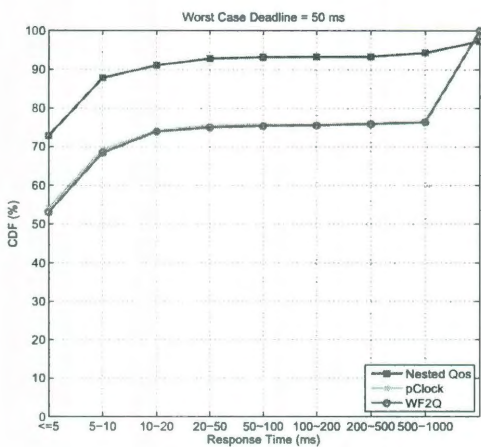
Nested QoS, *p*Clock, WF2Q.



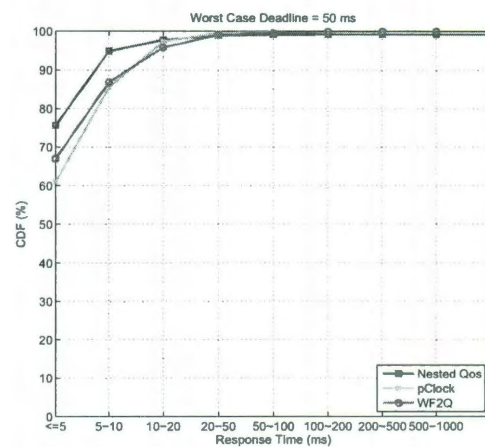
(a) Response time distribution of W1



(b) Response time distribution of W2

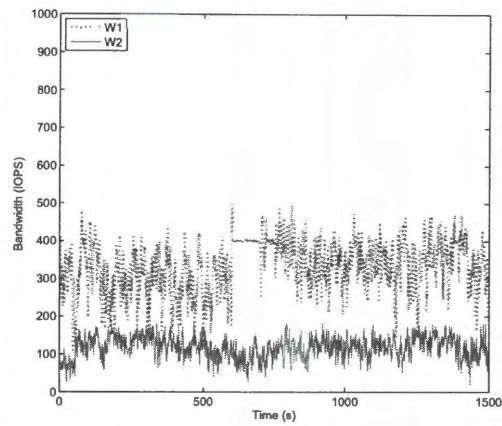


(c) Response time CDF of W1

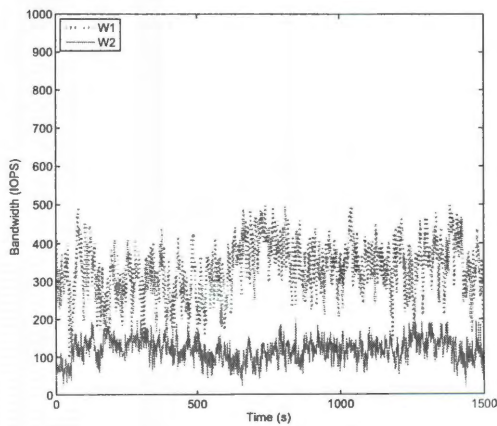
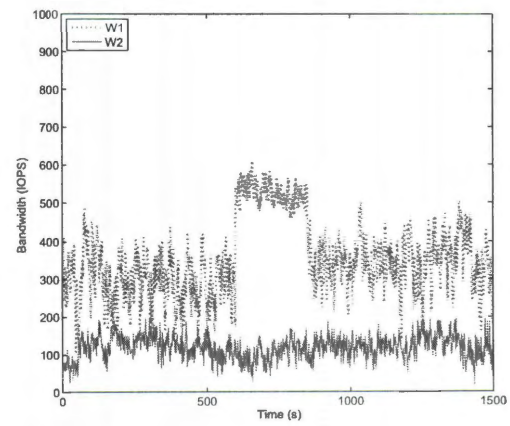


(d) Response time CDF of W2

Figure 4.4 : Response time distribution and CDF of W1(violation) and W2 with three scheduling methods: Nested QoS, pClock, WF2Q.



(a) Bandwidth allocation by Nested QoS

(b) Bandwidth allocation by *p*Clock

(c) Bandwidth allocation by WF2Q

Figure 4.5 : The bandwidth allocation for W1 (violation) and W2 by three methods: Nested QoS, *p*Clock, WF2Q.

4.2 Workload Isolation

Workload isolation is a basic requirement in shared storage systems. In this experiment we explore how well the Nested QoS can isolate badly-behaved workloads from well-behaved ones.

In the experiment, we use two block-level workloads $W1$ and $W2$. $W1$ is a WebSearch workload with a long term average arrival rate of 330 IOPS; $W2$ is a Financial Transaction workload with a long term average arrival rate of 120 IOPS. The arrival patterns are shown in Figure 4.1(a). In a second experiment $W1$ increases its instantaneous arrival rate to around 700 IOPS between time 600 and 700 seconds, as shown in Figure 4.1(b). We compared three schedulers: Nested QoS, p Clock and WF2Q [14]. We first look at how the three schedulers isolate $W1$ and $W2$ when both of them are well-behaved, as shown in Figure 4.1(a). By profiling the workloads, the token bucket parameters for $W1$ and $W2$ are set to $(20, 330IOPS)$, $(40, 360IOPS)$, $(200, 400IOPS)$ and $(7, 130IOPS)$, $(14, 143IOPS)$, $(28, 158IOPS)$ respectively. A system capacity of 628 IOPS is provisioned for the two traces. With this capacity, all methods can guarantee that no less than 90% of the requests finish within a deadline of $50ms$, and 95% of the requests finish within a deadline of $500ms$, and 100% of the requests of both workloads will finish in no more than $5000ms$. Figure 4.2(a) and (b) show the response time performance of $W1$ and $W2$ when both workloads are well behaved, and Figure 4.2(c) and (d) show the corresponding response time CDF of $W1$ and $W2$. Figure 4.3(a), (b) and (c) show the bandwidth allocation for $W1$

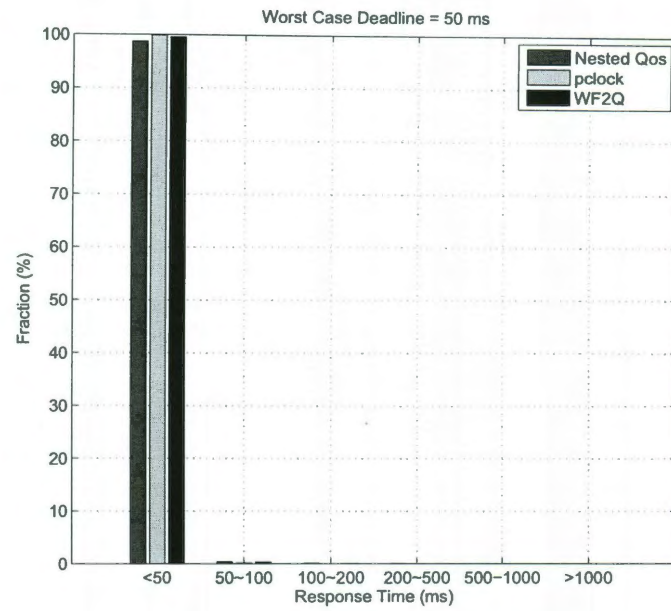
and $W2$. Both of the workloads bandwidth ($W1$ (330 IOPS) and $W2$ (120)IOPS) are guaranteed by all three methods, as shown in Figure 4.3.

Now we look at the performance of $W1$ and $W2$ when $W1$ violates SLAs. A good method should isolate the behavior of $W1$ and guarantee the performance of $W2$. Figure 4.4(a) and (b) shows the response time distribution of $W1$ (violation) and $W2$ when $W1$ violates its SLAs. The response time CDF of $W1$ (violation) and $W2$ are also shown in Figure 4.4(c) and (d). From Figure 4.2(b)(d) and Figure 4.4(b)(d) we can see that the well-behaved workload $W2$ is isolated from the bad behavior of $W1$. The performance of $W2$ does not change when $W1$ sends more requests. Performance of $W1$ is degraded because it sent more requests during $600s - 700s$, as shown in Figure 4.2(a)(c) and Figure 4.4(a)(c). A notable fact in Figure 4.2(a)(c) and Figure 4.4(a)(c) is that all the three methods show a performance degradation for $W1$, but the degradation is different in the three cases. Nested QoS can still guarantee that 94% of the requests meet their deadline, while $pClock$ and WF2Q are noticeably degraded to 75.4% and 75.3%. Theoretically, $pClock$ using EDF scheduler should have better performance than WF2Q, because the EDF scheduler is able to use the deadline difference from $W2$ to reduce the response time of $W1$, without affecting the performance of $W2$, while the WF2Q scheduler strictly allocates the capacity in proportion to the weights; hence the excess capacity is used to decrease the response time of the well-behaved flow even below its required value, and is not used to reduce the penalty faced by $W1$. In this experiment, the deadline is set to

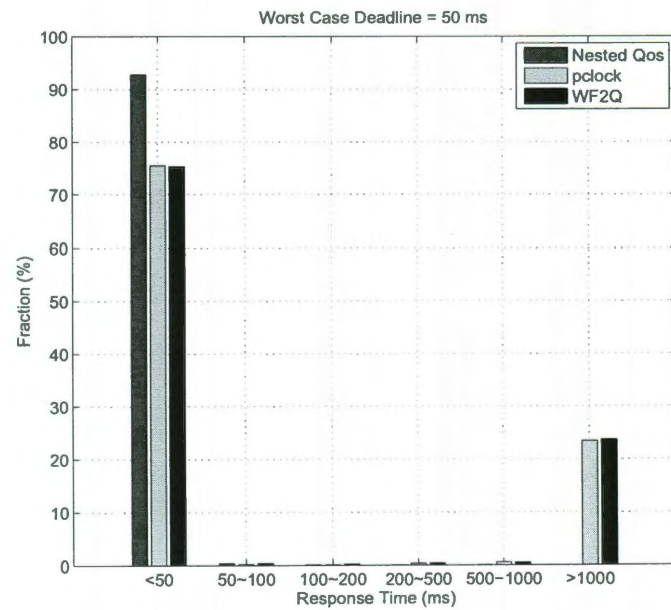
be 50 ms for both $W1$ and $W2$, and $pClock$ cannot use the spare capacity from $W2$. So the performance of $pClock$ and $WF2Q$ are almost the same. In Section 4.4 we provide the comparison of $pClock$ and $WF2Q$ with different deadline limits for the multiplexed workloads, and the results show the advantage of $pClock$ over the $WF2Q$ scheduler. In general, Nested QoS outperforms the other two methods because of its ability to isolate bad regions. We will explain that in detail next.

4.3 Bad-Region Isolation

Next we will explore how our method isolates the bad-region of a workload without affecting the good-regions, and maximizes the number of requests that meet their deadline. We use the same workloads described in Section 4.2; $W1$ has an average arrival rate 330 IOPS and $W2$ an average rate 120 IOPS. The deadlines for $W1$ and $W2$ are 50ms. The server capacity of 628 IOPS is provided for all the three scheduling methods. In the experiment $W2$ is always well behaved, while $W1$ violates its SLA by sending requests at a rate of about 700 IOPS during the 600s-700s interval (as shown in Figure 4.1 (b)). This corresponds to exceeding the stipulated arrivals by about 6% for the whole trace. As shown in Figure 4.6(a) and (b), Nested QoS allows a much greater fraction of $W1$ (about 94%) to meet its deadline compared to 75.4% and 75.3% achieved by $pClock$ and $WF2Q$, respectively. The measured response times during and after the badly-behaved region are shown in Figures 4.7(a) and (b) for the Nested QoS and $pClock$ schedulers, respectively. As can be seen, with Nested QoS

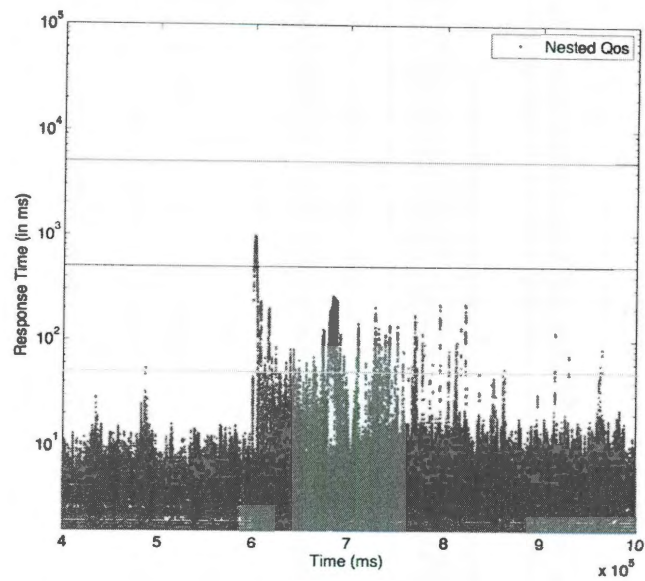


(a) Response time of W1

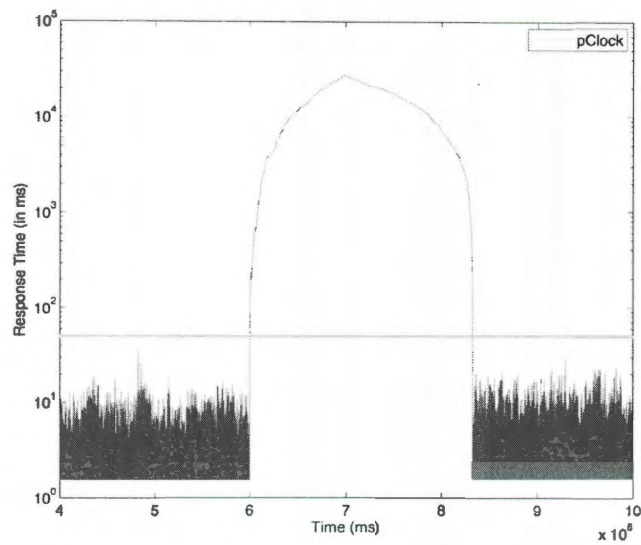


(b) Response time of W1 (violation)

Figure 4.6 : Response time distribution for W1 and W1 (violation) with three scheduling methods: Nested QoS, pClock, WF2Q.



(a) Response time of W1 (violation) with Nested QoS



(b) Response time of W1 (violation) with pClock

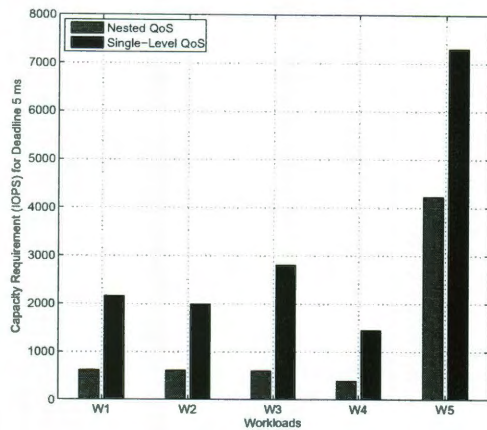
Figure 4.7 : W1 violates its SLA and sends more requests from 600s to 700s. Nested QoS isolates the bad region and still guarantee the well-behaved part. However pClock delays all of W1's requests from 600s all the way up to 790s.

most of the requests during this interval still meet their deadline, and only a few of them have longer response time. The well-behaved requests after this region (after $t = 700$ s) are not affected. In contrast, *pClock* delays all the requests of *W1* not only during the interval $(600 - 700)$ s, but all the way after the burst to about 790s. This is because when the violation happens the Nested QoS isolates the badly-behaved requests by moving them out of this request stream to a higher level class and allow well-behaved requests after the violation to meet their guaranteed deadlines.

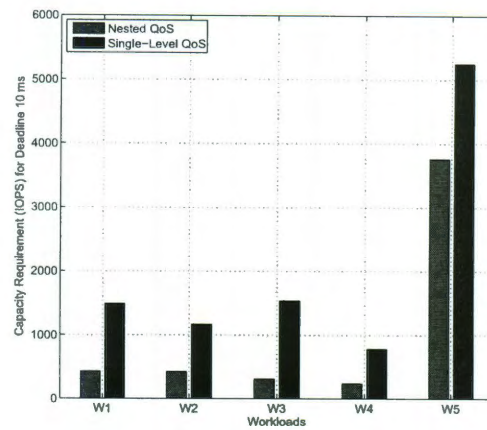
The performance of *W2* is the same with or without the violation by *W1*. We do not show the performance of *W2* because it is isolated from *W1*. We also compared the performance of *W2* using the WF2Q scheduler. The response time of *W1* is similar to that of *pClock*. This is because both *pClock* and WF2Q delay the violating requests which in turn affects the later requests.

4.4 Reduced Capacity Provisioning

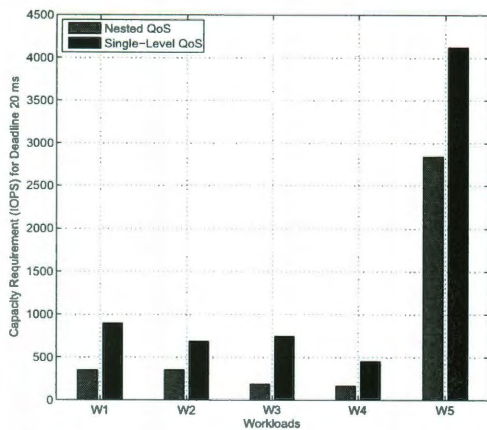
We now explore the relationship between capacity provisioning and performance. By profiling, the capacity required by different schedulers to achieve a certain QoS is determined empirically. We find that the Nested QoS scheduler provides significantly better performance at reduced capacity compared to the other schedulers. Our method reduces capacity using both decomposition and the EDF policy. The former reduces capacity by decomposing the workload and providing different response times for its badly behaved portions. EDF exploits the spare capacity arising from having



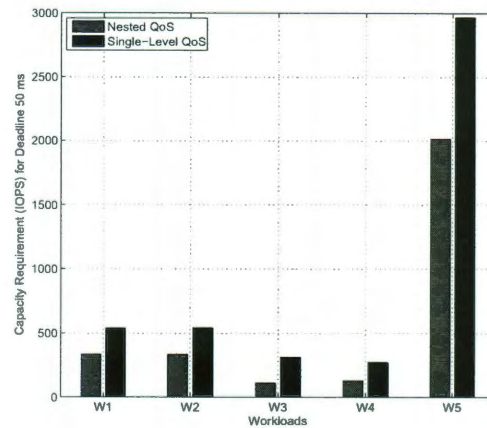
(a) Response time of 5ms



(b) Response time of 10 ms

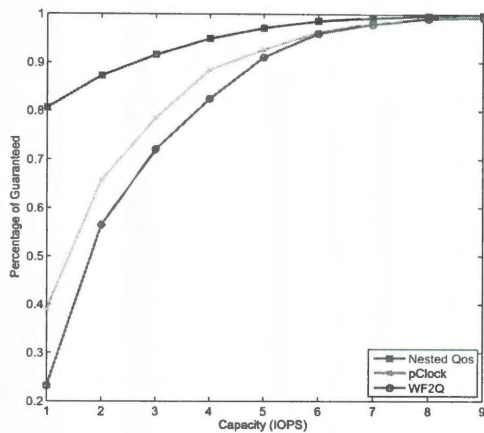


(c) Response time of 20 ms

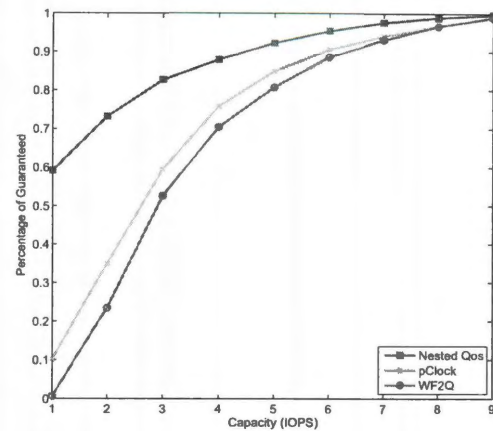


(d) Response time of 50 ms

Figure 4.8 : Reduced capacity requirements for different deadlines using Nested QoS



(a) Multiplexing similar traces



(b) Multiplexing dissimilar traces

Figure 4.9 : Reduced capacity requirements using Nested QoS

different deadlines for different clients.

We first evaluate the performance of the Nested QoS for a single workload. In this experiment, five workloads are used to evaluate Nested QoS separately. $W1$ and $W2$ are WebSearch workloads with a long term average arrival rate of 330 IOPS; $W3$ and $W4$ are Financial workloads with a long term average arrival rate of 100 IOPS; $W5$ is an Exchange Server workload with a long term average arrival rate of 910 IOPS. Our performance goal in this experiment is to have at least 90% of the requests meet a deadline of δ_1 , at least 95% of the requests meet a deadline of δ_2 , and all requests that satisfy the SLA face a maximum latency of δ_3 . Since $pClock$ and $WF2Q$ use a single-level QoS model, we set the performance goal to be a deadline of δ_1 for 100% of the workload. We compare the capacity requirements for the Nested QoS and the

single-level QoS for values of δ_1 equal to 5 ms, 10 ms, 20 ms and 50 ms. The capacity required in each case are shown in Figure 4.8. For all cases, Graduated QoS saves capacity significantly, while still providing comparable performance.

For the performance with multiple workloads, we first conducted experiments by multiplexing workloads of the same type. In this experiment, we use two *Exchange* workloads with deadlines of 50ms and 100ms respectively. We vary the server capacity from 2000 IOPS to 6000 IOPS and monitor the number of requests meeting their deadlines. Figure 4.9(a) shows the performance with the three schedulers. We can see that the Nested QoS can provide better performance guarantees than both *pClock* and WF2Q. As seen in Figure 4.9(a), with a capacity of 2000 IOPS, our method guarantees 80% of the workload while *pClock* and WF2Q can only guarantee 40% and 22% respectively. In order to achieve a 90% guarantee, our method requires about 2500 IOPS while *pClock* and WF2Q require about 3500 IOPS and 4000 IOPS respectively. The difference of *pClock* and WF2Q shows the benefit of EDF scheduling, while the gap between our method and WF2Q can be attributed to both the decomposition and EDF policy.

In the second experiment, we multiplex workloads of different types. Two *Exchange* workloads with deadlines of 50ms, and two Web Search *WS* workloads with deadlines of 100ms and one Financial Transaction *FinTran* workload with a deadline of 200ms are run concurrently. Figure 4.9(b) shows similar performance as Figure 4.9(a) for each scheduler.

4.5 Linux Implementations

We implement the Nested QoS framework as a Linux Loadable Module for Linux Kernel 2.6.32. The module builds a block device, which schedules requests using the Nested QoS algorithm, and sends requests to a physical backing device. The module also provides interfaces for users to set the parameters of the Nested QoS model: burst size, throughput and latency. A workload generator is also created to generate random IOs or replay IOs from a real workload trace file. The system is implemented in the Linux 2.6.32 kernel on a Dell Server with Intel(R) Core(TM)2 Quad 2.83GHz CPU and 4GB memory. The backing device is a 1TB Seagate SATA hard drive. The capacity of the hard drive is about 100 IOPS for random 4KB IO.

In this section, we will use several traces from MSRC storage to test the performance of the Nested QoS. We first examine the performance of the Nested QoS for a single workload. Then we apply the Nested QoS model to multiple workloads.

4.5.1 Single Workload

We first examine the performance of the Nested QoS for a single workload. The workload we used here is a *prxy* workload from Microsoft storage. The long term average throughput is 300 IOPS, and burst rate is up to 550 IOPS, as shown in Figure 4.10. The parameters for Nested QoS are (2, 100, 20ms), (3, 110, 100ms) and (5, 120, 1000ms), which show that the three latency levels are 20ms, 100ms, and 1000 ms. Because the backing device rate is about 100 IOPS, we slow down the arrival

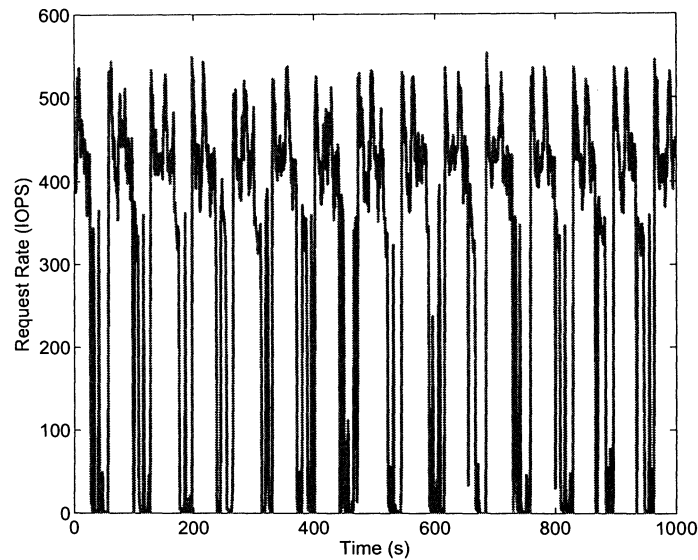


Figure 4.10 : Arrival rate of *prxy* workload

rate by a factor of 3.

Now we compare the latencies obtained by the Nested QoS and the pClock (same as the WF2Q for a single workload). Figure 4.11(a) shows the response time distribution and CDF using the Nested QoS and the pClock. As shown in Figure 4.11, the Nested QoS gets 76% of its requests finished in 20 ms, while pClock only gets 27% of its requests completed in 20 ms. The CDF of the response time is shown in Figure 4.11(b).

4.5.2 Multiple Workloads

Now we compare the performance of multiple workloads using the Nested QoS and the pClock. *prxy* is the workload we used in last section; *proj* is a workload from

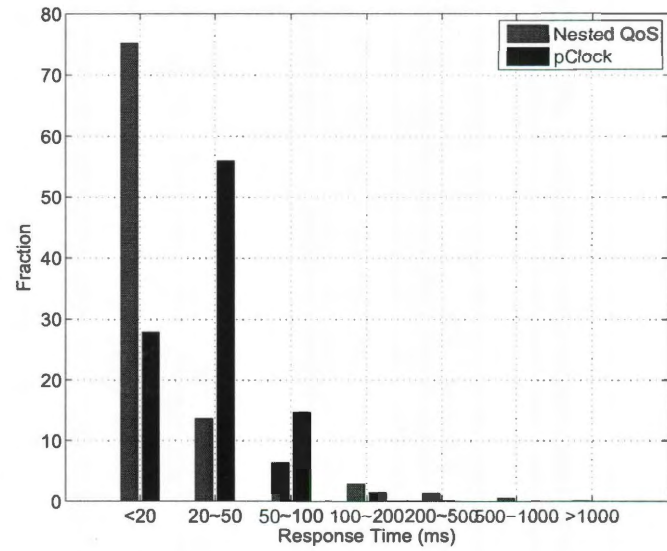
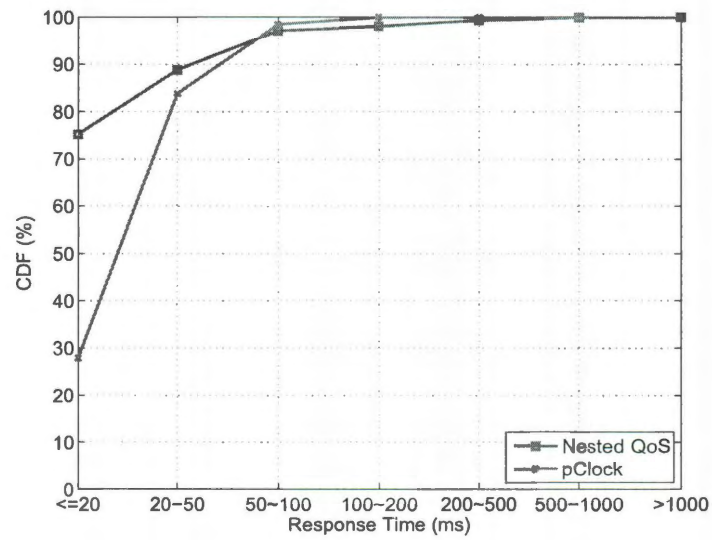
(a) Response time of *prxy*(b) CDF of response time of *prxy*

Figure 4.11 : Request response time distribution and CDF for *prxy* workload using Nested QoS and *pClock*

project server. The long term average throughput is 25 IOPS, and burst rate is up to 260 IOPS, as shown in Figure 4.12. We slow down the *prxy* by a factor of 4 and keep the *proj* at the original arrival rate. The parameters for the Nested QoS are (2, 100, 20ms), (3, 110, 200ms) and (5, 120, 1000ms) for *prxy*, and (2, 20, 20ms), (3, 22, 200ms) and (4, 25, 1000ms) for *proj*. Figure 4.13 and Figure 4.14 show the performance for *prxy*.

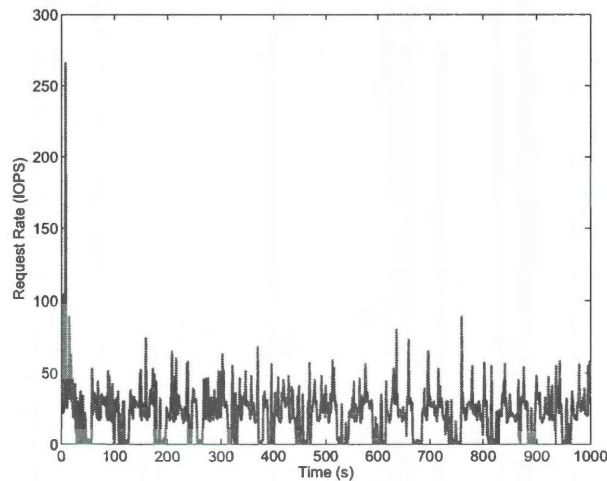


Figure 4.12 : Arrival rate of *proj* workload

As shown in the Figure 4.13, the Nested QoS gets 80% of requests finished in 20 ms for both the *prxy* and the *proj*, while for the pClock, less than 20% of its requests completed in 20 ms. The CDF of the response time is shown in Figure 4.14. This experiment shows that the Nested QoS is also suitable for multiple workloads, and can achieve performance isolation.

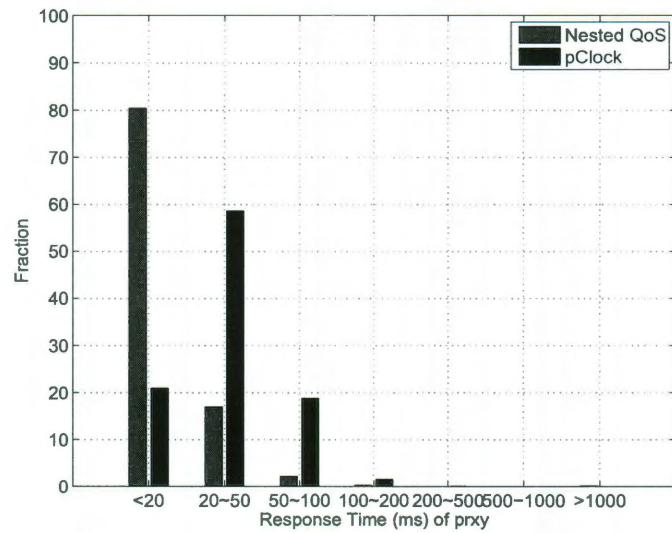
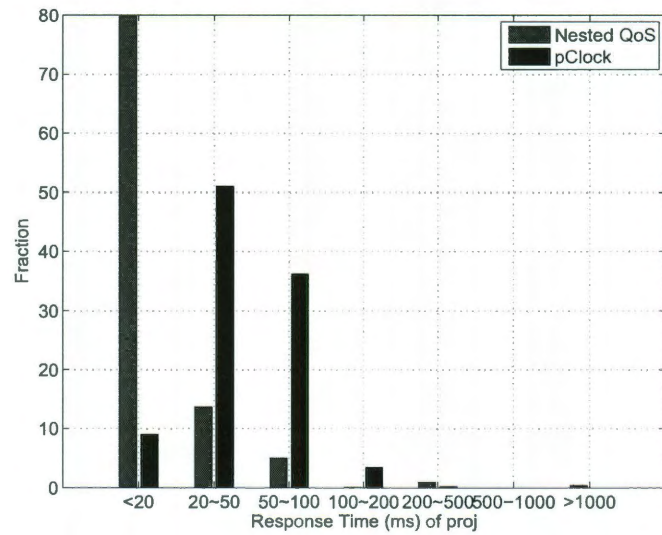
(a) *proxy* workload(b) *proj* workload

Figure 4.13 : Response time distribution for multiple workloads

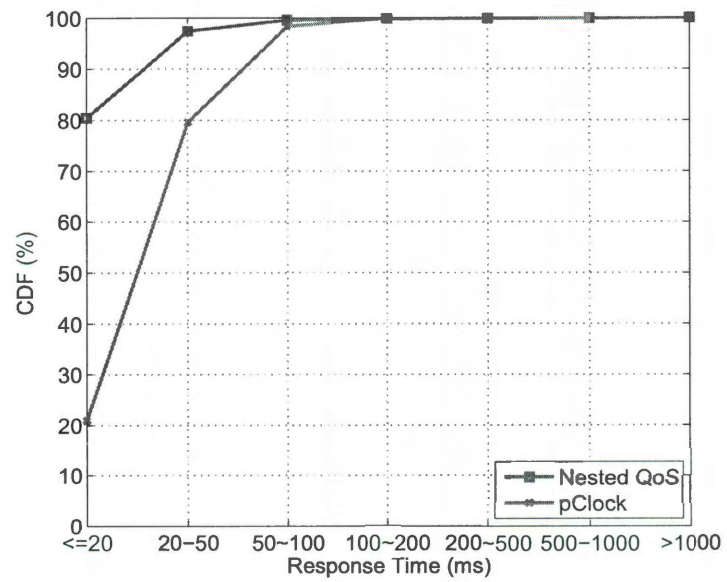
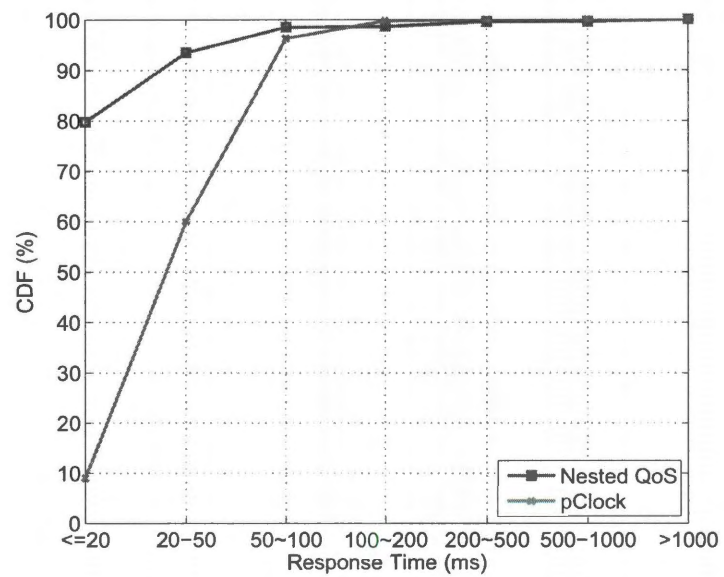
(a) *prxy* workload(b) *proj* workload

Figure 4.14 : Response time CDF for multiple workloads

4.6 Summary

In this chapter, we first use simulation to evaluate the Nested QoS model in three aspects: (1) contending workloads performance isolation, (2) local violation performance isolation, and (3) improved server provisioning. The results show that the Nested QoS model and framework can meet the three properties and get good performance. We also implemented a prototype in Linux, and demonstrated the working of the Nested QoS in an actual system.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis has proposed a novel Nested QoS model for providing performance SLAs in shared storage systems. The model allows defining flexible QoS based on the client's workload characteristics and willingness to pay, and it also allows the service provider to increase the degree of consolidation and to lower costs. A method of capacity provisioning for the Nested QoS is also provided. The model is easy to implement and enforce, and also easy to audit and arbitrate when disputes over SLAs violations between clients and service providers occur.

The thesis also provides a method to implement the Nested QoS model in storage systems. The scheduling framework based on the Nested QoS model employs two strategies together: systematically classifying workloads to provide each workload with a graduated QoS, and efficiently scheduling the classified requests of all the workloads. The results show that it achieves the three properties P1 to P3: isolation of different workloads from each other, isolation of the bursty portions of a single workload from its well behaved portions, and improved server utilization for small relaxations of the QoS.

5.2 Future Work

This thesis proposes a Nested QoS model for providing flexible performance SLAs in storage systems. However, there is one limitation which needs to be solved in the future.

The limitation is how to choose the parameters for the Nested QoS model. Currently, we use an off-line method to pick the (σ, ρ) in the model by profiling the workloads. In the future, we will improve this procedure and provide an online method to dynamically adjust the parameters based on the workload's properties.

Bibliography

- [1] “Ibm data centers.” <http://www-935.ibm.com/services/us/igs/smarterdatacenter.html?lnk=mhse>.
- [2] “Google cloud storage.” <http://code.google.com/apis/storage/>.
- [3] “Amazon simple storage service.” <http://aws.amazon.com/s3/>.
- [4] “Techtarget.” <http://searchitchannel.techtarget.com/>.
- [5] P. Goyal, H. M. Vin, and H. Cheng, “Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks,” *IEEE/ACM Trans. Netw.*, vol. 5, no. 5, pp. 690–704, 1997.
- [6] C. Lumb, A. Merchant, and G. Alvarez, “Façade: Virtual storage devices with performance guarantees,” *File and Storage technologies (FAST’03)*, pp. 131–144, March 2003.
- [7] A. Gulati, A. Merchant, and P. Varman, “pClock: An arrival curve based approach for QoS in shared storage systems,” in *ACM SIGMETRICS*, 2007.
- [8] H. Sariowan, R. L. Cruz, and G. C. Polyzos, “Scheduling for quality of service guarantees via service curves,” in *Proceedings of the International Conference on*

Computer Communications and Networks, pp. 512–520, 1995.

- [9] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron, “Everest: Scaling down peak loads through I/O off-loading,” in *Proceedings of OSDI*, 2008.
- [10] A. Riska and E. Riedel, “Long-range dependence at the disk drive level,” in *Proceedings of QEST*, 2006.
- [11] M. E. Gómez and V. Santonja, “On the impact of workload burstiness on disk performance,” in *Workload characterization of emerging computer applications*, 2001.
- [12] L. A. Barroso and U. Hölzle, “The case for energy-proportional computing,” *Computer*, vol. 40, pp. 33–37, December 2007.
- [13] A. Gulati, C. Kumar, and I. Ahmad, “Storage workload characterization and consolidation in virtualized environments,” in *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT '09)*, 2009.
- [14] J. C. R. Bennett and H. Zhang, “ WF^2Q : Worst-case fair weighted fair queueing,” in *INFOCOM (1)*, pp. 120–128, 1996.
- [15] J. Turner, “New directions in communications,” *Communications Magazine*, *IEEE 24 (10): 8C15. ISSN 0163-6804/1986*.
- [16] D. Sacks, “Demystifying storage networking,” *Technical Report, IBM*, 2001.
- [17] W. C. Preston, *Using SANs and NAS*. O’Reilly Media, 2002.

- [18] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, pp. 344–357, June 1993.
- [19] A. G. Greenberg and N. Madras, "How fair is fair queuing," *J. ACM*, vol. 39, no. 3, pp. 568–598, 1992.
- [20] R. L. Cruz, "Quality of service guarantees in virtual circuit switched networks," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 6, pp. 1048–1056, 1995.
- [21] S. Golestani, "A self-clocked fair queueing scheme for broadband applications," in *INFOCOMM'94*, pp. 636–646, April 1994.
- [22] S. Suri, G. Varghese, and G. Chandramenon, "Leap forward virtual clock: A new fair queueing scheme with guaranteed delay and throughput fairness," in *INFOCOMM'97*, April 1997.
- [23] A. Gulati, A. Merchant, and P. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," in *USENIX OSDI*, 2010.
- [24] A. K. Parekh and R. G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Trans. Netw.*, vol. 1, no. 3, pp. 344–357, 1993.
- [25] A. K. Parekh and R. G. Gallagher, "A generalized processor sharing approach to

- flow control in integrated services networks: the multiple node case,” *IEEE/ACM Trans. Netw.*, vol. 2, no. 2, pp. 137–150, 1994.
- [26] J.-Y. Le Boudec and P. Thiran, *Network calculus: a theory of deterministic queuing systems for the internet*. Berlin, Heidelberg: Springer-Verlag, 2001.
- [27] K. I. Park, *QoS In Packet Networks*. USA: Springer, 2005.
- [28] S. Floyd and V. Jacobson, “Random early detection gateways for congestion avoidance,” *IEEE/ACM Trans. Netw.*, vol. 1, pp. 397–413, Aug. 1993.
- [29] E. W. Knightly and H. Zhang, “D-bind: An accurate traffic model for providing QoS guarantees to VBR traffic,” *IEEE/ACM Transactions on Networking*, vol. 5, pp. 219–231, 1997.
- [30] L. Lu, K. Doshi, and P. Varman, “Workload decomposition for QoS in hosted storage services,” in *MW4SOC*, 2008.
- [31] L. Lu, K. Doshi, and P. Varman, “Graduated QoS by decomposing bursts: Don’t let the tail wag your server,” in *29th IEEE International Conference on Distributed Computing Systems*, 2009.
- [32] H. Wang and P. Varman, “Statistical workload shaping for storage systems,” in *HiPC’09*, pp. 274–283, 2009.
- [33] H. Wang, K. Doshi, and P. J. Varman, “Nested QoS: Adaptive burst decomposition for slo guarantees in virtualized servers,” *Intel Technology Journal Vol 16*

issue 2 Exploring Control and Autonomic Computing ISBN 978-1-934053-48-5,
June 2012.

- [34] J. R. Jump, “Yacsim reference manual.” <http://www.owl.net/rice.edu/elec428/yacsim/yacsim.man.ps>.
- [35] “Storage Performance Council (UMass Trace Repository),” 2007.
<http://traces.cs.umass.edu/index.php/Storage>.